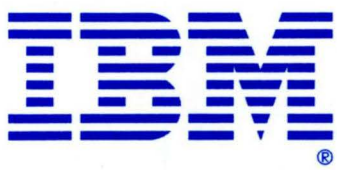
Volume 5, Number 4 May 1987

IBM Personal System/2™ Models 50, 60, 80
VGA, BIOS and Programming Considerations

IBM Personal System/2™ Seminar Proceedings

The Publication for Independent Developers of Products for IBM Personal System/2

**Published by International Business Machines Corporation
Entry Systems Division**



Changes are made periodically to the information herein; any such changes may be reported in subsequent Proceedings.

It is possible that this material may contain reference to, or information about IBM products (machines and programs), programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such products, programming or services in your country.

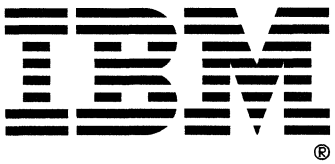
This publication could contain technical inaccuracies or typographical errors. Also, illustrations contained herein may show prototype equipment. Your system configuration may differ slightly. IBM believes the statements contained herein are accurate as of the date of publication of this document. However, IBM makes no warranty of any kind with respect to the accuracy or adequacy of the contents hereof.

This information is not intended to be a statement of direction or an assertion of future action. IBM expressly reserves the right to change or withdraw current products that may or may not have the same characteristics or codes listed in this publication. Should IBM modify its products in a way that may affect the information contained in this publication, IBM assumes no obligation whatever to inform any user of the modification(s).

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

All specifications are subject to change without notice.

Printed in the
United States
of America



Contents

Foreword

IBM Personal System/2™ Seminars and Proceedings provide information about new product announcements and enhancements to existing products, and are intended to assist independent developers in their hardware and software development efforts.

Over the past several years, the success of the IBM Personal Computer family was due in part to the efforts of independent developers, whose hardware and software products have become widely used. For its part, IBM helped these vendors by holding relevant technical seminars and publishing the proceedings of those seminars. The result was a mutually beneficial partnership and transfer of technical knowledge.

With the advent of the Personal System/2 family, IBM's seminar program will continue. Through these seminars and the corresponding proceedings, IBM will address the independent developers' need for technical information about the latest IBM products. In these and future proceedings, you will find technical information about subjects such as:

- IBM computer design and architecture
- IBM computer components and their interaction
- Memory capacities, speeds, transfer rates
- Input/output device capacities, speeds, access methods and rates
- Graphics and display technologies, programming considerations
- Printing technologies, programming considerations
- Operating system high level interfaces
- Development tools: capabilities, languages, program verification aids
- Compatibility considerations
- Communications: capabilities, offerings, statistics
- Enhancements to existing IBM hardware and software products
- Hints, tips and techniques to enhance your productivity

Through these seminars and proceedings, IBM intends to maintain its partnership with independent developers and assist them in successfully producing hardware and software products for the IBM Personal System/2 family.

Personal System/2 is a trademark of the IBM Corporation.

Personal System/2 Video

The video on the IBM Personal System/2 Models 50, 60 and 80 is generated by the IBM Video Graphics Array (VGA) chip and its associated circuitry (see Figure 1 on page 3). The associated circuitry consists of the video memory and a video Digital-to-Analog Converter (DAC). The 256K bytes of video memory are formed from four 64K x 8 memory maps. The red, green, and blue (RGB) outputs from the video DAC drive any of the IBM 31.5 kHz analog displays. Hereafter, the VGA chip and its associated circuitry are referred to as "VGA."

All video modes available on the IBM Monochrome Adapter, IBM Color/Graphics Adapter, and IBM Enhanced Graphics Adapter are supported, regardless of which analog display is connected. All VGA modes are available on all of the supported analog displays. Colors are displayed as shades of grey when the monochrome analog display is connected.

New Modes

New modes available are: 640 x 480 graphics in both 2 and 16 colors; 720 x 400 alphanumeric in both 16-color and monochrome; 360 x 400 16-color alphanumeric; and 320 x 200 graphics with 256 colors. In addition, all 200-line modes are double-scanned by VGA and displayed as 400 lines on the display. This means that each one-pixel-high horizontal scan line will be displayed twice on the display, providing improved legibility.

The VGA chip does the interfacing between the CPU and video memory. All data passes through the VGA chip when the CPU writes to or reads from video memory. The VGA chip controls the arbitration for video memory between the CPU and the CRT Controller function contained within the VGA chip. The user does not need to write to the display buffer during non-active display time to prevent snow on the screen; the VGA chip automatically prevents this from happening. The CPU will experience better performance when accessing the display buffer during non-active display times, because less interference from the CRT Controller function is occurring.

Video memory addressing is controlled by the VGA chip. The starting address of the video memory is programmable to three different starting addresses for compatibility with previous video adapters. BIOS will program the VGA chip appropriately during a video mode set.

Character Sets

In alphanumeric modes, the CPU writes ASCII character code and attribute data to video memory maps 0 and 1 respectively. The character generator is stored in video memory map 2 and is loaded by BIOS during an alphanumeric video mode set. BIOS downloads the character set font generator data from system ROM. Three fonts are contained in ROM. Each of these fonts contains dot patterns for 256 different characters. Two of the fonts are identical to those provided by the IBM Monochrome Display Adapter, the IBM Color/Graphics Adapter, and the IBM Enhanced Graphics Adapter. The third font is a new 9 x 16 character font. Up to eight 256-character fonts can be loaded into video memory map 2 at one time (EGA allows up to four). A BIOS interface exists to load user-defined fonts. As on EGA, BIOS calls select which of the fonts is actually used to form characters and to redefine the intensity bit in the attribute byte as a switch between two 256-character fonts. This allows 512 characters to be displayed on the screen at one time.

Video DAC

The VGA chip formats the information stored in video memory into an 8-bit digital value that is sent to the video Digital-to-Analog Converter (DAC). This 8-bit value allows access to a maximum of 256 registers inside the video DAC. For example, in the 2-color graphics modes, only two different 8-bit values would be presented to the video DAC; in the 256-color graphics mode 256, different 8-bit values would be presented to the video DAC. Each register inside the video DAC contains a color value that is selected from a choice of 256K colors; therefore, each color displayed on the screen is selected from a choice of 256K colors.

The DAC outputs three analog color signals (red, green, and blue) which are sent to the display's 15-pin connector. The monochrome analog display is concerned with only the green analog output, which is used to determine the shade of grey that will be displayed.

Video Enable/Disable

A BIOS call is used to enable/disable VGA. Disable means that VGA will not respond to video memory or I/O reads or writes. The contents of registers and video memory are preserved with the values present when the disable is invoked; VGA will continue to generate valid video output if it was doing so before it was disabled.

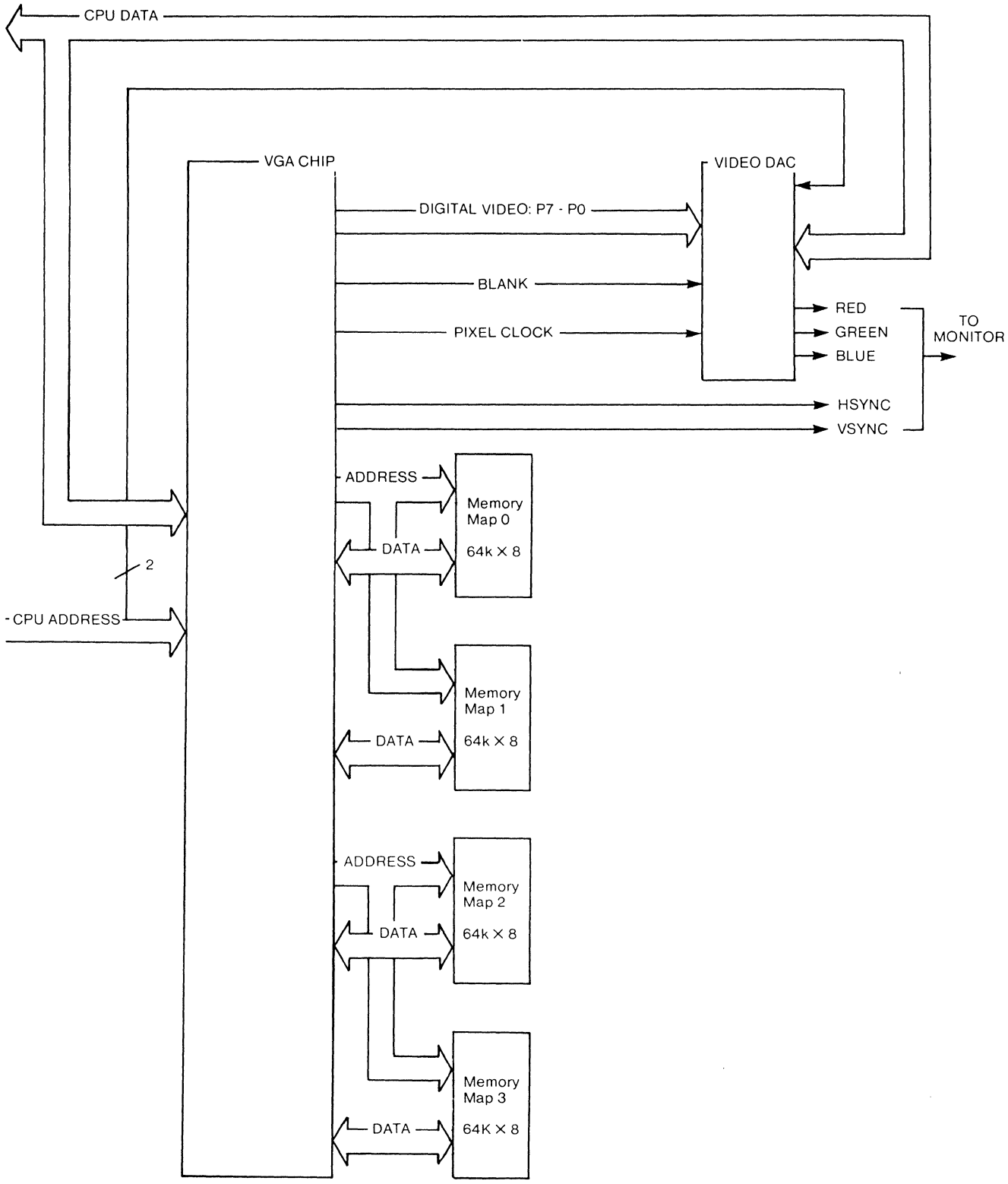


Figure 1. Video Subsystem

Personal System/2 Video Modes

The following table describes the video modes supported by BIOS:

MODE #	TYPE	COLORS	ALPHA FORMAT	BUFFER START	BOX SIZE	MAX. PAGES	VERTICAL FREQ.	RESOLUTION	DOUBLE SCAN ?	BORDER ?
0, 1	A/N	16/256K	40 × 25	B8000	8 × 8	8	70 Hz	320 × 200	YES	NO
2, 3	A/N	16/256K	80 × 25	B8000	8 × 8	8	70 Hz	640 × 200	YES	YES
0*, 1*	A/N	16/256K	40 × 25	B8000	8 × 14	8	70 Hz	320 × 350	NO	NO
2*, 3*	A/N	16/256K	80 × 25	B8000	8 × 14	8	70 Hz	640 × 350	NO	YES
0+, 1+	A/N	16/256K	40 × 25	B8000	9 × 16	8	70 Hz	360 × 400	NO	NO
2+, 3+	A/N	16/256K	80 × 25	B8000	9 × 16	8	70 Hz	720 × 400	NO	YES
4, 5	APA	4/256K	40 × 25	B8000	8 × 8	1	70 Hz	320 × 200	YES	NO
6	APA	2/256K	80 × 25	B8000	8 × 8	1	70 Hz	640 × 200	YES	YES
7	A/N	-	80 × 25	B0000	9 × 14	8	70 Hz	720 × 350	NO	YES
7+	A/N	-	80 × 25	B0000	9 × 16	8	70 Hz	720 × 400	NO	YES
D	APA	16/256K	40 × 25	A0000	8 × 8	8	70 Hz	320 × 200	YES	NO
E	APA	16/256K	80 × 25	A0000	8 × 8	4	70 Hz	640 × 200	YES	YES
F	APA	-	80 × 25	A0000	8 × 14	2	70 Hz	640 × 350	NO	YES
10	APA	16/256K	80 × 25	A0000	8 × 14	2	70 Hz	640 × 350	NO	YES
11	APA	2/256K	80 × 30	A0000	8 × 16	1	60 Hz	640 × 480	NO	YES
12	APA	16/256K	80 × 30	A0000	8 × 16	1	60 Hz	640 × 480	NO	YES
13	APA	256/256K	40 × 25	A0000	8 × 8	1	70 Hz	320 × 200	YES	YES

Figure 2. BIOS Video Modes

Modes 0 through 6 emulate the support provided by the IBM Color/Graphics Adapter (CGA). Mode 7 emulates the support provided by the IBM Monochrome Display Adapter (MDA). Modes D, E, F, 0*, 1*, 2*, 3*, and 10 emulate the support provided by the IBM Enhanced Graphics Adapter (EGA).

When a color analog display is used, each color is selected from a choice of 256K colors. When the monochrome analog display is used, each color is displayed as a shade of grey and selected from a choice of 64 shades.

All Modes Supported on All Displays

Previous adapters have required that a video mode's corresponding display be attached. For example, EGA requires that the Enhanced Color Display be attached to run mode 3* and the Monochrome Display to run mode 7. All Personal System/2 video modes are available on all the supported analog displays. Colors are displayed as shades of grey when the monochrome analog display is connected. Circuitry on the Personal System/2 system board detects which type of analog display is connected (color or monochrome). BIOS maps (sums) the colors into shades of grey. See the BIOS Interface Manual for more information on summing.

Auxiliary Video Connector

The Auxiliary Video Connector (AVC) is a 20-pin connector located in line with one of the Micro Channel Connectors on the system board. This connector allows video data from VGA to be passed to an option card, or allows the system board video buffers to be turned off and video from the option card to drive the video DAC and the 15-pin output connector that drives the analog display. The full Micro Channel is available for use by the option card.

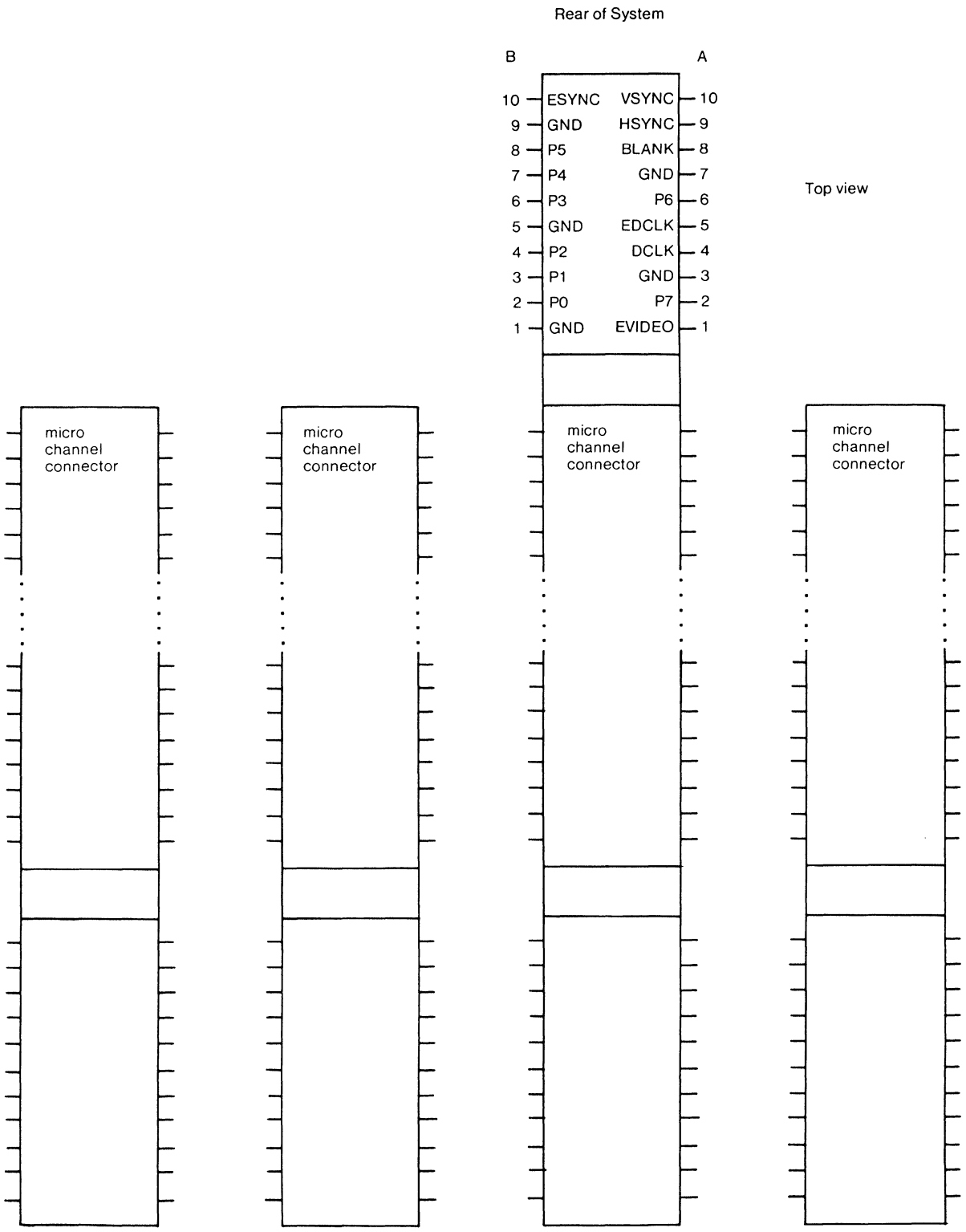


Figure 3. Auxiliary Video Connector Pinout

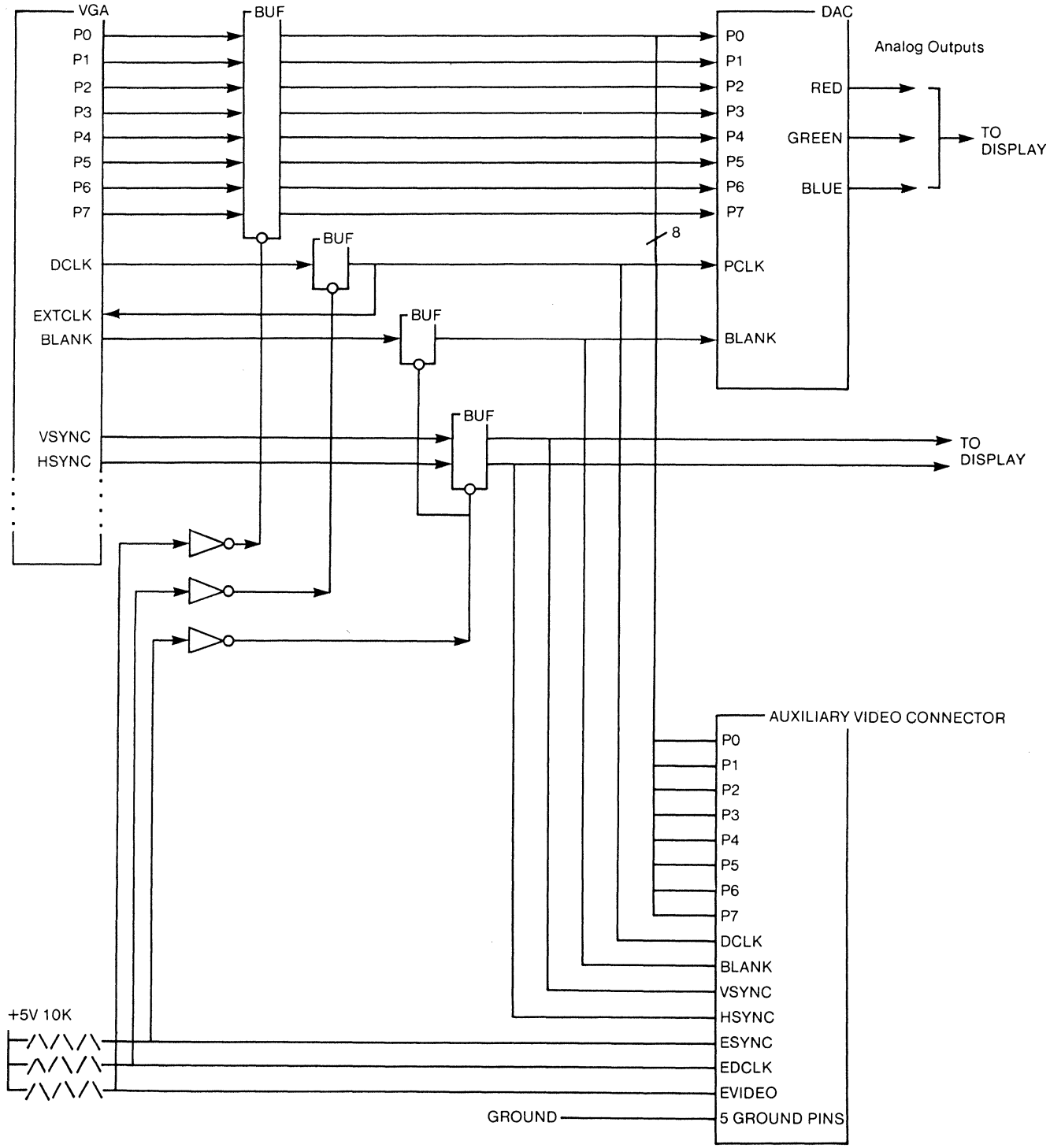


Figure 4. Auxiliary Video Connector Block Diagram

AVC Signal Descriptions

VSYNC

This signal is the vertical sync signal used to drive the display. See also the ESYNC signal description.

HSYNC

This signal is the horizontal sync signal used to drive the display. See also the ESYNC signal description.

BLANK

This signal is connected to the BLANK input of the video DAC and when active (= 0V) tells the DAC to drive its analog color outputs to 0V. See also the ESYNC signal description.

P7 - P0

These eight signals contain digital video information and comprise the pixel address inputs to the video DAC. See also the EVIDEO signal description.

DCLK

This signal is the video pixel clock that is used by the video DAC to latch the digital video signals (P7 - P0). P7 - P0 are latched on the rising edge of DCLK inside the DAC.

This signal is also connected to the EXTCLK input of the VGA chip, and may be driven by the AVC and used as the input clock to the VGA chip. When this configuration is used, *the VGA chip may not be the source of the digital video signals presented to the DAC (P7 - P0); rather, P7 - P0 must be driven from the AVC*. See also the EDCLK signal description.

ESYNC

This signal is the output enable signal for the buffer that drives the BLANK, VSYNC, and HSYNC signals. ESYNC is tied to +5V through a pull-up resistor so that an open circuit on the ESYNC pin produces +5V.

When the ESYNC signal = +5V, BLANK, VSYNC, and HSYNC are sourced from the VGA chip's BLANK, VSYNC, and HSYNC outputs respectively. When the ESYNC signal = 0V, BLANK, VSYNC, and HSYNC are driven from the AVC.

EVIDEO

This signal is the output enable signal for the buffer that drives the P7 - P0 signals. EVIDEO is tied to +5V through a pull-up resistor so that an open circuit on the EVIDEO pin produces +5V.

When the EVIDEO signal = +5V, P7 - P0 are sourced from the VGA chip's P7 - P0 outputs. When the EVIDEO signal = 0V, P7 - P0 are driven from the AVC.

EDCLK

This signal is the output enable signal for the buffer that drives the DCLK signal. EDCLK is tied to +5V through a pull-up resistor so that an open circuit on the EDCLK pin produces +5V.

When the EDCLK signal = +5V, the DCLK signal is sourced from the VGA chip's DCLK output, and received by the AVC and DAC.

When EDCLK = 0V, DCLK is driven from AVC to the EXTCLK input of the VGA chip and to the DAC. When this configuration is used, the VGA chip may not be the source of the digital video signals presented to the DAC (P7 - P0); rather P7 - P0 must be driven from the AVC. If EXTCLK is to be used as the input clock for the VGA chip, the Miscellaneous Output Register in the VGA chip must be programmed to select clock source 2.

Color Mapping

The Enhanced Graphics Adapter (EGA) allows a maximum of 16 different colors to be displayed at one time. EGA contains palette registers that allow each color to be selected from a choice of 64 possible colors. The six bits necessary to form the 64 possible colors comprise the information sent to the digital display connected to the EGA card.

A specific example is shown:

Consider 640 x 350 graphics mode (Mode hex 10). Assume the EGA card is being used and an Enhanced Color Display is attached. This means the six digital color signals being driven to the display from the EGA card are defined as (MSB)R'B'G' RGB(LSB). Assume that palette register hex A has been programmed to have a value of hex 3. If a dot on the screen has attribute hex A (Light Green), then the 4-bit attribute hex A from video memory will select palette register hex A. The contents of register hex A, hex 3A, are sent to the display. Note that the

primary and secondary green bits are turned on, but only the secondary bits for red and blue (thus producing light green on the display):

Color Mapping

	R′	B′	G′	R	G	B
$3A_{HEX}$ =	1	1	1	0	1	0_{BINARY}

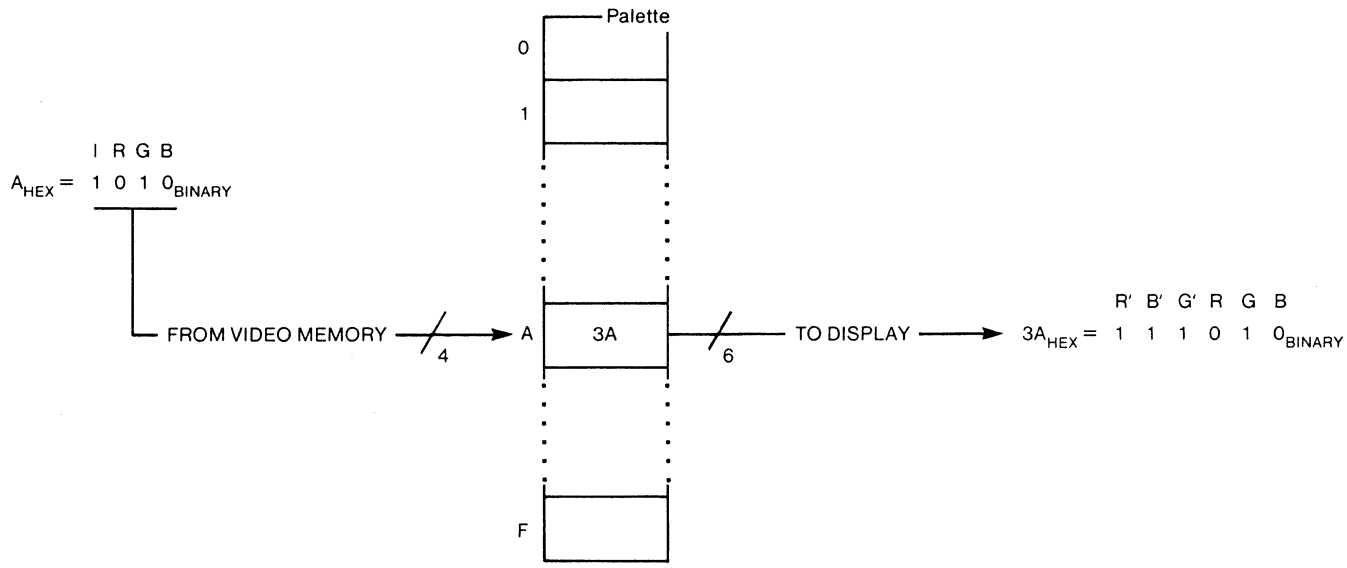


Figure 5. EGA Color Mapping Example

The VGA video system allows a maximum of 256 different colors to be displayed at one time. The VGA video system contains a video Digital-to-Analog Converter (DAC) that also contains color look-up table (CLUT) registers. The VGA chip formats information stored in video memory into an 8-bit digital value that is sent to the video DAC. This 8-bit value selects one of the CLUT registers, each of which is 18 bits wide. The 18 bits are broken down into three 6-bit fields, one each for red, green, and blue. This allows each color displayed to be chosen from a choice of 256K colors. The video DAC converts the 18-bit value contained in the CLUT register to three analog signals (red, green, and blue) that are sent to the analog display.

Assume the same application described previously is running on VGA. Assume the Color Select Register contains 0, and CLUT register hex 3A contains hex

15, hex 3F, and hex 15 in its red, green, and blue fields, respectively. The 4-bit attribute hex A from video memory will select palette register hex A as on EGA. Two bits from the Color Select Register are appended to the six-bit value from palette register hex A to form the 8-bit value sent to the video DAC. Because the Color Select Register contains 0, the value sent to the DAC is hex 3A. The CLUT register hex 3A in the DAC is accessed, which causes its contents to be sent to the digital-to-analog converters. Since the CLUT register hex 3A contains hex 15, hex 3F, and hex 15 in its red, green, and blue fields, respectively, the value sent to the analog display will be light green.

The mechanism described above functions in a similar fashion for all other video modes *except* mode hex 13 (256 color 320 x 200 Graphics). In this mode the palette registers are programmed so that the 8-bit attribute stored in video memory is sent intact to the video DAC. Do not modify the palette registers in this mode.

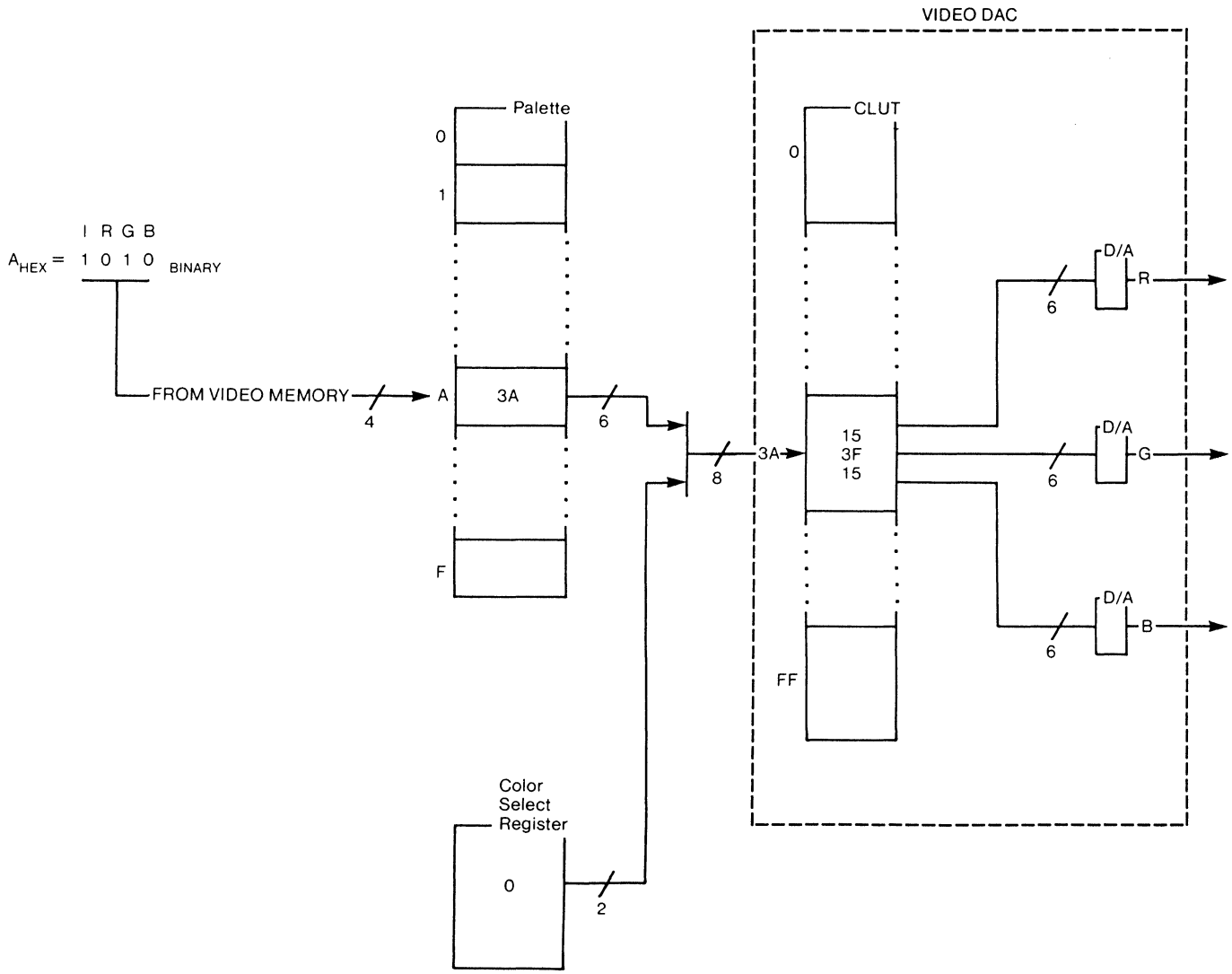
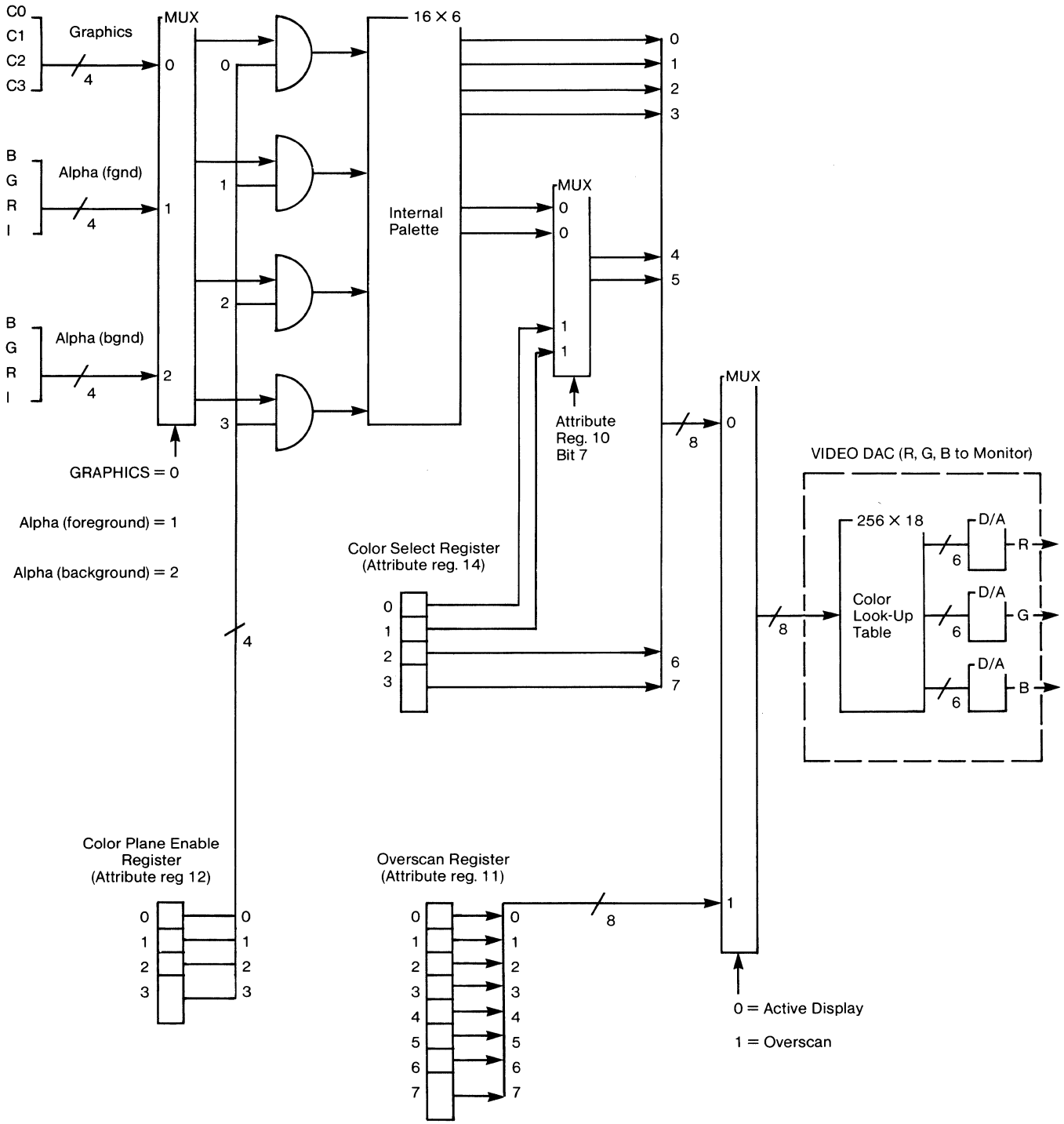


Figure 6. VGA Color Mapping Example



*Except 256 color mode: The 8-bit attribute stored in memory is sent to the DAC. DO NOT MODIFY INTERNAL PALETTE IN THIS MODE.

Figure 7. VGA Color Mapping*

VGA Support Functions

Horizontal Pel Panning

Horizontal pel panning allows the programmer to shift the video image one or more pel positions to the left on the analog display. The horizontal pel panning register in the attribute section of the VGA chip allows the video image to be panned up to 8, 7 or 3 pel positions depending on the video mode. In modes that use 9 pels per character box (0+, 1+, 2+, 3+, 7, 7+) the video image can be panned up to 8 pel positions using the pel panning register. In all other modes, except mode hex 13, the pel panning register will provide for panning of up to 7 pel positions. 3 pel positions can be panned in mode hex 13. Continuous pel panning can be achieved by incrementing the pel panning register (once each vertical retrace period) until the maximum value for a given mode is reached, then resetting pel panning to zero and incrementing the display start address in the CRT controller subsection of the VGA chip. As the video image is panned to the left, data will be panned into the video image from the right. The data panned into the image will be that which is immediately following the normal end of line data in video memory. This means that if the data following the normal end of line data is the start of the next scan line (as is the default for modes set by BIOS), then data panned onto the right side of the display will come from the left side of the next lower graphics scan line or alphanumeric character line. This wrap-back effect can be avoided by using the logical line width register in the CRT controller subsection to provide space between scan lines in video memory. Normally, the logical line width register is programmed with the scan line character width. Increasing this value will provide extra memory bytes between each line. Note that BIOS will not support this feature because individual character, and pel addressing will be changed.

Smooth Scrolling

The VGA chip provides support for vertical smooth scrolling in alphanumeric modes via the preset row scan register in the CRT controller subsection. This register determines which scan line of the first character row begins the display. When set to zero (the default value), the display begins with the top (zero) scan line of each character box on the first row, displaying each character in its entirety. When programmed to a non-zero value, the display starts somewhere in the middle of each character box, showing some lower portion of the characters. Vertical smooth scrolling can be achieved by stepping the preset row scan register (once each vertical retrace period) from zero to one less than the character box height. The second character row can then be smooth scrolled into the first character row by resetting the preset row scan register to zero, and loading the start address registers with the address of the second character row. The appearance of vertical smooth scrolling is that the top row of characters moves up and is scrolled off the top of the display, while a new row of characters is scrolled onto the bottom of the display. The new row of characters is the character row that follows the original last row in video memory.

Split Screen

The VGA chip supports a dual screen display. The top portion of the display is designated as screen A, and the bottom portion of the display is designated as screen B as shown in Figure 8.

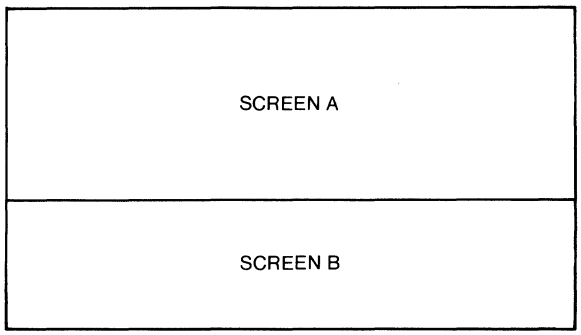


Figure 8. Dual Screen Definition

Figure 9 on page 12 shows the screen mapping for a system containing a 32K byte alphanumeric storage buffer. The VGA Video Subsystem has a 32K byte storage buffer in alphanumeric mode. Information displayed on screen A is defined by the start address high and low registers (hex 0C and hex 0D) of the CRT Controller subsection of the VGA chip. Information displayed on screen B always begins at address hex 0000. Even though this example is for an alphanumeric mode, a split screen is possible for graphics modes also.

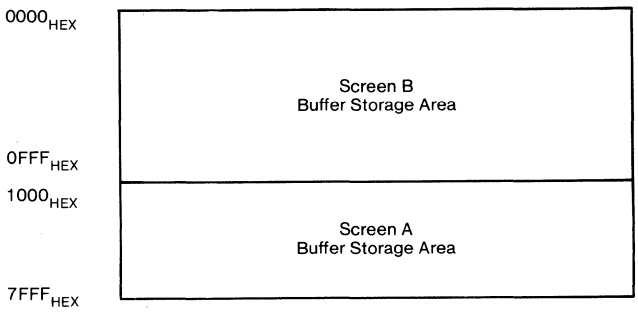


Figure 9. Screen Mapping within the Display Buffer Address Space

The Line Compare Register of the CRT Controller subsection is utilized to perform the split screen function. The CRT controller subsection has an internal horizontal scan line counter, and logic which compares the horizontal scan line counter value to the Line Compare Register value and clears the memory address generator when a compare occurs. The linear address generator then sequentially addresses the display buffer starting at location zero.

Screen B can be smoothly scrolled onto the display by updating the Line compare in synchronization with the vertical retrace signal. Note that in video modes that are double scanned (modes 0, 1, 2, 3, 4, 5, 6, hex D, hex E, and hex 13), the line compare register should be programmed with even numbers only. The information on screen B is immune from scrolling operations which utilize the Start Address registers to scroll through the Screen A address map, and from operations which use the preset row scan register.

The pel panning compatibility bit in the attribute mode control register determines whether or not horizontal pel panning operations on screen A will affect screen B. If this bit is logical zero, then screen B will pan with screen A. If this bit is logical one, then screen B will be immune to the panning operation of screen A.

Logical Memory Read and Write Modes

The VGA chip provides various memory read and write modes that relieve some of the burden of graphics memory manipulation. Two read and four write modes are available. These are described below.

Read Modes

There are two ways to do video memory reads. When read type 0 is selected using the Graphics Mode Register, the video memory returns to the CPU the 8-bit value determined by the logical decode of the memory address, and the Read Map Select Register if applicable. When read type 1 is selected using the Graphics Mode Register, the 8-bit value returned will be the result of the color compare operation controlled by the Color Compare and Color Don't Care Registers. The data flow for the color compare operations is illustrated in Figure 10 on page 13.

The color compare logic is designed for use in modes that use the video memory maps as bit planes. In mode hex 12, for example, maps 3, 2, 1, and 0 are intensity, red, green, and blue bit planes respectively. This means that a byte in a given map contains that bit plane's value for 8 consecutive PELs on the display. For instance the first byte of map 0 would contain the blue bit for the first 8 PELs on the display.

The color compare logic compares a byte from each map with that map's corresponding bit in the color compare register. Each bit in a map's byte is compared with the corresponding color compare bit, yielding an 8-bit intermediate result that is a 1 in each bit position where a match is found. If a map's corresponding color don't care register bit is a 0, that map's 8-bit intermediate result is forced to all 1's, indicating valid compares in all bit positions. The four intermediate results (one for each map) are compared in a bit-wise fashion, returning an 8-bit result to the CPU that is a 1 in each bit position where all four maps have a match.

For example, in mode hex 12, assume the Color Compare Register contains 0101_{Binary}, the Color Don't Care Register contains 0111_{Binary}, and maps 2, 1, and 0 contain 01111111_{Binary}, 01000000_{Binary}, and 11011111_{Binary} respectively. The intermediate results for maps 3, 2, 1, and 0 would be 11111111_{Binary}, 01111111_{Binary}, 10111111_{Binary}, and 11011111_{Binary} respectively. The value returned to the CPU would be 00011111_{Binary}.

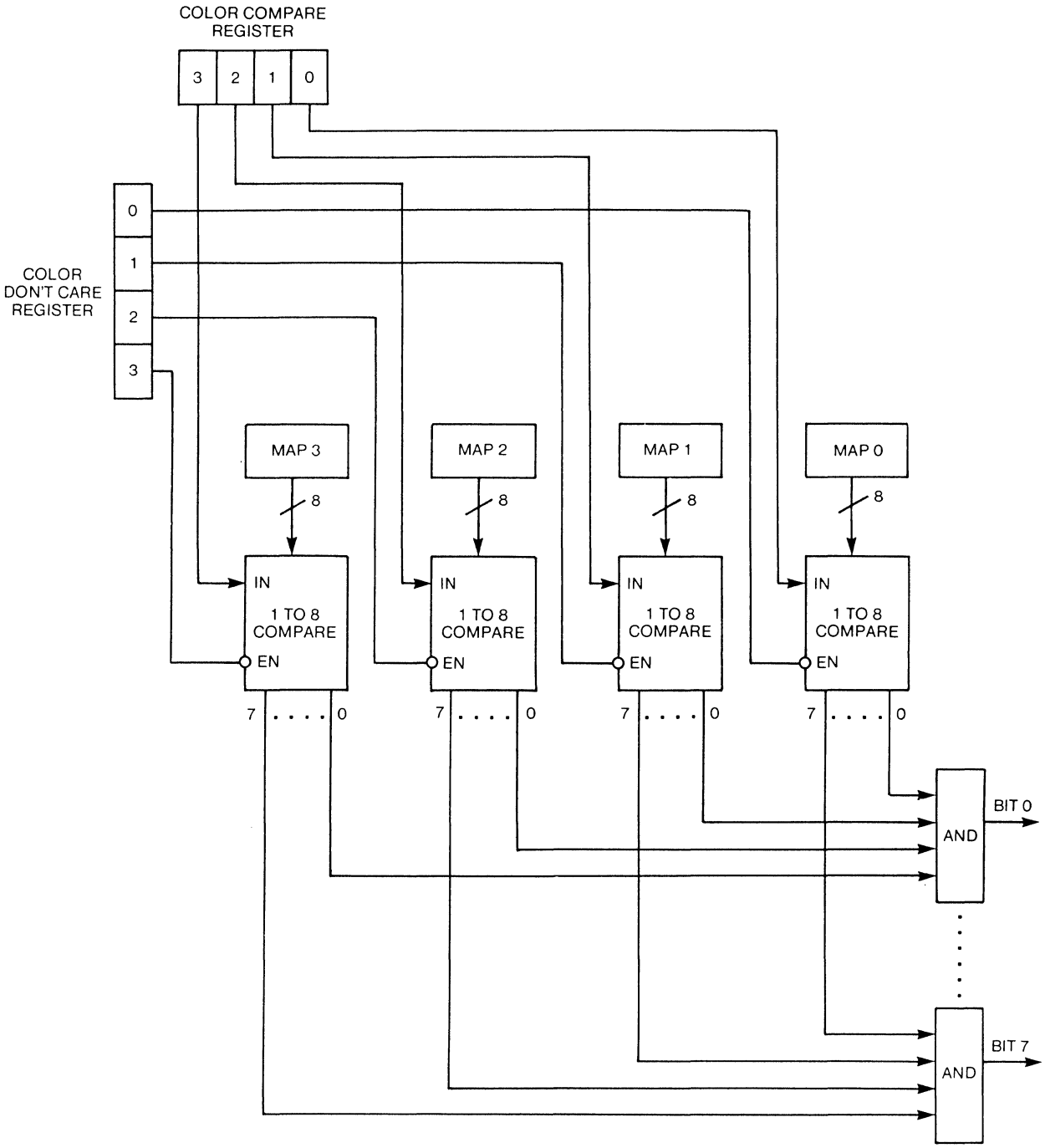


Figure 10. VGA Color Compare Operations

Write Modes

One of the four available write modes is selected by programming the Graphics Mode Register. Figure 11 on page 15 describes the data flow for write operations and is referred to in the following paragraphs. As with the color compare logic, the write logic is designed to support modes that use the video memory maps as bit planes. In mode hex 12 for example, maps 3, 2, 1, and 0 are the intensity, red, green, and blue bit planes respectively. This means that a byte in a given map contains that bit plane's value for 8 consecutive PELs on the display. For instance, the first byte of map 0 would contain the blue bit for the first 8 PELs on the display.

Write Mode 00: In write mode 00_{Binary}, the 8 bits of incoming CPU data are right rotated by the number of bits specified in the Data Rotate Register, with the old least significant bit becoming the new most significant bit. If the set/reset function is disabled for a map, this rotated 8-bit value is sent to the logic

function unit for that map. If the set/reset function is enabled for a map, the bit contained in the Set/Reset Register for that map is expanded to 8 bits and that value is sent to the logic function unit for that map. (Expanded means that the set/reset bit is repeated 8 times; if the set/reset bit is 1, its expansion is 11111111_{Binary}.) The set/reset function is enabled by setting a map's corresponding bit = 1 in the Enable Set/Reset Register.

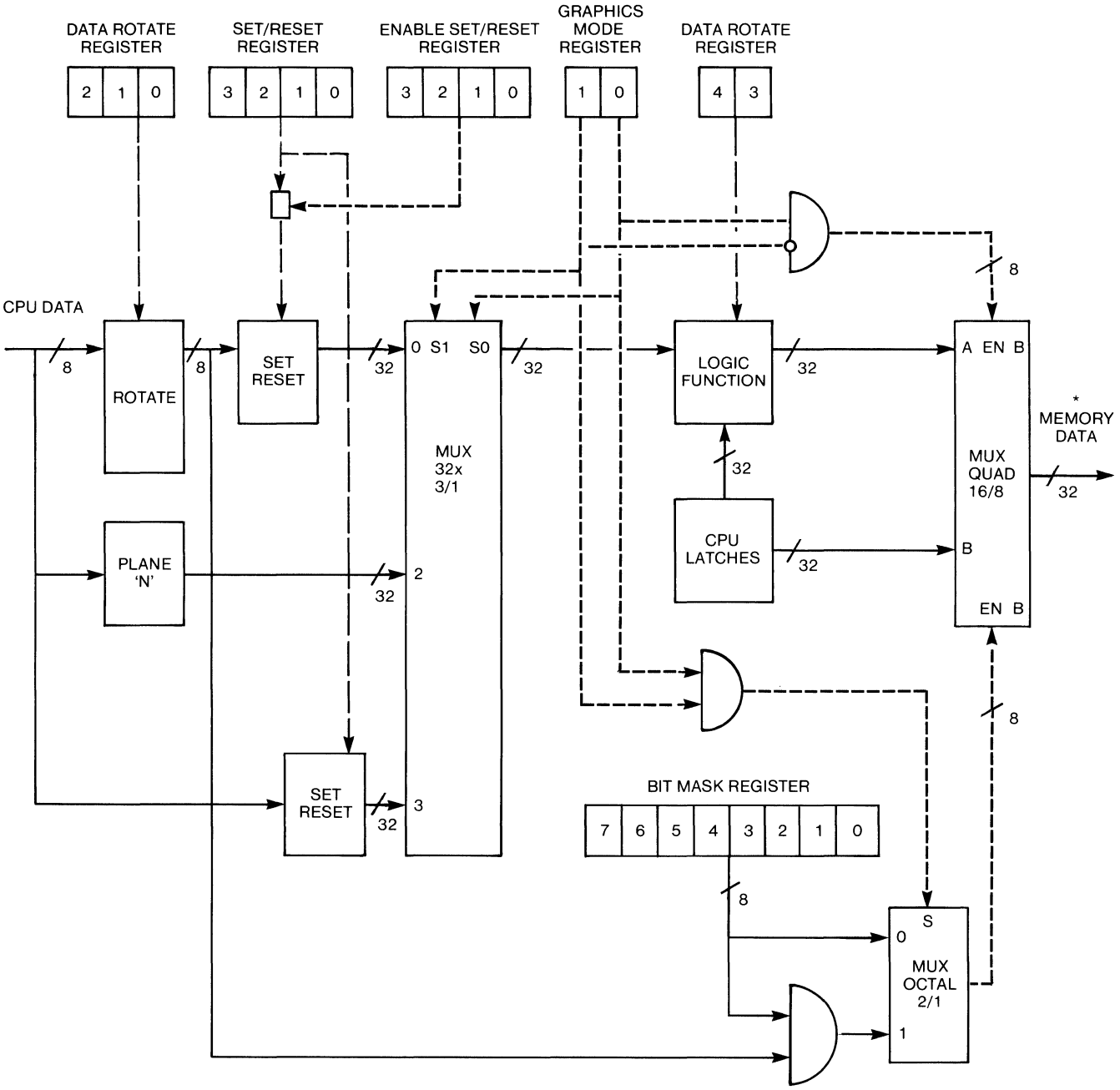
The logic function unit inputs an 8-bit value for each memory map as described above as well as an 8-bit CPU latch value for each map. (The CPU latches contain 32 bits of data from the last read from video memory.) The logic function unit performs logical operations based on the function specified by the function select bits in the Data Rotate Register. The 32-bit output of the logic function unit (8 bits per map) is operated on by the 8-bit-wide Bit Mask Register. If any bit in the Bit Mask Register is a logical one, then the corresponding bit in each map's 8-bit value will take on the value stored in the CPU latches for that bit. (This means that the corresponding bit in each map will be unaffected by the write IF the memory write operation was preceded by a memory read to the same address.) The data resulting from the bit mask operation is written to the four memory maps.

Write Mode 01: In write mode 01_{Binary}, the contents of the CPU latches are written out to the four memory maps. The CPU latches contain 32 bits of data from the last read from video memory. This mode is useful for filling a large number of memory locations with the same value.

Write Mode 10: In write mode 10_{Binary}, the four least-significant bits of the 8 bits of incoming CPU data are each expanded out to 8 bits of data and presented to the logic function unit. CPU data bits 3, 2, 1, and 0 are expanded out to 8 bits of data for maps 3, 2, 1, and 0 respectively. The logic function unit and Bit Mask Register operate as described in the second paragraph under "Write Mode 00" on page 13.

Write Mode 11: In write mode 11_{Binary}, the 8 bits of incoming CPU data are right rotated by the number of bits specified in the Data Rotate Register, with the old least significant bit becoming the new most significant bit. This rotated value is logically "anded" with the contents of the Bit Mask Register to form a mask value whose use is described below.

The bit contained in the Set/Reset Register for each map is expanded to 8 bits and that value is sent to the logic function unit for that map. (Note that the Enable Set/Reset Register has no effect.) The logic function unit inputs an 8-bit value for each memory map as described above as well as an 8-bit CPU latch value for each map. (The CPU latches contain 32 bits of data from the last read from video memory.) The logic function unit performs logical operations based on the function specified by the function select bits in the Data Rotate Register. The 32-bit output of the logic function unit (8 bits per map) is operated on by the 8-bit mask value that is formed as described above. If any bit in the mask value is a logical one, then the corresponding bit in each map's 8-bit value will take on the value stored in the CPU latches for that bit. The data resulting from the bit mask operation is written to the four memory maps.



*Which maps are actually written with data is under the control of the CPU memory address and depending on the mode selected, the Map Mask Register.

Figure 11. Data Flow for VGA Memory Write Operations

Compatibility BIOS

Introduction

The BIOS microcode for the IBM Personal System/2 differs from the IBM Personal Computer family. The BIOS for IBM Personal System/2 Models 50, 60 and 80 supports two interfaces: the existing BIOS interrupt driven interface and the new Advanced BIOS (ABIOS) Call/Return interface.

The BIOS (interrupt-driven interface) has been maintained for the IBM Personal System/2 to ensure a high degree of compatibility. New functions have been added to support the unique features of the IBM Personal System/2 and to assist in masking system hardware differences.

It is strongly recommended that application programmers use only the BIOS and DOS interfaces and not program directly to the hardware. This is essential in order to achieve portability between the IBM Personal Computer family and IBM Personal System/2. Applications that bypass the BIOS interface and directly access routines and/or storage locations may work in one system but not on another. The Advanced BIOS is intended to be accessed by the device driver layer of an operating system. It is not intended to be an application programming interface (API). Programs should interface with the operating system API to achieve compatibility across systems.

Read-Only Memory (ROM)

The system board ROM is made up of four 32K by 8-bit modules. The four modules are arranged in a 128K by 8-bit configuration. The ROM is assigned address hex E0000 through hex FFFFF. The ROM contains Power-On Self Test (POST) code, BIOS, ABIOS, and BASIC.

IBM BIOS Interface Technical Reference Manual

A new manual is available whose contents solely address BIOS information. It includes BIOS and Advanced BIOS interfaces. The BIOS section of the document contains interface definitions and return status for the IBM Personal Computer family and IBM Personal System/2. This manual is intended for use by software developers. Software developers may use this information to develop software for IBM Personal Computers and IBM Personal System/2. The System Technical Reference Manual no longer contains BIOS information.

Interrupt 10H (Video)

The video subsystem for IBM Personal System/2 Models 50, 60 and 80 is a highly compatible extension of the IBM Enhanced Graphics Adapter (EGA) that is integrated on the system board in the Video Graphics Array (VGA) and Video Digital-to-Analog Converter (DAC) chips. The video BIOS is based on the EGA BIOS and provides all EGA functions with extensions and additional functions. Extensions include support for higher resolution text and graphics with extended palette and character generator loading capabilities. The video BIOS provides the interfaces shown in Figure 12.

AH (hex)	Description
00 *	Mode Set
01	Set Cursor Type
02	Set Cursor Position
03	Read Cursor Position
04	Read Light Pen Position
05	Select Active Display Page
06	Scroll Active Page Up
07	Scroll Active Page Down
08	Read Character at Current Character Position
09	Write Character(s) at Current Cursor Position
0A	Write Character(s) only at Current Cursor Position
0B	Set Color Palette
0C	Write Dot
0D	Read Dot
0E	Write Teletype to Active Page
0F	Return Current Video State
10 *	Set Palette Registers
11 *	Character Generator Load
12 *	Alternate Select
13	Write String
14-19	Reserved
1A **	Display Combination Code (DCC)
1B **	Return Functionality/State Information
1C **	Save/Restore Video State

* Extensions
** New Functions

Figure 12. Video Interface Reference

Mode Set

The mode set function provides support for all video modes supported by the IBM Color/Graphics Monitor Adapter, the IBM Monochrome Display and Printer Adapter, and the IBM Enhanced Graphics Adapter. In addition, new mode support includes: 360x400 text in 16 colors, 720x400 text in 16 colors and monochrome, 640x480 graphics in both 2 and 16 colors, and 320x200 graphics in 256 colors. All video modes are available independent of the display that is attached to the system. The mode set function provides support for the modes shown in Figure 13.

MODE #	TYPE	MAX COLORS	ALPHA FORMAT	BUFFER START	BOX SIZE	MAX PAGES	DISPLAY PEL DIMENSIONS
0	A/N	16/256K	40X25	B8000	8x8	8	320X200
0	A/N	16/256K	40X25	B8000	8x14	8	320X350
0*	A/N	16/256K	40X25	B8000	9x16	8	360X400
1	A/N	16/256K	40X25	B8000	8x8	8	320X200
1	A/N	16/256K	40X25	B8000	8x14	8	320X350
1*	A/N	16/256K	40X25	B8000	9x16	8	360X400
2	A/N	16/256K	80X25	B8000	8x8	8	640X200
2	A/N	16/256K	80X25	B8000	8x14	8	640X350
2*	A/N	16/256K	80X25	B8000	9x16	8	720X400
3	A/N	16/256K	80X25	B8000	8x8	8	640X200
3	A/N	16/256K	80X25	B8000	8x14	8	640X350
3*	A/N	16/256K	80X25	B8000	9x16	8	720X400
4	APA	4/256K	40X25	B8000	8x8	1	320X200
5	APA	4/256K	40X25	B8000	8x8	1	320X200
6	APA	2/256K	80X25	B8000	8x8	1	640X200
7	A/N	MONO	80X25	B0000	9x14	8	720X350
7*	A/N	MONO	80X25	B0000	9x16	8	720X400
08-0C	RESERVED						
0D	APA	16/256K	40X25	A0000	8x8	8	320X200
0E	APA	16/256K	80X25	A0000	8x8	4	640X200
0F	APA	MONO	80X25	A0000	8x14	2	640X350
10	APA	16/256K	80X25	A0000	8x14	2	640X350
11*	APA	2/256K	80X30	A0000	8x16	1	640X480
12*	APA	16/256K	80X30	A0000	8x16	1	640x480
13*	APA	256/256K	40X25	A0000	8x8	1	320x200
* New VGA Modes							

Figure 13. Video Mode Table

The Scrolling, Read/Write Character and Read/Write Dot functions support the new video modes.

The IBM Personal System/2 Models 50, 60 and 80 Power-On Self Test code senses the type of display that is attached to the system. Based on this determination, the default video mode with a IBM 8503 Monochrome Display attached is monochrome mode 7 (720x400), and with an IBM 8512 Color Display or IBM 8513 Color Display attached is color mode 3 (720x400).

An IBM 8503 Monochrome Display operating in color modes represents colors as gray shades. There are 16 out of 64 possible gray shades available in 16 color modes and 64 gray shades available in the 256 color mode (mode hex 13).

Selection between text mode scan lines (200, 350 and 400) is accomplished through the Alternate Select function. Scan lines are the number of horizontal pel lines that the hardware displays on the screen. The system defaults to the highest scan line resolution of 400 scan lines. Selection of 200 and 350 scan lines in text modes is provided to emulate the text support that is available on the IBM Color/Graphics Monitor Adapter, the IBM Monochrome Display and Printer Adapter and the IBM Enhanced Graphics Adapter.

All 400-scan-line text modes represent characters in a 9-pel-wide character box to improve text readability. The BIOS mode set function loads the 8x16 character font, and then extends certain characters to be represented in the wider character box. The VGA hardware automatically adds a blank 9th pel for each character for all ASCII character codes except those in the range of hex C0 to hex DF. For ASCII character codes in this range, the 8th pel is duplicated in the 9th pel position. This allows text graphics characters to connect across character boxes.

The mode set function supports dynamic overrides for alphanumeric character fonts and graphics fonts as supported through the IBM Enhanced Graphics Adapter save pointer function. This support has been extended to allow alternate palette definitions, and support for 512 alphanumeric character fonts.

Mode Set Default Palette Loading

The VGA has an internal set of 16 palette registers that duplicate the 16 6-bit palette registers (2 bits for each color) on the EGA. The BIOS mode set function loads these internal palette registers with the same values that are programmed by the EGA BIOS. The 6-bit output of the internal palette is used as an index into the extended video DAC palette.

The video DAC contains 256 18-bit palette registers (6 bits for each color) called color registers. The first 64 color registers are loaded with emulation values such that compatible colors are displayed on the analog monitors when the EGA compatible palette is programmed for the 200 or 350 scan line modes used on the following displays:

- 5151 = IBM Monochrome Display
- 5153 = IBM Color Display
- 5154 = IBM Enhanced Color Display

This allows EGA applications to get compatible results on VGA. The remaining 192 color registers are not loaded or modified during a mode set and can be used through the VGA paging capability discussed on page 19.

For the 200 scan line modes the internal EGA compatible palette is programmed such that 4 bits of each palette register represents the Intensity, Red, Green, and Blue bits used by the IBM Color Display Model 5153.

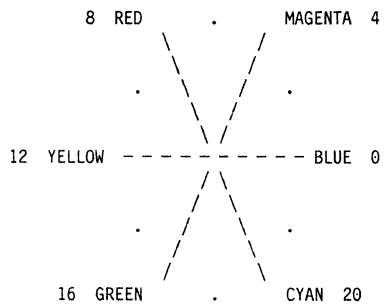
For color modes with 200 scan lines, the video DAC's color registers are loaded with the appropriate values such that compatible colors are displayed on analog color displays.

For monochrome modes, the video DAC's color registers are loaded with values of black, white, and intensified white which correspond to the monochrome attributes of no video, normal video, and intensified video.

For the EGA-compatible 350-line modes, the video DAC's first 64 color registers are loaded with appropriate values so that compatible colors are displayed on the analog displays.

For the new mode 13H a special palette made up of 16 default colors, 16 gray levels and 216 colors selected from a Hue-Lightness-Saturation (HLS) model is programmed into the video DAC's color registers. These colors cover 24 hues in 3 lightness values (bright, medium and dark), and 3 saturation values (pure, impure and dark). This palette defines a wide spectrum of colors to choose from.

The 24 hues are evenly spaced around the color wheel at 15 degree intervals with Blue at 0. These hues are assigned numeric values from 0 to 23.



The three lightness values are bright, medium and dark. The three saturation values are pure, impure and grayish. Saturation or purity is a measure of how pure a color is. A color made up of only 1 or 2 primaries (Red, Green, or Blue) is a pure color and a color made up of 3 primaries is an impure color. As the value of the smallest of the three primary colors is increased the purity of the color decreases. If all three primaries are equal, the color is an achromatic gray level. Figure 14 maps lightness (LGT) and saturation (SAT) terms into numeric values.

Lightness	Saturation
Bright 0	Pure 0
Medium 1	Impure 1
Dark 2	Gray 2

Figure 14. Lightness and Saturation Table

A color in this HLS color space can be converted into a color register index for use in mode 13H. The index is calculated from the HUE, LGT, and SAT numerical values using the following equation:

```
Index = (LGT*72) + (SAT*24) + (HUE+32)
```

For example:

medium-gray-yellow = (1*72) + (2*24) + (12+32) = 164
dark-pure-green = (2*72) + (0*24) + (16+32) = 192
bright-impure-blue = (0*72) + (1*24) + (0+32) = 56

The complete mode 13H 256-color palette is summarized in Figure 15.

0 - 15	16 default colors
16 - 31	16 gray levels
32 - 55	24 bright pure hues
56 - 79	24 bright impure hues
80 - 103	24 bright grayish hues
104 - 127	24 medium pure hues
128 - 151	24 medium impure hues
152 - 175	24 medium grayish hues
176 - 199	24 dark pure hues
200 - 223	24 dark impure hues
224 - 247	24 dark grayish hues
248 - 255	8 user-defined

Figure 15. 256 Color Palette Table

Palette Register Loading

The BIOS palette support on IBM Personal System/2 Models 50, 60 and 80 has been extended. These extensions include new interfaces to read the VGA's 16 internal palette registers and overscan (border) register which are write-only registers on the EGA. Also included are read/write interfaces for the video DAC color registers, and support for the video DAC paging capability.

The video DAC is external to the VGA chip and has 256 color registers. Each color register has red,

green and blue components (6 bits each) that make up an 18-bit value that allows 256K possible colors. The BIOS interfaces support reading and writing of a single color register or a block of color registers.

In order to support "color" modes on the monochrome analog displays, the BIOS has a color register summing function that represents colors as gray shades. The BIOS summing function is automatically enabled when an IBM 8503 Monochrome Display is attached to the system and a color mode is selected. It can also be enabled through the Alternate Select function when an IBM 8512 Color Display or IBM 8513 Color Display is attached to the system.

Color summing is an algorithm that combines 30% of the red component, 59% of the green component and 11% of the blue component to produce an intensity value between 0 and 63 that is loaded into all three components of the color register.

When color summing is enabled, the 16 gray shades corresponding to the default 16 colors are preselected instead of using the color summing algorithm in order to provide maximum contrast. This provides the best possible appearance for existing applications that use the default color settings.

The VGA paging capability allows software to switch between alternate color palettes loaded above the first 64 color registers. This allows up to 256 different colors to be accessed through page switching. There are two different color register paging modes supported by the VGA and BIOS: a 64-register block mode that allows selection of up to 4 pages, and a 16-register block mode that allows selection of up to 16 pages. The mode set function defaults to the 64-register block mode, with the first block of 64 color registers active.

Character Generator Loading

The character generator support on IBM Personal System/2 Models 50, 60 and 80 has been extended to include the ability to load up to 8 character fonts simultaneously into the VGA character generator RAM, and support for a new 8x16 ROM character font. The EGA block specifier BIOS interface has been extended to allow selection of any one (or two for a 512 character set) of the 8 blocks provided in VGA.

Display Combination Code (DCC)

The Display Combination Code (DCC) is a new Video BIOS function that provides the capability to determine which video controller and attached display are in the system. This is accomplished by preserving the results of the video device presence test that occurs during the Power-On Self Test (POST). Through this BIOS function, video device presence can be determined.

The DCC can be read by invoking an interrupt 10H request with (AH) = 1AH and (AL) = 0. BIOS will return a 1AH in (AL) to indicate that the DCC function is supported. Any other value returned in (AL) indicates that other means should be used to determine display type.

The DCC BIOS call returns two numbers which represent the Active Display Device (in the BL register) and the Alternate Display Device (in the BH register). Each number represents a unique device configuration as listed below:

- 00H - No Display
- 01H - IBM Monochrome Display and Printer Adapter
- 02H - IBM Color/Graphics Monitor Adapter
- 03H - Reserved
- 04H - IBM Enhanced Graphics Adapter (color display)
- 05H - IBM Enhanced Graphics Adapter (monochrome display)
- 06H - IBM Professional Graphics Controller
- 07H - Video Graphics Array (VGA) (analog monochrome display)
- 08H - Video Graphics Array (VGA) (analog color display)
- 09H - Reserved
- 0AH - Reserved
- 0BH - IBM Personal System/2 Model 30 (analog monchrome display)
- 0CH - IBM Personal System/2 Model 30 (analog color display)

The following programming example shows how the DCC can be utilized to determine display types with the DCC BIOS call and a simple table lookup.

```
dcc_??  db      'Unknown','$'
dcc_00  db      'No Display','$'
dcc_01  db      'Monochrome Display and Printer Adapter','$'
dcc_02  db      'Color/Graphics Monitor Adapter','$'
dcc_04  db      'Enhanced Graphics Adapter (Color)','$'
dcc_05  db      'Enhanced Graphics Adapter (Monochrome)','$'
dcc_06  db      'Professional Graphics Controller','$'
dcc_07  db      'Video Graphics Array (Monochrome)','$'
dcc_08  db      'Video Graphics Array (Color)','$'
dcc_0B  db      'IBM Personal System/2 Model 30 (Monochrome)','$'
dcc_0C  db      'IBM Personal System/2 Model 30 (Color)','$'
dcc_no  db      'No DCC information available','$'
dcc_act db      ' (active) ','$'
dcc_alt db      ' (alternate) ','$'

max_dcc equ     12

dcc_table label word
        dw      offset dcc_00
        dw      offset dcc_01
        dw      offset dcc_02
        dw      offset dcc_??
        dw      offset dcc_04
        dw      offset dcc_05
        dw      offset dcc_06
        dw      offset dcc_07
        dw      offset dcc_08
        dw      offset dcc_??
        dw      offset dcc_??
        dw      offset dcc_0B
        dw      offset dcc_0C

; --- Check if Display Combination Code is Available

Start_dcc:
        mov     ax,1A00h                ; read DCC call; al=00
        int     10h                     ; call video BIOS
        cmp     al,1Ah                  ; al should be 1A
        je      found_dcc               ; if 1A, bx has display codes
        mov     dx,offset dcc_no        ; not available
        mov     ah,09h                  ; DOS print string$
        int     21h
        jmp     Exit

; --- check for unknown primary display type

found_dcc:
        cmp     bl,max_dcc              ; is this>max dcc or 0FF?
        jbe     ok_dcc                  ; if unsigned below/equal its ok
        mov     dx,offset dcc_??        ; unknown display
        mov     ah,09h                  ; DOS print string$
        int     21h
        jmp     Exit

; --- report active display name

ok_dcc:
        push    bx                      ; save (alternate is in bh)
        xor     bh,bh                   ;
        shl     bx,1                    ; compute word index
        mov     dx,dcc_table[bx]        ; display code message
        mov     ah,09h                  ; DOS print string$
        int     21h
        mov     dx,offset dcc_act       ; this is active display
        mov     ah,09h                  ; DOS print string$
        int     21h

; --- check for unknown alternate display type

        pop     bx                      ; alternate display code
        cmp     bh,max_dcc              ; is this>max dcc or 0FF?
        jbe     ok_alt                  ; if unsigned below/equal its ok
        mov     dx,offset dcc_??        ; unknown display
        mov     ah,09h                  ; DOS print string$
        int     21h
        mov     dx,offset dcc_alt       ; this is alternate display
        mov     ah,09h                  ; DOS print string$
        int     21h
        jmp     Exit

; --- report alternate display name

ok_alt:
        mov     bl,bh                   ; alternate code is in bh
        xor     bh,bh                   ;
        shl     bx,1                    ; compute word index
        mov     dx,dcc_table[bx]        ; display code message
        mov     ah,09h                  ; DOS print string$
        int     21h
        mov     dx,offset dcc_alt       ; this is alternate display
        mov     ah,09h                  ; DOS print string$
        int     21h
Exit:
```

The DCC is defined to be extendable by creating an alternate DCC table through the Save Pointer BIOS interface.

Return Functionality/State Information

The video BIOS Functionality function is defined to return video state and functionality information to application and system level software. One part of the returned information defines the static functionality of the video hardware and BIOS. (i.e. modes supported, ROM fonts available, etc.). The other part is dynamic information that indicates the current state of the video hardware and BIOS (i.e. current mode, current character height, etc.).

When the functionality call is made, the information returned is for the currently active video function. In a multiple display environment, the alternate video functionality information is available when that alternate video is active.

Save/Restore Video State

The save/restore interface is defined to save the entire video state or selective parts of the video state. The video state sections supported on IBM Personal System/2 Models 50, 60 and 80 are the video hardware state, the video BIOS data area state, and the video DAC state.

Cursor Emulation

Cursor Emulation is invoked during a Video BIOS Set Cursor Type call (AH=01H) when the VGA is active and cursor emulation is active. The power-on default is for cursor emulation to be active. If cursor emulation is inactive, BIOS will set the requested cursor type unaltered. The state of cursor emulation can be controlled by software through the Video BIOS Alternate Select function.

The cursor emulation code attempts to determine the 'type' of cursor being set by looking at the start value in (CH), the end value in (CL) and the current character height. The cursor is adjusted as necessary for different character cell sizes. Figure 16 on page 21 shows the recognized cursor types.

Cursor Type	Action
No Cursor (START Bit 5 ON/Bit 6 OFF)	PASS
Split Cursor (END < START)	ADJUST
Overbar Cursor (START < END =< 3)	PASS
Underline Cursor (START+2 >= END)	ADJUST
Full-Block Cursor (START =< 2)	ADJUST
Half-Block Cursor (START > 2)	ADJUST

Figure 16. Recognized Cursor Emulation Types

The algorithm first checks for the no-cursor case, then for a split cursor. No-cursors are passed through. Split Cursors are inverted to a Block Cursor (VGA hardware does not support split cursors). The algorithm then checks to see if the requested cursor type needs any adjustment (i.e. does it match the current character size) with the three tests shown in Figure 17.

Adjustment Test	Action
START OR END >= Character Height	ADJUST
END = Character Height - 1	PASS
START = Character Height - 2	PASS

Figure 17. Three Cursor Adjustment Tests

If the cursor type requires adjustment, the algorithm determines which type of cursor (Underline, Block, etc) it is and adjusts it as appropriate. The adjustment methods are shown in Figure 18.

Cursor Type	Adjustment
Overbar	PASS
Underline	Move START/END to Bottom of Cell (BoC)
Full-Block	Move END to BoC
Half-Block	START = Character Height/2; move END to BoC

Figure 18. Cursor Adjustment Methods

The Underline is moved up 1 line in the character cell for Underline Cursors and tall fonts (16 or greater character height). This algorithm correctly recognizes and adjusts the common cursors that are used by most software, and is general enough to support both bigger and smaller character sizes and non-standard character sizes.

Interrupt 13H (Diskette and Disk)

The Diskette BIOS interface is maintained as defined on the IBM Personal Computer AT. IBM Personal System/2 Models 50, 60 and 80 use 3.5-inch diskette drives. This difference is transparent to most programs that use the documented BIOS and DOS interfaces.

A new interface to set the media type for format has been added to support new diskette drive types and enhance usability. This function will set the correct data rate for the format function. If the change line status is found to be active, Diskette BIOS will attempt to reset the change line status to the inactive state, then set the correct data rate. If the attempt to reset the change line fails, then Diskette BIOS will indicate a 'Time-Out' error code hex 80 in register (AH) and the carry flag is set.

This new function is intended to be called once before issuing the format function and must be recalled upon indication from the format function that the media has been changed. Media is required to be present in the drive in order for this function to complete successfully.

The Fixed Disk BIOS interface is maintained as defined on the IBM Personal Computer AT. Error return codes were added to return values for format failure, control data address mark and bad arbitration level.

A new fixed disk function has been added to park the fixed disk drive heads. This function call will position the heads over the landing zone on the specified drive. This landing zone may differ from the physical park which occurs on some units when they are powered off.

Both Diskette and Fixed Disk BIOS use direct memory access (DMA) for data transfers and therefore support overlapped I/O with other devices.

Interrupt 15H (System Services)

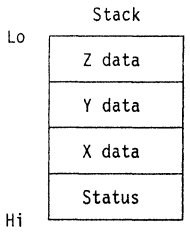
The System Services BIOS interfaces are maintained as defined on the IBM Personal Computer AT. Extensions have been added to support new features available on IBM Personal System/2 Models 50, 60 and 80 as follows:

Pointing Device

The Pointing Device BIOS interface provides support for the IBM Personal System/2 Models 50, 60 and 80 mouse and provides the following functions:

- Enable or Disable the Pointing Device
- Reset the Pointing Device
- Set sample rate
- Set resolution
- Read device type
- Interface initialization
- Read status
- Set scaling
- Device driver installation

After the device driver is installed, BIOS passes data to the device driver on the stack as follows:



- Z data will always be 0
- Y data
  - Bit 0 = Least significant Y bit
  - Bit 7 = Most significant Y bit
  - Bit 8 - Bit 15 = 0
- X data
  - Bit 0 = Least significant X bit
  - Bit 7 = Most significant X bit
  - Bit 8 - Bit 15 = 0
- Status
  - Bit 0 = Left button status
  - Bit 1 = Right button status
  - Bit 2 = 0
  - Bit 3 = 1
  - Bit 4 = X data sign
  - Bit 5 = Y data sign
  - Bit 6 = X data overflow
  - Bit 7 = Y data overflow
  - Bit 8 - Bit 15 = 0

The IBM Personal System/2 Models 50, 60 and 80 pointing device interface defaults to a disabled state at power-on time. The following sequence should be used to setup the IBM Personal System/2 Models 50, 60 and 80 pointing device interface:

- Install the device driver
- Setup the interface
- Enable the pointing device

Program Option Select

The Program Option Select (POS) interface will provide the following services:

- Return base POS address
- Enable an adapter in a particular slot for setup mode
- Disable an adapter in a particular slot from setup mode

System Configuration Parameters

This function returns a double-word pointer to a system configuration parameters table. This table is defined as follows:

```
dw      8           Length of table
db      Model       Model byte
db      SubModel    Sub-model byte
db      Revision    BIOS level
db      ?
        Bit 7 = DMA channel 3 use by BIOS
        Bit 6 = Cascaded interrupt level 2
        Bit 5 = Real time clock available
        Bit 4 = Keyboard scan code hook 1Ah
        Bit 3 = Reserved
        Bit 2 = Extended BIOS data area
        Bit 1 = Micro Channel bus implemented
        Bit 0 = Reserved
db      0           Reserved
db      0           Reserved
db      0           Reserved
db      0           Reserved
```

Figure 19 on page 28 is a System Identification chart that lists the Model and SubModel values for IBM systems.

Extended BIOS Data Area

This interface returns the segment of the extended BIOS data area. This additional BIOS data area has been allocated on new systems. The Extended BIOS data area is allocated by reserving a block of memory (1K) at the top of real mode system memory. This area may be increased as necessary on future systems. The BIOS "Memory Size Determination" function will return the system memory size minus the size of the Extended BIOS data area.

Interrupt 16H (Keyboard)

The Keyboard BIOS interface is enhanced to support new features of the 101/102-Key Keyboard. New function codes are added to Write Keyboard Buffer and Return Extended Keyboard information. The new functions are Extended Keyboard Read, Extended Keystroke Status and Extended Shift Status, which can be viewed as supersets of their existing standard counterparts. The Set Typematic Rate function is added to fully support the programmable typematic rates and delays available on the 101/102-Key Keyboard. A new function, keyboard scan code intercept, is provide through interrupt hex 15, (AH) = 4FH. This facility allows for a keystroke to be changed or absorbed by the system without directly accessing the hardware keyboard port. By intercepting INT hex 15 and monitoring the (AH) register, programs can process the keyboard scan codes as desired. All other keyboard functions are maintained for compatibility with the IBM Personal Computer family.

Programming Considerations

Software Compatibility

The IBM Personal System/2 Models 50, 60 and 80 have a high level of programming compatibility with the existing IBM Personal Computer family and allow most existing software applications to run unmodified. In order to create new programs that will be compatible with IBM Personal Computers and IBM Personal System/2, applications must program to the common interfaces across all models, or develop model-specific programs to take advantage of features supported on certain models.

Normally, programs written for one microprocessor would not run on another microprocessor. But because the IBM Personal System/2 and IBM Personal Computers use a common microprocessor family, most programs need not be modified.

Compatibility among microprocessors alone is not sufficient because applications normally use device services (BIOS) and operating systems (DOS). In order to allow for maximum program compatibility, Personal System/2 maintains the BIOS interfaces and support for IBM DOS Version 3.30. The BIOS microcode implementation on Personal System/2 Models 50, 60 and 80 is different from other IBM Personal Computers. If an application bypasses the BIOS interrupt calls and directly accesses routines and storage locations in one model, the application may not run in another model. Some routines may be similar and some BIOS storage locations may be the same; however, it is strongly recommended that applications use only the BIOS and DOS interfaces.

The following software compatibility guidelines are offered as suggestions for developing programs that will work across all IBM Personal Computers and IBM Personal System/2 models.

- Use high level languages, such as Pascal, C, or FORTRAN, whenever possible because language compilers are much less sensitive to the underlying hardware than assembler language programs.
- Programs should use only the documented interfaces to DOS and BIOS. Programs that write directly to hardware registers or take advantage of undocumented "hooks" are introducing risks that they will not run properly on the next generation of systems.

  Programs should use BIOS calls to derive system configuration information. BIOS provides interfaces that can be used to obtain configuration information such as the number and type of drives installed, the amount of system memory, the type of video subsystem, and so on. Programs that attempt to derive such information from hardware registers run the risk of problems in the future if the hardware unavoidably changes as a result of function enhancements.
- Don't keep interrupts disabled any longer than absolutely required. Communication devices in particular are sensitive to having their interrupts serviced promptly to avoid loss of data.
- Provide a stack that is large enough for your application plus other system functions such as BIOS and DOS calls and hardware interrupts. You should always set up a stack that is at least 256 bytes larger than what your application would require for itself.
- Avoid "time-critical" programming, that is, software such as hard-coded timing loops which depend upon such things as processor type, processor clock speed, the number of memory wait states, and the level of interrupt and DMA activity within the system.
- Whenever possible, programs should not use self-modifying code. The 80286 and 80386 processors have different degrees of instruction pipelining. Several instructions may be prefetched which may prevent self-modifying code from working correctly. A JMP or CALL instruction should be used to clear the instruction prefetch queue before attempting to execute self-modified code.

  Programs that rely on "self-modifying" code may require revision to run in "protected" mode because a code segment is not writeable in protected mode.
- The BIOS routines for the IBM Personal System/2 products have defined an Extension to the BIOS Data Area at the top of user memory just below the 640KB boundary. BIOS protects this memory by decreasing the apparent system memory size. The Memory Size Determination BIOS function will report a memory size which is decreased by the amount of space used for the Extended BIOS Data Area. DOS relies on the BIOS function for memory size and hence will not allocate memory

at addresses which conflict with the Extended BIOS Data Area. However, if an application blindly uses memory it may overwrite the Extended BIOS data which could result in improper system operation.

Using the same assembly language and the BIOS and DOS interfaces help to achieve application compatibility. However, there are still several factors which need to be taken into consideration. This section describes the major differences between the systems in the IBM Personal Computer family and the IBM Personal System/2. Also included are programming considerations that should be considered when designing application software.

Video Presence Testing

The following procedure is recommended to determine which IBM video controllers are present, and should be used as a guideline for writing appropriate presence tests for application software. It should be noted that the Read Display Combination Code is supported on all IBM Personal System/2 products.

1. The first step is to issue an interrupt 10H request with (AH) = 1AH and (AL) = 00H (Read Display Combination Code). On return from this request, if (AL) is not equal to 1AH, the Display Combination Code function is not supported and step 2 should be followed to determine video presence.

   On return, if (AL) is equal to 1AH, then the information returned in (BX) defines the video environment. The active display code is returned in (BL). The alternate display code (if any) is returned in (BH). Refer to interrupt 10H interface description for function (AH) = 1AH for display code definitions.

2. This step should be followed to determine the presence of an IBM Enhanced Graphics Adapter (EGA) when the Display Combination Code function is not supported. Issue an interrupt 10H request with (AH) = 12H and (BL) = 10H (Return EGA Information). On return from this request, if (BL) is equal to 10H, an EGA is not present and step 3 should be followed.

   On return, if (BL) is not equal to 10H, then an EGA is present. Note that an IBM Color/Graphics Monitor Adapter or an IBM Monochrome Display and Printer Adapter could also be present based on the EGA switch settings.

3. This step should be followed only after steps 1 and 2 are completed. The possible video controllers present at this point are the IBM Color/Graphics Monitor Adapter, the IBM Monochrome Display and Printer Adapter, or both. Perform a presence test on video buffer addresses 0B8000H and 0B0000H to determine which of these video adapters is present.

It should be pointed out that the IBM Display Adapter/A is not a CGA-compatible design and hence is not considered here.

Video Mode Switching

The following procedure is recommended when writing applications that switch between monochrome and color video modes. A correct video function presence test, as described above, is required. A system can have one of three possible video environments:

1. A single video controller that supports either color video modes or monochrome video modes. If a color display is present, then only color video modes are available. If a monochrome display is present then only monochrome video modes are available.

2. Two video controllers, one that supports color video modes, and another that supports monochrome video modes. In this case both monochrome and color video modes are available. To switch from monochrome modes to color modes or from color modes to monochrome modes, the application should change the system equipment flag video bits (40:10 bits 4 and 5) to monochrome or color as desired and invoke the appropriate video BIOS mode set request.

3. A single video controller that supports both color video modes and monochrome video modes on the same display. To determine if this is supported an application can issue an interrupt 10H request for function (AH) = 1BH (Return Functionality/State Information).

   On return, if (AL) is not equal to 1BH, then the Return Functionality/State Bios function is not supported. Support for both color and monochrome video modes on a single video display is not available.

   On return, if (AL) is equal to 1BH then use the returned information to determine if the All Modes on All monitors function is active. If active then color and monochrome modes are available and the application should change the system equipment flag video bits to monochrome or color, and invoke a video BIOS mode set request. If inactive, then only color modes or monochrome

modes are available based on the video presence test.

Diskette Compatibility

IBM Personal System/2 Models 50, 60 and 80 use 3.5-inch diskette drives rather than 5.25-inch drives. The differences between these drives are transparent to most programs that use the documented BIOS and DOS interfaces.

Programs should assure that the parameters passed in (DL) (drive number) and (AL) (sector count) are valid. For the diskette format function, the value in the AL Register does not effect the operation and is not required on entry.

If a Reset function (AH = 00H) is called without specifying DL, then the returned status in AH may be invalid. It is recommended in this case to use function Read Status of Last Operation (AH = 01H) to obtain the correct diskette status.

Before formatting a diskette, programs should issue a Set Media Type for Format (AH = 18H) function call. The Read Drive Parameters (AH = 08H) function returns the correct parameters to use for the Set Media Type for Format function.

The diskette change line associated with the 3.5-inch diskette drives will indicate when the media in a drive is changed. This signal reduces the risk of erroneous I/O to a diskette that has just been inserted. This may also improve performance by allowing buffers such as DOS's directory and file allocation table to remain in memory as long as the diskette is not changed.

Programs should use BIOS function Read DASD Type (AH = 15H) to determine if the diskette drive supports the diskette change line. If the drive does not support the change line, then the program never receives an error code of 06H (Diskette Changed) from BIOS functions Read, Write, Verify, or Format. In addition, the Disk Change Line Status function (AH = 16H) will always return Change Line Active (AH = 06H).

The diskette drive controller reads and writes both high and low-density media. When switching from one density to another, the following changes occur:

- The clock rate is 8 MHz for high density media, 4 MHz for low-density media.
- The step rate value is hex 0A for high-density media, hex 0D for low-density media.

Fixed Disk Compatibility

The fixed disk BIOS for the ST-506 resides in the system ROM. A 16-byte parameter table defines the drive type parameters used by BIOS to interface with the drive. These drive type tables are contained in the system ROM and the number of drive types supported has been extended to 32 types. All drive types available for Personal Computer AT are supported.

The ESDI fixed disk controller BIOS resides in a ROM chip on the ESDI adapter. The drive type information used by BIOS is obtained directly from the drive and is not stored in ROM.

Programs should obtain drive type parameter information through the BIOS interface because drive parameters are handled differently between ESDI and ST-506. Also the BIOS interface to obtain fixed disk operation status should be used because this status information is stored in different BIOS Data Area locations for ST-506 and ESDI.

The ESDI hardware interface uses a relative block address interface. The BIOS masks this difference by converting the head, cylinder and sector passed to BIOS to the appropriate relative block address.

The diagnostic format for ESDI is different than the support available for ST-506. The "format-desired-cylinder" is not supported on ESDI because the controller is based on a relative block address structure and handles defects differently than ST-506 drives. A diagnostic format unit function has been provided to format ESDI drives.

Hardware Compatibility

System-oriented programming sometimes requires application software to bypass operating system and BIOS interfaces. When this occurs, the software may be implemented based on hardware details that may be different across existing IBM Personal Computers and IBM Personal System/2. These situations require software to manage the differences or be modified when new models are introduced.

Following are some hardware considerations for implementing system oriented software.

Interrupt Handling for IRQ2 and IRQ9

Hardware interrupt request 9 (IRQ9) is defined as the replacement interrupt level for IRQ2 in systems where there are two interrupt controllers. The hardware IRQ2 becomes the cascade level for attaching the second interrupt controller and the IRQ2 pin on the I/O channel is connected to IRQ9. All devices using IRQ2 should process interrupts the same whether residing in a single or multiple interrupt controller environment. System software redirects all IRQ9 interrupts to the IRQ2 interrupt handler through a software INT 0AH.

Operation of devices on IRQ2 is as follows:

1. A device drives the interrupt request active on the IRQ2 pin on the I/O channel.
2. This IRQ request is mapped in hardware to the IRQ9 input on the second interrupt controller.
3. When the interrupt occurs, the system processor will pass control to the IRQ9 (interrupt 71H) interrupt handler.
4. This system interrupt handler will perform an end of interrupt (EOI) to the second interrupt controller and pass control to the IRQ2 (interrupt 0AH) interrupt handler.
5. The IRQ2 interrupt handler, when processing the interrupt, will cause the device to reset the interrupting condition (for level sensitive systems) and issue an EOI to the first interrupt controller which completes servicing of the IRQ2 request.

Interrupt Handling for IRQ13

The interrupt level 13 handler is used for capturing math coprocessor exceptions and redirecting control to the NMI vector which for compatibility reasons is used by applications to hook in their math coprocessor exception handlers. The interrupt level 13 handler has been carefully designed to support math coprocessor exception handling for the IBM Personal System/2 products.

Level-Sensitive Interrupts

Hardware interrupts are level-sensitive on IBM Personal System/2 Models 50, 60 and 80 whereas they are edge-triggered on IBM Personal Computers. For edge-triggered interrupt systems, the interrupt controller will clear its internal interrupt-in-progress latch when the interrupt routine sends an End Of Interrupt (EOI) command to the controller regardless of whether the incoming interrupt request (to the controller) is still active.

With level-sensitive interrupts, the interrupt condition must be removed by the interrupt handler before the End of Interrupt command is issued to the interrupt controller to prevent the old interrupt from being recognized again.

A level-sensitive interrupt handler must clear the interrupting condition, usually by reading or writing to an I/O port on the device. After clearing the interrupting condition, save the interrupt flag state (PUSHF), disable processor interrupts (CLI) and delay (JMP$+2) prior to sending the EOI to the interrupt controller to ensure that the interrupt request is removed prior to re-enabling of the interrupt controller. Another delay (JMP $+2) should be executed after the EOI is sent and before restoring the state of the interrupt flag (POPF).

Input/Output to Interrupt Controller

Any code that accesses one of the internal registers of the interrupt controller must perform this access with interrupts disabled. The following code fragment is an example of code that enables a particular interrupt level by modifying the interrupt mask register.

```
PUSHF               ; Save current interrupt state.
CLI                 ; Disable interrupts.
IN      AL,21H      ; Input mask from controller 1.
JMP     $+2         ; Small I/O delay.
                    ; "And" with value to unmask
AND     AL,UNMASK_X ; interrupt x.
OUT     21H,AL      ; Output updated mask.
JMP     $+2         ; Small I/O delay.
POPF                ; Restore interrupt state.
```

Accessing Hardware Registers

A program that goes straight to the hardware should disable interrupts around code that reads or writes to registers which require index registers or internal byte pointer registers, or otherwise require a specific sequence of I/O operations to be performed.

For example, to access a VGA Attribute register, the VGA Attribute Index register must first be set to the desired Attribute register and then the Attribute register can be read or written. The Video DAC and Real Time Clock chips contains registers that are accessed in a similar manner.

The DMA controller relies on the state of an internal byte pointer that automatically advances for each byte read or written to a DMA register.

Interrupts must be disabled when performing such accesses to avoid the possibility of an interrupt occurring during the access, which results in control transferring to a program which accesses the same hardware causing the index register to be corrupted.

Some hardware registers have constraints on consecutive I/O operations which require that a minimum time be provided between such operations. To avoid violating such timing requirements it is suggested, where practical, to insert a JMP $+2 between consecutive I/O operations to the same hardware controller. The System Technical Reference contains additional information on hardware register timing requirements.

System Identification

The BIOS Return System Configuration function (Interrupt 15H with register AH = C0H) should be used to determine the system identity. Figure 19 defines the model and sub-model values for IBM Personal Computers and IBM Personal System/2:

System	Model Byte	Sub Model
IBM Personal Computer	0FFh	N/A
IBM Personal Computer XT	0FEh	N/A
IBM Personal Computer XT (256/640 System Board)	0FBh	00
IBM PCjr	0FDh	N/A
IBM Personal Computer AT	0FCh	00, 01
IBM Personal Computer Convertible	0F9h	00
IBM Personal Computer XT Model 286	0FCh	02
IBM Personal System/2 Model 30	0FAh	00
IBM Personal System/2 Model 50	0FCh	04
IBM Personal System/2 Model 60	0FCh	05
IBM Personal System/2 Model 80	0F8h	00, 01

Figure 19. Model Bytes

Copy Protection

The following methods of copy protection may not work on systems using the 3.5-inch 1.44 MB Diskette Drive.

Bypassing Diskette BIOS Routines

- Data Transfer Rate: BIOS selects the proper data transfer rate for the media being used.
- Diskette Parameters Table: Copy protection that creates its own Diskette Parameters Table may not work on these drives.
- Track-to-track access time is set by BIOS for the media being read or written.
- The diskette change signal may not be reset if BIOS is bypassed.
- Rotational Speed: The time between two events on a diskette is a function of the drive. The rotational speed of the 3.5-inch 1.44 Mb Diskette Drive is 300 RPM.

Write Current Control

Copy protection that uses write current control will not work because the controller selects the proper write current for the media being used.

80386 Differences

- Clock Speed

  The 80386 operates at 16 MHz or 20 MHz and hence executes instructions significantly faster than an 80286 processor. Software programming loops (which should be avoided where possible) may need adjustment. Also, to assure the delay required by some I/O devices, jump instructions (JMP $+2) should be placed between consecutive I/O operations. Programmers should avoid writing code that makes use of assumed delays with the system microprocessor operating in parallel with an 80287 Math Coprocessor. Instead the FWAIT instruction should be used to synchronize the system microprocessor with the math coprocessor when necessary. For example, before accessing the result of a coprocessor operation a FWAIT should be executed.

- Overlap of OUT and following instructions.

  Due to the pipelined design of the 80386 it is possible for an I/O bus cycle to not complete until after the 80386 has completed one or more instructions following the I/O instruction. To guarantee the order of execution a JMP SHORT $+2 instruction can be placed between an I/O instruction and the following instruction.

- Order of Accessing Unaligned Operands

  For unaligned word memory operands which require two memory cycles to obtain the entire memory operand, the 80386 transfers the high-order byte first and then the low-order byte. This will have no effect if real memory is being accessed. However, it will cause improper operation of memory-mapped I/O devices which require (or expect) the memory operand to be accessed low-order byte first. Special programming considerations may be required for memory-mapped I/O devices to avoid unaligned memory accesses. One method is to perform only byte accesses to unaligned operands.

- Lock Prefix

  The 80386 restricts the use of LOCK to only those instructions that perform a read followed by a write to a memory operand such as Exchange Memory with Register or Increment Memory. An undefined opcode exception (INT 6) results from using LOCK before any other instruction.

- 16 Megabyte Wraparound

  Any program written for an 80286 system that relies on the wrapping of addresses beyond 16M bytes to the first megabyte address range will not work correctly on the 80386 because such addresses do not wrap but rather fall into the 17M byte address range.

- NT and IOPL Bits

  The 80386 allows the NT and IOPL bits to be modified in real mode unlike the 80286 which forces these bits to be zero.

- New 80386 Descriptor Types

  The 80386 introduces several new system descriptor types to support 80386-type tasks.

- Use of 32-Bit Registers and 32-Bit Addressing Mode

  The 80386 32-bit registers and a new 32-bit addressing mode are accessible in all the 80386 operating modes. However, it is recommended that such features should only be used when running on a 32-bit operating system.

  Figure 20 illustrates a problem that would occur in a multi-tasking environment with a 16-bit Operating System when 32-bit registers were used by more than one task.

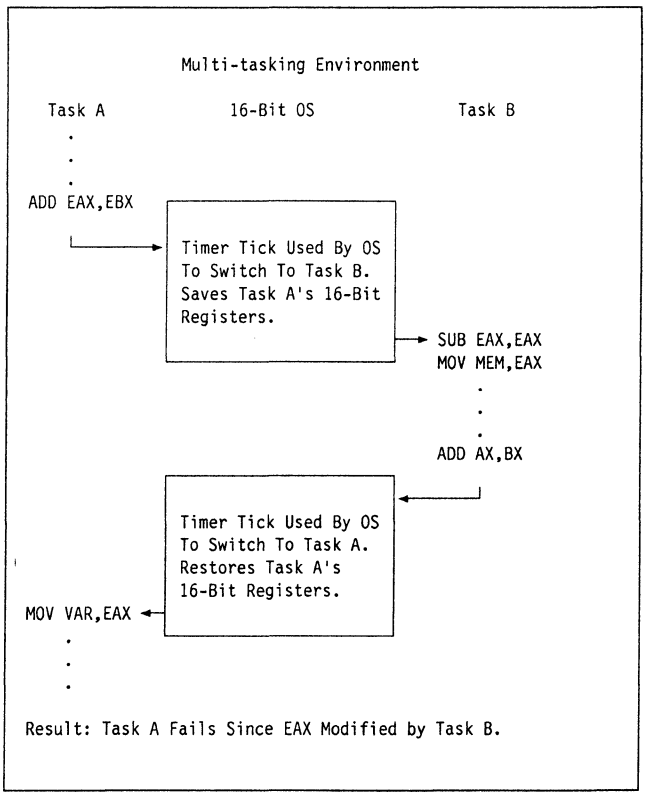


Figure 20. Multi-Tasking Environment

1. First let's say that Task A has been executing for a while and is now executing an ADD instruction, adding to the 32-bit EAX register the contents of the 32-bit EBX register.
2. A timer tick interrupt occurs and the 16-bit operating system switches control to Task B after saving Task A's 16-bit registers.
3. Task B resumes its execution and executes a subtract EAX from EAX instruction which sets EAX equal to 0. It then stores the result to memory location MEM. Task B continues to execute until the next timer tick interrupt occurs.
4. At which time, the operating system switches control back to Task A after restoring Task A's 16-bit registers.
5. Task A resumes execution at its next instruction, which is Move to memory operand VAR, the contents of EAX.

However, since EAX was modified by Task B and not preserved across the task switch, Task A will store the wrong result into VAR and hence will fail to execute correctly.

One possible method to avoid such a problem is to briefly disable interrupts while performing 32-bit operations.

80386 B-1 Stepping Level

The following items describes anomalies or errata that apply to systems which contain the B-1 stepping level of the 80386 microprocessor:

- Wrong CX update with REP INS

  The CX register is not updated correctly in case of REP INS (any INPut string instruction with REPeat prefix), followed by an early-start instruction (push, pop or memory reference instructions). After any REP prefixed instruction execution, CX is supposed to be 0. But in the case of such an INS instruction, CX is not updated correctly and is hex 0FFFF. It should be noted that INS is still executed the correct number of times and DI is updated properly.

- LSL cannot precede PUSH/POP

  If the instruction executed immediately after a Load Segment Limit (LSL) instruction does a stack operation the value of SP after the operation may be incorrect. Stack operations resulting from non-instruction sources like exceptions or interrupts following the LSL do not corrupt SP.

  A work around is to follow a LSL instruction with a NOP instruction.

- LSL/LAR/VERR/VERW malfunction with a NULL selector.

  An LSL, LAR, VERR or VERW executed with a NULL selector (i.e. bits 15 through 2 of the selector set to zero) will operate on the descriptor at entry 0 of the GDT, instead of unconditionally clearing the ZF flag.

  A work around is to fill in the "NULL descriptor" (i.e. the descriptor at entry 0 of the GDT) with all zeroes which is an invalid descriptor type.

- FSAVE/FSTENV Opcode Field Incorrect

  For some math coprocessor instructions the opcode will be stored incorrectly in the FSAVE/FSTENV Format Image. The system ROM IRQ13 code will "fix" the problem after an unmasked exception occurs.

- New 80386 Code

  Programmers writing code that will only run on an 80386 system should refer to the IBM Personal System/2 Model 80 Technical Reference manual for descriptions of the 80386 B-1 step anomalies that apply to operating systems which support Virtual 8086 Mode and Paging.

Math Coprocessor Considerations

The IBM Personal System/2 Model 80 Technical Reference manual contains greater detail than is described here and should be referenced for further information about these items.

Math Coprocessor Presence Testing: The BIOS Equipment Function should be used where possible as the method for detecting the presence of the math coprocessor.

If a programmer finds it impossible to use the BIOS call, the following technique can be used to determine the presence of the math coprocessor.

- First, the coprocessor is initialized by the No Wait Floating Point Initialize instruction.
- Next, the Floating Point Store Control Word instruction is executed to store the contents of the control word register to a memory location previously initialized to zero.
- The stored value, after masking off undefined bits, is compared against the expected control word value.

- If the compare is ok, then the Floating Point Store Status Word instruction is executed to store the contents of the status register to a memory location previously initialized to all 1's.
- The stored value, after masking off undefined bits, is compared against the expected status word value.
- If both the control and status registers contain the expected values, the math coprocessor is present.

Math Coprocessor Instructions With No Coprocessor: For proper system operation, programs should not attempt to execute math coprocessor instructions when no coprocessor is present.

The 80386 requires a response from the 80387 during the execution of almost all math coprocessor instructions. The 80386 will stop processing indefinitely waiting for a signal from the 80387 if a math coprocessor instruction is executed with no 80387 installed.

The 80286 microprocessor uses a different protocol to communicate with its 80287 math coprocessor. It will proceed to execute other instructions after executing a math coprocessor instruction even if no 80287 is present. It signals the 80287 to perform a math instruction but does not monitor to see if the instruction is started or completed.

Note: If the processor is set to math coprocessor emulation mode and a routine is present to emulate math coprocessor instructions, then math coprocessor instructions can be executed.

80387 Math Coprocessor Considerations

- Infinity Model: The 80387 only supports the Affine Infinity Model (signed infinity). It does not support the Projective Infinity Model which is available on 8087/80287 math coprocessors.
- Data Types

  Support for the following special data types were dropped; Pseudo-Zero, Pseudo-NaN, Pseudo-Infinity, and Unnormal.

  The 80387 splits the special data type NaN into two data types, Quiet NaN and Signaling NaN.
- Denormalized Numbers

  The 80387 for some functions will automatically "normalize" denormalized operands.
- Real/Protect Mode Switching

  The 80387 follows the mode of the 80386, i.e. it is in protect mode whenever the 80386 is in protect mode. However, on an 80286 system the 80287 switches to protect mode via a FSETPM instruction and remains in protect mode until reset.
- Loading of Internal Constants

  The 80387 rounds internal constants according to the rounding mode in effect.
- Priority of Denormal Exception

  The 80387 does not treat the denormal exception as the highest priority exception when more than one exception is outstanding.
- FXTRACT instruction

  The 80387 does not produce the same result as an 80287 for FXTRACT of zero. The 80387 generates a zero-divide exception.

Power On Self Test (POST)

POST is the procedure that is automatically executed when the system is powered on. POST is executed out of System Board ROM.

Once invoked, POST will perform diagnostics on all system board devices to verify the system's capability to support user functions.

In addition to performing a basic test of the system, POST initializes devices integrated within the system and also any adapters that have been added to the system.

At its conclusion, POST informs the user of any errors detected by displaying the appropriate error codes to the screen. POST also utilizes the speaker to indicate success or error conditions.

POST provides a way to integrate adapters with an on-board ROM code into the system. The address range from C8000H thru E0000H is scanned in 2K blocks to search for a valid adapter ROM. A valid ROM has the following signature in ROM at the offset indicated:

Byte 0: 55H

Byte 1: AAH

Byte 2: Length of ROM in 512 byte blocks

Byte 3: Entry via CALL FAR

A checksum is also performed to verify the integrity of the ROM. Each byte, for the specified length, is summed (modulo 100H). The resulting sum must be zero for a valid ROM.

The following new features are supported by the IBM Personal System/2 Models 50, 60 and 80 POST:

- Programmable Option Select
- Memory Relocation
- POST Error Processor
- Power-On Password

The IBM Personal System/2 Models 50, 60 and 80 architecture requires that all hardware switches on the system board and adapters be replaced by latches that are accessible by software. This feature is known as Programmable Option Select (POS). Every time power is turned on, POST configures the system by writing data to the POS latches on the system board and adapter cards. For IBM Personal System/2 Model 50, POST obtains this data from CMOS RAM. For IBM Personal System/2 Models 60 and 80, POST obtains this data from Extended 2K CMOS RAM. Both the CMOS RAM and the Extended 2K CMOS RAM are non-volatile.

Before writing data to the POS latches, POST determines that the battery is good and that the checksums for the CMOS RAMs are correct. If not, POST assumes a minimum configuration.

Data for the POS latches are stored in the CMOS RAMs by either Automatic Configuration or the System Configuration Utility found on the Reference Diskette.

When a portion of memory is bad, POST disables and relocates blocks of memory to form a contiguous block of usable memory. For IBM Personal System/2 Models 50 and 60, relocation takes place in 16K blocks and can only be done for memory on the memory expansion cards. For the Model 80, relocation takes place in 1M blocks and is only done for memory on the system board.

Post Error Processor

During POST, non-fatal errors are displayed as numeric error codes. The errors are also logged in the extended BIOS data area through an interrupt 15h function call. A maximum of five error codes can be logged. If the Reference Diskette is in drive A, the diskette is booted without waiting for the user to press the F1 key.

When the Reference Diskette is booted, a pointer to the logged error codes is returned by a different interrupt 15h function call. A descriptive message is displayed for each logged error code. The error message will guide the user in the next step to take. For errors that indicate configuration changes, the user will be asked to press a key to run the Automatic Configuration utility. If the POST error processor program does not have a message for an error code, the program searches the diskette for a message file. The name of the file must be @DEVICE.PEP, where DEVICE is the three-character ASCII representation of the decimal device number. For example, if the network adapter had a POST error of 3015, the Reference Diskette would search for a file by the name of @030.PEP for error information. The POST

error processor files must be in the format defined in the System Technical Reference manual for POST error processor message files.

If the POST error processor program could not find a message file for the error code, a generic message is displayed.

Security

A physical keylock mechanism provides security for the system covers.

Power-on Security

The system maintains power-on security by storing a password in battery-powered CMOS RAM which is locked by POST to restrict access to the POST security routines. The password can be from one to seven characters in length. It may consist of any combination of the ASCII character keys. The password is stored as scan codes, so the keys used to store the password must be the same as those entered when powering on the computer (i.e. the numbers on the top row of the keyboard and those on the numeric keypad are not interchangeable).

The power-on password must be initialized from an option on the Reference Diskette. After the password has been initialized, it can only be removed or changed at power-on time. The user is prompted at system power-on (by a key symbol displayed in the upper left corner of the display) to enter the power-on password. The user has three tries to type in the correct password, after which the system halts. After the password has been checked during power-on, the area of CMOS RAM containing the password is masked off. It cannot be read or written until the computer is powered-off and back on again.

The password may be changed by entering the old password, followed by a slash (/) followed by the new password.

Network Server Mode

The Network server mode allows the system to boot from the fixed disk with the keyboard locked. When this mode is enabled, the user will not be prompted for the power-on password unless a diskette is in drive A.

The power-on password is loaded into the keyboard controller and keyboard security is enabled. Typing the password will unlock the keyboard. Network server mode can be enabled or disabled through the Set Features option on the Reference Diskette. Network server mode can only be enabled when there is a power-on password.

Keyboard Security

Keyboard security is a method of locking out keyboard input until the correct password is entered. The keyboard password is stored in the keyboard controller and does not have to be the same as the power-on password.

The keyboard can be locked by running the KP.COM program, which can be installed on a DOS formatted fixed disk or diskette by the Set Features option on the Reference diskette. The keyboard password can be changed by using the /c option when running the KP.COM program.

Automatic Configuration

In the event of a 161 or 162 POST configuration error, Automatic Configuration will configure the entire system to the first non-conflicting values for each item defined in the Adapter Description File (ADF) for each adapter. If a 164 or 165 error occurs, only the I/O slot(s) where an adapter change occurred will be configured. Adapters are configured in the order of I/O slot number in which they are installed. Automatic Configuration will not backtrack to previously configured adapters to resolve any conflicts that may arise while configuring adapters in numerically higher slot numbers. Conflicts can be resolved by the user toggling the resource options in the Set Configuration option on the Reference Diskette. Any adapter that has a resource conflict will be disabled. Resource conflicts are identified by marking them with an asterisk.

Automatic configuration determines which system resources an adapter is using by matching the POS data stored in CMOS RAM to the POS data in the adapter's ADF file. The system resources include I/O address space, memory address space, arbitration level, and interrupt level.

Adapter Description Files (ADF)

Each different adapter in the system must have an Adapter Description File that defines the system resources and POS data used by the adapter. The ADF file must be named @CARDID.ADF, where CARDID is the four character ASCII representation of the adapter's hexadecimal ID (high byte first). If the configuration cannot find an ADF file for an adapter, the adapter will be disabled.

The ADF files are written in ASCII text and must follow the Adapter Description File Syntax rules (defined later). Information for the system board, ST-506 type fixed disk adapter, and memory adapters are built into the configuration program and therefore do not require ADF files. The ADF files are merged from the diskette shipped with the adapter onto the Reference Diskette through a menu option on the Reference Diskette. A file is merged to the Reference Diskette only if that file is dated later than a file with the same name on the Reference Diskette or if a particular file does not exist on the Reference Diskette.

Merged file names are :

- *.DGS
- *.ADF
- *.PEP
- COMMAND.COM
- SC.EXE
- CMD.COM
- DIAGS.COM

Adapter Description File Syntax

The Adapter Description File Syntax is defined in greater detail in the System Technical Reference.

Primary Keywords

- AdapterID 0XXXXh
- AdapterName "Adapter Name"
- NumBytes number
- NamedItem
- Prompt "Prompt String"
- Choice "Choice Name" POS Setting Resource Setting
- Help "Help String"

Note: Keywords are not case-sensitive and hence upper and lower case letters can be mixed as desired.

The first keyword required is the AdapterID which must be followed by the hexadecimal representation of the adapter ID.

The AdapterName keyword is followed by the user name for the adapter in quotes.

The NumBytes keyword is followed by a number which represents the number of POS register settings to be defined in the adapter description file.

The NamedItem keyword, which is not required, indicates that an adapter option will be specified next. For adapters with resources that are fixed, specify them in a NamedItem so that conflict checking can be reported properly.

The Prompt keyword is used to define the Option name that follows within quotes. An option is something that can be specified for the adapter such as Port Number, Arbitration Level, ROM Memory Location, and so on.

The Choice keyword is used to define the name for a particular choice for the option. After the Choice Name is the POS Setting which programs the adapter appropriately and the Resource Setting which identifies the resources used for this choice. The syntax for the POS Setting and Resource Setting will be described later.

The Help keyword is used to define a block of text of up to 1000 characters within quotes which will be used as a help screen for the option. The help screen will be displayed if the user presses the F1 key in the Set Configuration utility.

POS Setting

- POS[N] = XXXXXXXXb

Resource Setting

- Arbitration Level(s): ARB level
- Interrupt Level(s): INT level
- I/O Addresses: IO XXXXh-XXXXh
- ROM Memory Addresses: MEM 0XXXXXh-0XXXXXh

The POS keyword is used to define the setting for a particular POS register. POS must be followed by a number within brackets followed by an equal sign followed by 8 binary digits which define the setting followed by the letter b. POS[0] through POS[3] are used to define the POS setting for POS registers hex 102 through hex 105. More than one POS register can be specified for a given choice statement. Bit 0 of POS Byte hex 102 should always be defined as a don't care(x) in the ADF File.

The ARB keyword is used to define one or more arbitration levels that are required for a particular

choice. If more than one arbitration level is defined the levels are separated by blanks.

The INT keyword is used to define one or more interrupt levels that are required for a particular choice. If more than one interrupt level is defined, the levels are separated by blanks.

The IO keyword is used to define one or more I/O address ranges used for registers on the adapter. The I/O address range is defined by an I/O address followed by a dash and a higher I/O address. Additional I/O address ranges are separated by blanks.

The MEM keyword is used to define one or more I/O ROM address ranges that identify memory used in the hex C0000 to hex DFFFF memory space. The ROM memory address range is defined by a memory address followed by a dash and a higher memory address. Additional ROM memory address ranges are separated by blanks.

Note: The Adapter Description File Syntax is defined in greater detail in the System Technical Reference.

Adapter Description File Example

Following is an example of an Adapter Description File for the IBM Personal System/2 Multiprotocol Adapter/A. The name of the file for this adapter is @DEFF.adf. An explanation of each numbered item begins on page 36.

```
AdapterId 0DEFFh 1

AdapterName "IBM Multi-Protocol Communications Adapter"

NumBytes 2 3

NamedItem 4

Prompt "Communications Port"

choice "SDLC_1"   pos[0]=XXX1000Xb  io 0380h-038ch  int 3 4
choice "SDLC_2"   pos[0]=XXX1001Xb  io 03a0h-03ach  int 3 4
choice "BISYNC_1" pos[0]=XXX1100Xb  io 0380h-0389h  int 3 4
choice "BISYNC_2" pos[0]=XXX1101Xb  io 03a0h-03a9h  int 3 4
choice "SERIAL_1" pos[0]=XXX0000Xb  io 03f8h-03ffh  int 4
choice "SERIAL_2" pos[0]=XXX0001Xb  io 02f8h-02ffh  int 3
choice "SERIAL_3" pos[0]=XXX0010Xb  io 3220h-3227h  int 3
choice "SERIAL_4" pos[0]=XXX0011Xb  io 3228h-322fh  int 3
choice "SERIAL_5" pos[0]=XXX0100Xb  io 4220h-4227h  int 3
choice "SERIAL_6" pos[0]=XXX0101Xb  io 4228h-422fh  int 3
choice "SERIAL_7" pos[0]=XXX0110Xb  io 5220h-5227h  int 3
choice "SERIAL_8" pos[0]=XXX0111Xb  io 5228h-522fh  int 3

Help

"This port can be assigned as a:  primary (SDLC1)
or secondary (SDLC2) sdlc, primary (BISYNC1) or
secondary (BISYNC2) bisync, or as a serial port (Serial
1 through Serial 8).  Use the F5=Previous and the
F6=Next keys to change this assignment.  Conflicting
assignments are marked with an asterisk and must be
changed."
```

```
NamedItem 5
Prompt "Arbitration Level for SDLC"

choice "Level_1"   pos[1]=XXXX0001b  arb  1
choice "Level_0"   pos[1]=XXXX0000b  arb  0
choice "Level_2"   pos[1]=XXXX0010b  arb  2
choice "Level_3"   pos[1]=XXXX0011b  arb  3
choice "Level_4"   pos[1]=XXXX0100b  arb  4
choice "Level_5"   pos[1]=XXXX0101b  arb  5
choice "Level_6"   pos[1]=XXXX0110b  arb  6
choice "Level_7"   pos[1]=XXXX0111b  arb  7
choice "Level_8"   pos[1]=XXXX1000b  arb  8
choice "Level_9"   pos[1]=XXXX1001b  arb  9
choice "Level_10"  pos[1]=XXXX1010b  arb 10
choice "Level_11"  pos[1]=XXXX1011b  arb 11
choice "Level_12"  pos[1]=XXXX1100b  arb 12
choice "Level_13"  pos[1]=XXXX1101b  arb 13
choice "Level_14"  pos[1]=XXXX1110b  arb 14

Help

"This assignment need only be changed if it is in
conflict with another assignment.  Conflicting
assignments are marked with an asterisk.  Use the
F5=Previous and the F6=Next keys to change arbitration
level assignments.  Using arbitration levels, this
adapter accesses memory directly without burdening the
computer's main microprocessor.  An arbitration level
of 0 has the highest priority, and increasing levels
have corresponding decreased priority"
```

1 The AdapterID for this adapter is hex 0DEFF. This is an ASCII representation of the ID generated by the adapter. The high byte is followed by the low byte. The AdapterID is required for all ADF files.

2 The AdapterName is "IBM Multi-Protocol Communications Adapter". The AdapterName is required for all ADF files.

3 The NumBytes 2 in this file indicates the adapter uses two POS bytes located at hex 0102 and 0103.

4 This is the first NamedItem for the adapter. The title of the field is "Communications Port". The user can toggle between the 12 different NamedChoices. Each NamedChoice has a unique PosSetting assigned to it in bit locations 1 through 4 of POS byte hex 0102 (pos [0]). Also shown is a ResourceSetting that corresponds to the PosSetting of the NamedChoice. The resources allocated in this NamedItem are I/O addresses and interrupt levels. A help string for this NamedItem is provided below the last NamedChoice.

5 This is the second NamedItem for the adapter. The title of the field is "Arbitration Level for SDLC". The user can toggle between the 14 different NamedChoices. Each NamedChoice has a unique PosSetting assigned to it in bit locations 0 through 3 of POS byte hex 0103 (pos [1]). Also shown is a ResourceSetting that corresponds to the PosSetting of the NamedChoice. The resources allocated in this NamedItem are arbitration levels. A help string for this NamedItem is provided below the last NamedChoice.

IBM Cache Program

Cache Implementation

The cache program (IBMCACHE.SYS) is implemented as a DOS device driver that installs itself on interrupt 13H and 15H. It uses the character device model to install itself, but not to have any particular interface.

Basic Cache Operation

The cache program, when installed, "hooks" interrupt 13H, which is the disk/diskette BIOS function entry point. If the operation is not for a hard file, the cache passes control to the original BIOS routine. If the operation is for a hard file, the cache determines if the requested function is one of the following:

- Read Sectors
- Write Sectors
- Write Long
- Format Track
- Format Unit

Read Sectors

When the cache is requested to perform a read, it first checks to see how much data is to be read. If the number of sectors is greater than or equal to two times the page size, the cache will just pass the read request on to BIOS directly. This is done for two reasons. The first is to prevent the cache from being "flushed" by a large read request. The second is that, for large read requests, the cache "gets in the way" because it breaks the request into page-sized requests to the disk, and this slows down large read operations because of the latency time required between requests.

Write Sectors

When the cache is requested to perform a write, it determines which pages in the cache are affected by the write. The affected pages are updated with the new data that is in the user's buffer. Once all of the affected pages have been updated, the cache issues a write request to BIOS for the originally requested operation. In this way, write only incurs a small overhead of updating buffers and the I/O operation proceeds at full speed once the cache has been updated.

Write Long

When a write long is requested, the cache determines which pages are affected and "flushes" them; that is, they are returned to the free list. This approach was taken because write long may be part of DOS's error recovery on the hard file, and the cache's interpretation of the data in the affected sectors may no longer be valid.

Format Track and Format Unit

When any type of format command is issued to the hard file, the entire cache is flushed. The reasoning for this is that all of the data on the hard file will be over-written and the cache will contain no valid data. This will prevent previous data from appearing on the hard disk after a format.

Differences Between Base and Extended Storage

The cache program can use either base (below 640K) or extended (above 1M) memory. The cache organization changes subtly when extended memory is used. Rather than keep the cache directory structure in extended memory with the page buffer, the buffer that is normally associated with the directory is replaced with a GDT (Global Descriptor Table) entry. The GDT entry points to the buffer in extended memory. This structure allows the cache directory to be manipulated with the same code no matter where the data buffers are. This keeps the differences between base and extended memory accesses just to how the data is moved between the user's buffer and the cache's buffer(s).

A side effect of the cache having buffers either in base or extended memory is that the cache has an intermediate buffer. This intermediate buffer is to allow data being read from the device to be placed in a known area in base storage before it is moved to the cache buffer and user buffer. This intermediate buffer is exactly one page in size. The size of the page is determined when the cache is initialized.

The cache uses three techniques to prevent it from interfering with other programs that use extended memory. Those techniques are:

1. The cache allocates its buffers from the top of extended memory to the bottom. That is, the cache buffers start at higher addresses and go to

lower addresses. Most programs start at lower addresses and go to higher ones.

2. The cache "hooks" interrupt 15H sub-function 88H which returns the amount of extended memory in the PC. The cache returns the amount of extended memory that it is not using.
3. The cache conforms to the VDISK standard in that if interrupt 19H points to a VDISK device driver, the cache will determine how much extended memory the VDISK(s) are using and determine if there is sufficient storage remaining for the cache to be installed.

INT 13H Extensions

The cache provides two extensions to the interrupt 13H interface. These are provided for cache testability and improved functionality when used with removable media. They are as follows:

Get Cache Statistics

This function allows a program to get a pointer to a data area that is maintained by the cache program. This area has values of interest that relate to cache performance. It contains other values such as cache version number, size of a cache page, and other miscellany. The layout of the statistics area is to follow.

The INT 13H calling sequence for this function is:

```
Input:
  AH    = 1DH
  AL    = 01H

Output:
  CY    = error
  NCY   = no error
  AH    = error code (0 if CY not set)
  ES:BX= pointer to statistics area (only if CY not set)
```

Flush Cache

This function allows a program to flush the cache. That is, the cache can be completely reset as to data and errors that it has encountered. This is useful when a drive is to be formatted or if the drive happens to be a removable drive and the drive is to be changed.

The INT 13H calling sequence for this function is:

```
Input:
  AH    = 1DH
  AL    = 02H

Output:
  CY    = error
  NCY   = no error
  AH    = error code (0 if CY not set)
```

Statistics Area

Warning: All values in this area are to be considered read-only unless specifically noted otherwise. Modification of values in this area by other programs will cause the likely loss of user data on the physical device due to incorrect operation of the cache program.

The following data areas contain statistics about how the cache is performing and what kind of requests were made. These fields can be safely modified or reset without affecting the operation of the cache. They are maintained by the cache strictly for information to external programs; the cache makes no use of these fields for correct cache operation.

The following is the number of read requests that have been made since the cache was started or was reset. It is NOT the number of times INT 13 AH=2 has been issued.

```
                public          num_read_req
num_read_req    dd              0
```

The following is the number of read requests that were satisfied by the cache. Hit ratio can be determined by the following: HR = NUM_HITS / NUM_READ_REQ

```
                public          num_hits
num_hits        dd              0
```

The following is the number of times INT 13 AH=2 has been issued.

```
                public          num_reads
num_reads       dd              0
```

The following is the number of sectors (total) that have been requested during INT 13 AH=2. The average size of a read request can be obtained by the following: AR = NUM_SEC_REQ / NUM_READS

```
                public          num_sec_req
num_sec_req     dd              0
```

The following are reserved bytes. They should not be expected to contain any valid data. Likewise, they should not be modified, because incorrect cache

operation with possible loss of data on the physical device is likely.

```
                public          reserved_area
reserved_area   dw              ?
                dw              ?
                dw              ?
```

The following are pointers to the error list. They can be used by the statistics program to dump the error list. The list is maintained in contiguous storage delineated by ERROR_BUFF at the start and ERROR_END pointing past the last structure. There is a higher level structure imposed on the error records, but for reporting purposes, the size of the structure and only part of the structure will be defined. See the end of the statistics area for a definition of the error structure.

```
                public          error_buff
error_buff      label           dword
                dw              ?
                dw              ?
                public          error_end
error_end       label           dword
                dw              ?
                dw              ?
```

The following is the top of extended memory as maintained by the cache. This value should not be used directly, but rather the INT 15H interface that is part of BIOS should be used to determine the top of extended memory.

```
                public          em_top
em_top          dw              0
```

The following is a flag that indicates if the cache is using extended memory. Incorrect cache operation with loss of data on the physical device is likely if this flag is changed once the cache is in operation (0 = base memory contains the buffers, 1 = extended memory contains the buffers).

```
                public          mem_flg
mem_flg         db              0
```

The following is the model byte of the machine that is currently running the cache program.

```
                public          model_byte
model_byte      db              0
```

The following contains the version number of the cache program. This can be used to print out the version number in a status program and/or it can be used to verify the version of the cache program. The version number will have the format of D.DA where D is a digit from 0 to 9 and A is an alphanumeric from A-Z and 0-9 or blank. The initial release of the cache program has this field set to '1.0'.

```
                public          version
version         db              '  '
```

The following is the amount of storage in KB that is allocated to the cache for its buffers. This value should not be changed once the cache is in operation.

```
                public          cache_ram
cache_ram       dw              ?
```

The following is the number of sectors in a page. This field should never be changed! It is for informational use only. Changing this field once the cache is in operation will cause incorrect operation of the cache and possible loss of data on the physical device.

```
                public          num_pages
num_pages       dw              ?
```

The following defines the error structure that is needed by reporting programs. Any fields that are specific to the layout of the cache error recovery mechanism are marked reserved and are not to be used or modified. The entire error recovery structure is considered READ-ONLY and must NOT be modified.

```
error_struct    struc
error_rba       dd      ?       ; starting RBA of page with error(s)
error_drive     db      ?       ; drive containing RBA
error_bits      db      ?       ; bits indicating which sector(s) have
                                ; error(s).  0 indicates a sector with
                                ; an error.  LSB is first sector in
                                ; page.  If a page is less than 8
                                ; sectors, unused bits are set to 0.
error_reserved  dw      ?       ; reserved word.
error_struct    ends
```

Display Statistics Sample Program

The following is a sample C program that reads and displays the cache statistics information.

```
#include <stdio.h>

struct  errors
            {
            long        page_rba;
            char        drive;
            char        err_flags;
            short       next_err;
            };

struct  statbuf
            {
            long        numread;
            long        numhit;
            long        numreq;
            long        numsecs;
            long        *trace_buff;
            short       trace_head;
            struct      errors  *err_list;
            short       *err_end;
            short       em_top;
            char        mem_flg;
            char        model_byte;
            char        version[4];
            short       cache_ram;
            short       num_pages;
            };


main(argc,argv)
int     argc;
char    *argv[];
{

short   trace_index;
short   trace_buf_head;
short   error_end;
short   i;
short   reset;
```

```
int     rc;
int     hitratio;
double  nrd;
double  nhit;
double  hit_ratio;
double  reqs;
double  secs;
double  avg_read;
struct  statbuf *stat_ptr;

reset = 0;
rc = getstat(&stat_ptr);
switch (rc)
     {
     case 0:
          break;
     case -2:
          printf("%s: Disk Cache program is not installed\n",argv[0]);
          exit(-2);
     case -1:
          printf("%s: Error getting the Cache Statistics\n",argv[0]);
          exit(-1);
     default:
          printf("%s: Error %d returned from getting the Cache Statistics\n",argv[0],rc);
          exit(rc);
     }

if (argc > 1)
     {
     if (strcmp(argv[1],"/e") == 0)
          {
          printf("Error List Address = %08lX\n",(long)stat_ptr->err_list);
          error_end = *(stat_ptr->err_end);
          error_end -= (((long)stat_ptr->err_list) & 0x0000ffff);
          error_end >>= 3;
          printf("Error List End Index = %d\n",error_end);
          if (error_end == 0)
               printf("No Errors Encountered by Cache\n");
          else {
               for (i = 0; i < error_end; i++)
                    {
                    printf("%08lX  ",stat_ptr->err_list[i].page_rba);
                    printf("%02X  ",stat_ptr->err_list[i].drive & 0x00ff);
                    printf("%02X  ",stat_ptr->err_list[i].err_flags);
                    printf("%04X  \n",stat_ptr->err_list[i].next_err);
                    }
               }
          }
     if (strcmp(argv[1],"/r") == 0)
          {
          reset = 1;
          }
     }

printf("Cache Program Version ");
for
     (
     i = 0;
     i < 4;
     i++
     )
     printf("%c",stat_ptr->version[i]);
printf("\n");

printf("Cache Size               = %dKB\n",stat_ptr->cache_ram);
printf("Cache Page Size          = %d Sectors\n",stat_ptr->num_pages);

nrd = stat_ptr->numread;
nhit = stat_ptr->numhit;
hit_ratio = nhit / nrd;
hitratio = 100. * hit_ratio;

printf("Cache Hit Ratio          = %d%%\n",hitratio);

reqs = stat_ptr->numreq;
secs = stat_ptr->numsecs;
avg_read = secs / reqs;

printf("Average Sectors per Read = %G\n",avg_read);

if (reset)
     {
     stat_ptr->numread = 0;
     stat_ptr->numhit = 0;
     stat_ptr->numreq = 0;
     stat_ptr->numsecs = 0;
     }
exit(0);
}
```

This code sample is used by the C program above to get the cache statistics from the interrupt 13H function call.

```
                page            60,132
                title           Disk Cache - Get Statistics
                page
;
;
;                       GETSTAT
;
;       This routine will return the statistics that the cache has been
; gathering since it started.
;
;       Calling Sequence
;
;       char    **stat_ptr;
;       int     rc;
;
;       rc = getstat(stat_ptr);
;       if (rc != 0)
;            ... cache not installed ...
;
_TEXT           segment         word public 'CODE'
                assume          cs:_TEXT


                public          _getstat
_getstat        proc            far
                push            bp
                mov             bp,sp
                push            di
                push            si
                push            ds
;
; Get a pointer to the start of the statistics
;
                mov             ah,01dh         ; get statistics pointer
                mov             al,01h
                mov             dl,80h
                int             13h
                jc              getstat_error   ; error?
                mov             ax,es           ; cache installed?
                or              ax,bx
                jz              no_cache_error  ; no - error
;
; Cache program is installed.  Return a pointer to the statistics
;
have_cache      label           near
                lds             di,[bp+6]       ; point to where pointer goes
                mov             [di],bx
                mov             [di+2],es
;
; Return to the caller
;
                sub             ax,ax
getstat_exit    label           near
                pop             ds
                pop             si
                pop             di
                mov             sp,bp
                pop             bp
                ret
;
; Return with an error getting the pointer
;
getstat_error   label           near
                mov             ax,-2
                jmp             getstat_exit
;
; Return indicating that the cache is not installed
;
no_cache_error  label           near
                mov             ax,-1
                jmp             getstat_exit
_getstat        endp
_TEXT           ends
                end
```

Advanced BIOS

Introduction

Advanced BIOS (ABIOS) is a software layer that isolates an operating system from the low-level system hardware interface. The operating system makes functional requests of ABIOS (read, write) rather than directly manipulating the I/O ports and control words of the system hardware. This allows the details of the hardware attachments and the timings of the hardware interfaces to be altered without disturbing the operating system components above the ABIOS interface.

The ROM BIOS on the IBM Personal Computer Family (BIOS) operates as a single-tasking component whose addressing capabilities are limited to less than 1 megabyte of memory and only in the real mode of the Intel microprocessor. ABIOS supports addressing above one megabyte using the protected mode of the Intel microprocessor. ABIOS is contained in ROM but does not preclude a RAM implementation. ABIOS can be operated in the real address mode (real mode), the protected virtual address mode (protected mode), or in a bimodal environment using both the real mode and the protected mode. ABIOS provides a data structure for implementing a protected mode operating system or bimodal operation (real and protected modes).

The requests to ABIOS made by an operating system fall into three categories: single-staged, discrete multistaged, and continuous multistaged. Single-staged requests perform the requested function before returning to the caller. Discrete multistaged requests start an action or operation that involves a delay before the operation is completed. Continuous multistaged requests start an action or operation that also involves a delay but never ends. For multistaged operations, control is returned to the caller during these delays so the processing time may be used. An interrupt from the I/O device usually indicates completion of a stage of the operation.

The following figure shows the three categories of ABIOS requests:

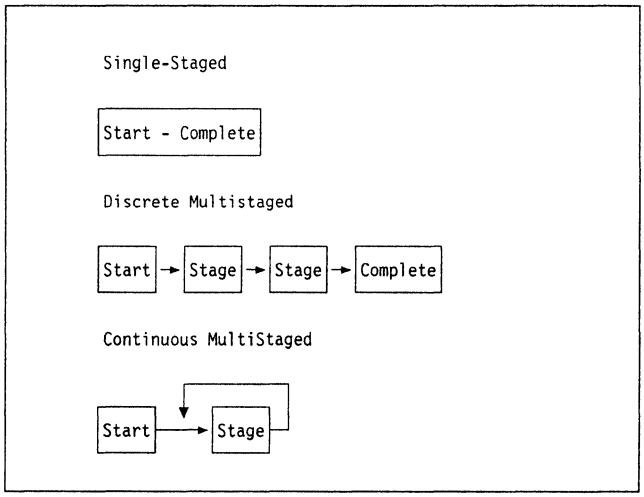


Figure 21. Types of Requests

Data Structures

Requests to ABIOS made by an operating system are made through transfer conventions provided by the ABIOS structure. These conventions require data structures that link the Operating System to the device function routines of each supported device. These data structures include the common data area, function transfer tables, and device blocks. They reside in system memory and are initialized during ABIOS initialization.

The transfer conventions provided by ABIOS are defined to allow operations that use the real mode and/or the protected mode of an Intel microprocessor. To provide flexibility in implementing a protected mode or bimodal operating system, the common data area links all ABIOS pointers into a single structure. This structure contains the Function Transfer Table pointers, the Device Block pointers, and the ABIOS data pointers. The common data area links all ABIOS pointers in a single structure to allow an operating system to manage ABIOS requests in both operating environments of the Intel microprocessors.

Intel is a trademark of Intel Corporation.

ABIOS entry points are stored in a vector table called the function transfer table. Each supported ABIOS device has an associated function transfer table. The first three entries of the function transfer table are structured entry points called the Start Routine, the Interrupt Routine, and the Time Out Routine.

ABIOS routines require a permanent work area for each device called the ABIOS device block. Hardware port addresses, interrupt levels, and device state information are the types of information stored in the device block.

Initialization

Initialization is a defined protocol between ABIOS and an operating system. The operating system plays a major role in the initialization process, including starting the process. Until the operating system starts the initialization process, ABIOS cannot be used. This initialization process must occur in the real mode of the microprocessor, and consists of three steps:

1. The operating system calls BIOS to build the system parameters table. This table describes the number of devices available in the system, the ABIOS common entry points, and the system stack requirements.
2. The operating system calls BIOS to build the initialization table. This table defines the initialization information for each device that the system supports. This information is used to initialize device blocks and function transfer tables.
3. The operating system allocates memory for the common data area using the initialization information returned in step 2. The memory for device blocks and function transfer tables is allocated and the device block and function transfer table pointers are initialized in the common data area. The operating system then calls ABIOS to build the device blocks and function transfer tables for each device.

The flow of the initialization process is shown in Figure 22.

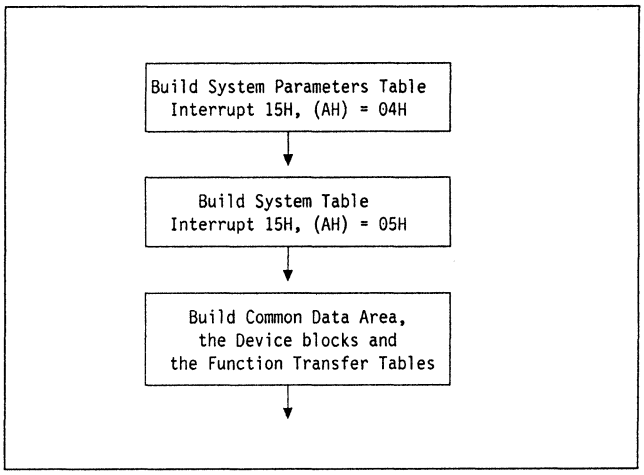


Figure 22. Flow of the Initialization Process

Transfer Conventions

After ABIOS is initialized, requests are presented through a parameter block called the Request Block. The Request Block has fields that identify the target device, requested operation, details of the request, memory locations involved in a data transfer, and the status of the staged/completed request.

ABIOS is implemented as a call-return programming model using one of two methods of calling, the ABIOS Transfer Convention, or the Operating System Transfer Convention. These two calling conventions allow an operating system flexibility in calling ABIOS. Both calling conventions use the stack to pass request information to the target ABIOS device routine.

The ABIOS Transfer Convention is the simplest calling sequence for the operating system. The operating system passes the common data area pointer and the Request Block pointer to one of three common entry points: the Common Start Routine, the Common Interrupt Routine, or the Common Time Out Routine. These common entry points are returned to the operating system during initialization. The common entry routines use the Request Block information and the common data area pointer to get the device block pointer and the function transfer table pointer from the common data area. The common entry routine then gets the requested ABIOS routine starting address from the function transfer table, and calls the ABIOS device routine.

The flow of the ABIOS Transfer Convention is shown in Figure 23.

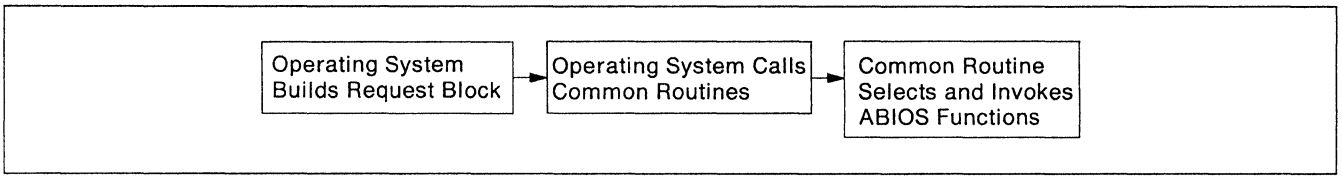


Figure 23. Flow of ABIOS Transfer Convention

The Operating System Transfer Convention requires the operating system to determine the starting address for the requested ABIOS device function. This allows the operating system flexibility in maintaining ABIOS device function addresses that are called frequently. This method is useful for handling interrupts from character and programmed I/O devices that call a single routine repeatedly. The common data area, Request Block, Function Transfer Table, and Device Block pointers are required on entry to the ABIOS device function.

The flow of the Operating System Transfer Convention is shown in Figure 24.

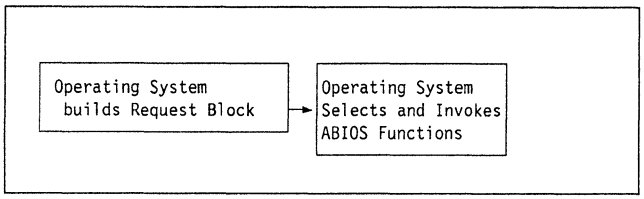


Figure 24. Flow of Operating System Transfer Convention

Interrupt Processing

For multistaged requests, interrupts from hardware devices cause the microprocessor to branch to predefined addresses in the interrupt vector table. ABIOS expects the operating system to receive control at interrupt time. ABIOS provides interrupt routines for the processing of ABIOS interrupts.

Common Data Area

The Common Data Area structure contains function transfer table pointers, device block pointers, and the ABIOS data pointers. These data pointers are established during initialization and contain information for each device the system supports. The common data area links all ABIOS pointers in a single structure to allow an operating system to manage ABIOS requests in both operating environments of the Intel microprocessors. The common data area is required in all operating modes. These are the protected mode only, real mode only, and bimodal implementations.

On each request to ABIOS, a segment or selector with an assumed offset of 0 is passed to ABIOS, which points to the common data area. This pointer is referred to as the Common Data Area Anchor pointer.

Figure 25 shows the common data area and its relationship with the other ABIOS data structures:

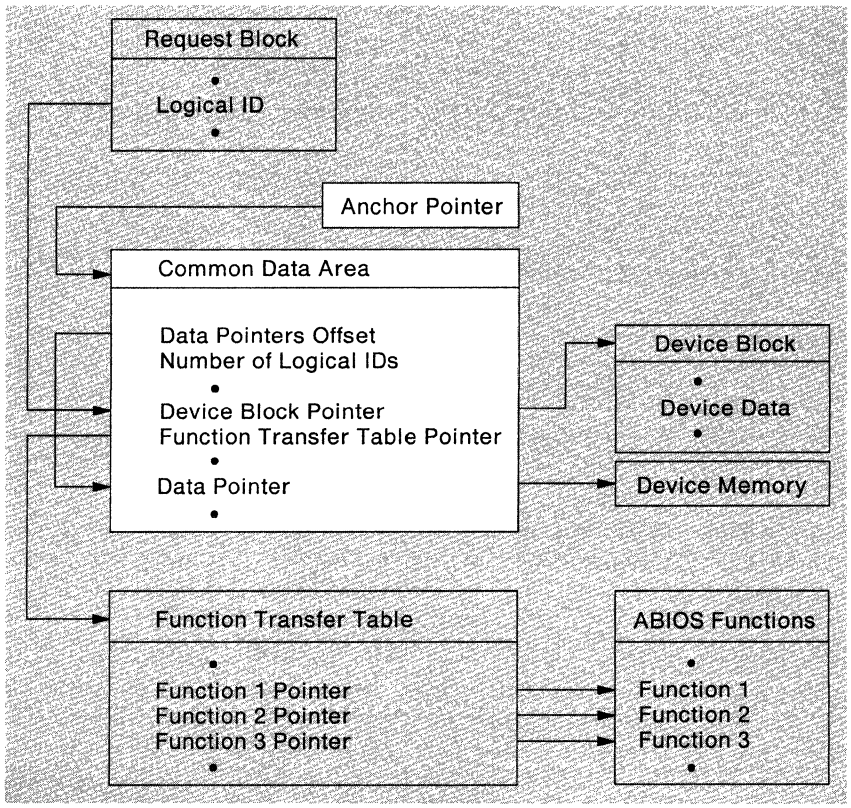


Figure 25. Flow of Common Data Area

Figure 26 shows a detailed representation of the common data area:

Field	Offset	Length
Offset to Data Pointer 0	+00H	2
Count of Logical IDs	+02H	2
Reserved	+04H	4
Device Block Pointer Logical ID **1**	+08H	4
Function Transfer Table Pointer Logical ID **1**	+0CH	4
Device Block Pointer Logical ID **2**	+10H	4
Function Transfer Table Pointer Logical ID **2**	+14H	4
.	.	.
.	.	.
Device Block Pointer Logical ID **n**	+(08H*n)	4
Function Transfer Table Pointer Logical ID **n**	+(08H*n)+04H	4
Data Pointer Length **p**	+(08H*n)+08H	2
Data Pointer Offset **p**	+(08H*n)+0AH	2
Data Pointer Segment **p**	+(08H*n)+0CH	2
Data Pointer Length **p - 1**	+(08H*n)+0EH	2
Data Pointer Offset **p - 1**	+(08H*n)+10H	2
Data Pointer Segment **p - 1**	+(08H*n)+12H	2
.	.	.
.	.	.
Data Pointer Length 0	+(08H*n)+(06H*p)+08H	2
Data Pointer Offset 0	+(08H*n)+(06H*p)+0AH	2
Data Pointer Segment 0	+(08H*n)+(06H*p)+0CH	2
Data Pointer Count	+(08H*n)+(06H*p)+0EH	2

n - is the number of Logical IDs
p - is the number of Data pointers allocated minus 1

Figure 26. Common Data Area

The common data area entries are described in detail below:

Offset to Data Pointer 0: This field combined with the Anchor pointer produces a pointer to the Data Pointer Length 0 field.

Count of Logical IDs: This field contains the number of device block and Function Transfer Table pointer pairs.

Device Block Pointers: These fields contain the pointers to the device blocks for the given Logical IDs.

Function Transfer Table Pointers: These fields contain the pointers to the function transfer tables for the given Logical IDs.

Data Pointer Lengths: These fields contain the lengths of the data areas that are pointed to by the associated data pointer.

Data Pointer Offsets: These fields contain the offsets of the data areas. Each segment is combined with its associated Data Pointer Segment to produce a pointer to the data area.

Data Pointer Segments: These fields contain the segments of the data areas. Each is combined with its associated Data Pointer Offset to produce a pointer to the data area.

Data Pointer Count: This field contains the number of data pointers.

Function Transfer Table

ABIOS entry points are stored in a vector table called the function transfer table. This table contains the doubleword address pointers for each ABIOS function. Reserved function pointers are initialized to 0:0. Each Logical ID (entry in the common data area) has a function transfer table pointer. Multiple Logical IDs can have function transfer table pointers that point to the same function transfer table.

Figure 27 shows a function transfer table and its relationship with the common data area.

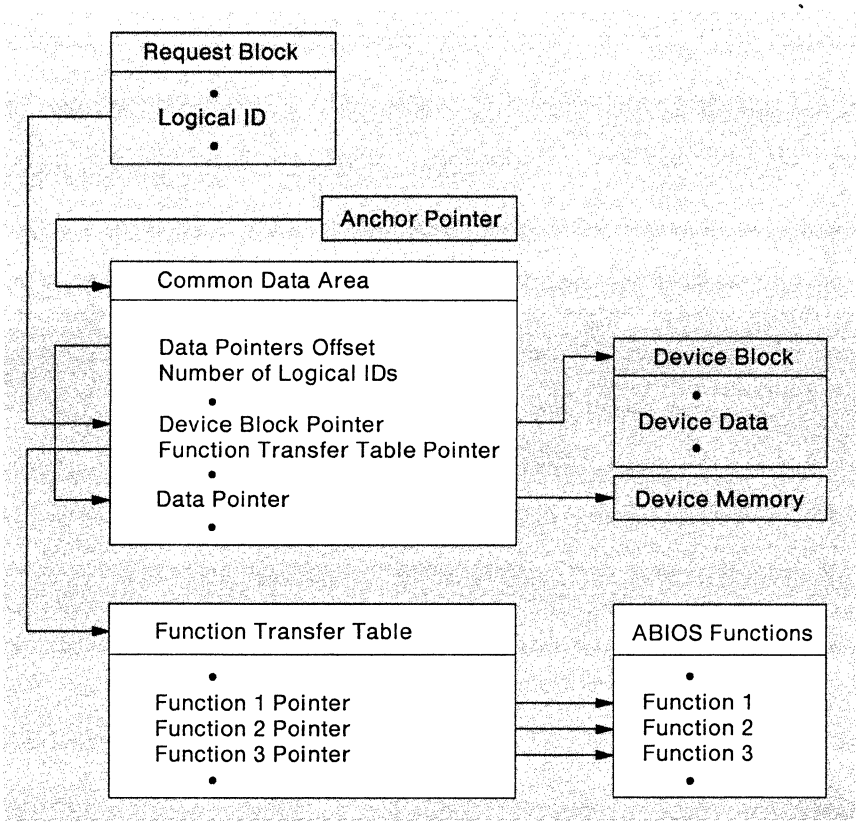


Figure 27. Flow of Function Transfer Table

The operating system allocates and fills in a Request Block, including the Logical ID (defines which device) and Function (defines which function). Based on the information contained in the Request Block, the function transfer table pointer and device block pointer can be located in the common data area for the requested device. The operating system uses the function transfer table pointer to start requests, process interrupts, and handle any timeouts that occur. Each pointer in the function transfer table is a doubleword pointer to a function routine. Figure 28 shows the function transfer table.

Function	Offset	Length
Start Routine Pointer	+00	4
Interrupt Routine Pointer	+04	4
Time Out Routine Pointer	+08	4
Function Count	+0C	2
Reserved	+0E	2
Function 1 Pointer	+10	4
Function 2 Pointer	+14	4
.	.	
.	.	
Function n Pointer	+0C+(4*N)	4

Figure 28. Function Transfer Table

Descriptions of the function transfer table entries follow:

Start Routine: The Start Routine pointer is a double-word pointer, and is called (using call far indirect) to start a request. This routine validates the Function field, the Request Block Length field, and the Unit field. All registers are saved and restored across a call to this routine.

Interrupt Routine: The Interrupt Routine pointer is a double-word pointer, and is called (using call far indirect) to resume a multistaged request upon indication from the hardware. All multistaged requests are resumed through this routine if the operation is not complete. All registers are saved and restored across a call to this routine. If this Function Transfer Table corresponds to a device that does not interrupt, the Interrupt Routine Pointer field is initialized to 0:0.

Time Out Routine: The Time Out Routine pointer is a double-word pointer, and is called (using call far indirect) to terminate a request that fails to receive a hardware interrupt in a specified time. This routine aborts the request and leaves the hardware controller in a known, initial state. All registers are saved and restored across a call to this routine. If this function transfer table corresponds to a device that does not interrupt, or a device that interrupts but never times out, the Time Out Routine Pointer field is initialized to 0:0.

Function Count: This is the word count of functions.

Reserved: This is a reserved word (allocated even if count of functions equal 0).

Function 1: This is a doubleword pointer to the Function 1.

Function 2: This is a doubleword pointer to the Function 2.

Function n: This is a doubleword pointer to the Function **n** routine.

Device Block

ABIOS routines require a permanent work area per device called the ABIOS device block. Hardware port addresses, interrupt levels, and device status information are the types of information stored in the device block.

The Device Block contains both public and private data. The public data in the device block is a readable area whose format is common across all device blocks. This area should not be altered by the operating system. Private data in the device block is used internally by ABIOS and its format and content may not be identical in all implementations of ABIOS. The operating system should neither examine nor alter private data, and IBM reserves the right to alter the contents of the private portion of the device block.

Figure 29 shows a device block and its relationship with the common data area:

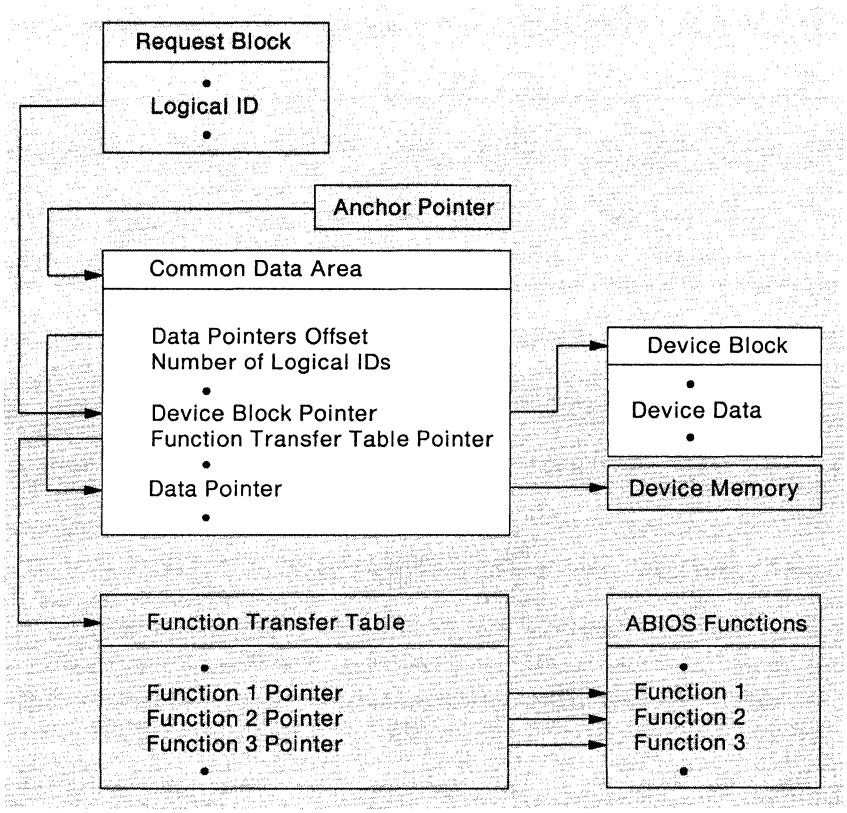


Figure 29. Flow of Device Block

Every ABIOS device has an associated device block. The device block is shown in the following figure, and an explanation of each field follows:

Field	Offset	Length	Access
Device Block Length	+00H	2	Public Read
Revision	+02H	1	Public Read
Secondary Device ID	+03H	1	Public Read
Logical ID	+04H	2	Public Read
Device ID	+06H	2	Public Read
Count of Logical ID Exclusive Port Pairs	+08H	2	Public Read
Count of Logical ID Common Port Pairs	+0AH	2	Public Read
Logical ID Exclusive Port Pairs 0	?	4	Public Read
Logical ID Exclusive Port Pairs 1	?	4	Public Read
.			
.			
Logical ID Exclusive Port Pairs **N**	?	4	Public Read
Logical ID Common Port Pairs 0	?	4	Public Read
Logical ID Common Port Pairs 1	?	4	Public Read
.			
.			
Logical ID Common Port Pairs **N**	?	4	Public Read
Device Unique Data Area Length	?	2	Private
Device Unique Data Area	?	?	Private
Count of Units	?	2	Private
Unit Unique Data Area Length	?	2	Private
Unit Unique Data Area	?	?	Private

? - is a placeholder for variable values
N - is the count of port pairs

Figure 30. Device Block

Device Block Length: This field is a word containing the number of bytes in the device block, including the Device Block Length field. The maximum specifiable length is 64KB minus 1. The required size of the device block for a particular device is returned during ABIOS initialization.

Revision: This byte is used to indicate the level of the supporting code for a device. The initial value of the base level is 0. For each succeeding version of ABIOS code for a particular Device ID and Secondary Device ID, the Revision field is increased by 1. That is, the Revision field is increased by 1 if a new level of ABIOS code is developed for existing hardware.

Secondary Device ID: This byte is used to determine the level of hardware that an ABIOS implementation supports. The initial value of the base level is 0. The Secondary Device ID field is increased by 1 when a new level of code is developed for a previously defined Device ID that supports new hardware. When the Secondary Device ID field is increased, the Revision field reverts to 0.

Logical ID: Logical ID indicates the logical name of the device associated with a Device Block. It is analogous to the software interrupt number used by BIOS to access different device types. Logical ID fields are built dynamically during initialization and the Logical ID for a given device is determined by the index of its entry in the Common Data Area.

Device ID: Device ID indicates the type of device addressed by a function request and the level of ABIOS function that is supported.

The assigned values of this field are shown in the following figure.

Device	Device ID Value
ABIOS Internal Calls	00H
Diskette	01H
Disk	02H
Video	03H
Keyboard	04H
Parallel Port	05H
Asynchronous Communications	06H
System Timer	07H
Real Time Clock Timer	08H
System Services	09H
Nonmaskable Interrupt	0AH
Pointing Device	0BH
Reserved	0CH
Reserved	0DH
Nonvolatile Random Access Memory (NVRAM)	0EH
Direct Memory Access (DMA)	0FH
Programmable Option Select (POS)	10H
Reserved	11H - 15H
Keyboard Security	16H
Reserved	17H - FFFFH

Figure 31. Device ID Values

The value hex 00 of the Device ID field is reserved for ABIOS internal calls; that is, ABIOS calling ABIOS.

Count of Logical ID Exclusive Port Pairs: This is the count of Logical ID exclusive port pairs. Logical ID exclusive ports are ports used exclusively by a particular Logical ID. Examples are the diskette ports, disk ports, asynchronous communication ports, parallel ports, and video ports. If the Count of Logical ID Exclusive Port Pairs field equals 0, no space is allocated for the Count of Logical ID Exclusive Port Pairs field.

Count of Logical ID Common Port Pairs: This is the count of Logical ID Common Port Pairs. The Logical ID common ports are ports that are shared across more than one Logical ID field. Examples are the DMA controller ports, keyboard controller ports, and NVRAM ports. Each Logical ID that uses one of these ports contains an entry in the Logical ID Common Port Pair fields of the device block. If this field equals 0, no space is allocated for the Logical ID Common Port Pair fields.

Logical ID Exclusive Port Pairs: These are the Logical ID Exclusive Port Pairs. The first word of the doubleword is the starting I/O port number of a range of I/O port numbers. The second word is the ending I/O port number of the range.

Logical ID Common Port Pairs: These are the Logical ID common port pairs. The first word of the doubleword is the starting I/O port number of a range of I/O port numbers. The second word is the ending I/O port number of the range.

Note: Every port that an ABIOS Logical ID inputs from or outputs to is contained in either the Logical ID exclusive port pair fields or the Logical ID common port pair fields.

Device Unique Data Area Length: This field contains the length, in bytes, of the device unique data area for this device.

Device Unique Data Area: This field is reserved for data unique to a device. Parameters describing the device and working data that span the Device ID are kept in this area. This area contains private data for ABIOS and its content or format may change. Examples of the data kept in this area are interrupt level, arbitration level, and device status.

Count of Units: This field contains the count of the unit unique data areas in the device block. If this field equals 0, the Count of Units field is the last field in the device block.

Unit Unique Data Area Length: This field contains the length, in bytes, of a single entry in the repeatable unit unique data area, excluding the Unit Unique Data Area Length field. This field exists only if the Count of Units field is greater than 0.

Unit Unique Data Area: This field is a private repeatable area reserved for each unit of the Device ID. For example, if diskette is the Device ID, a particular diskette drive is a unit. Parameters describing the unit and working data that span individual requests are kept in this area. This area is private data for ABIOS and its content or format may change. This field exists only if the Count of Units field is greater than 0.

Initialization

ABIOS is initialized in an "on demand" fashion. The Operating System makes specific calls to BIOS and ABIOS to achieve the startup. The real mode common data area must be initialized before any requests can be made to ABIOS. Initialization is performed in the real mode of the microprocessor, and includes building the System Parameters Table, the Initialization Table, and the ABIOS Common Data Area.

Figure 32 shows the flow of the real mode common data area initialization.

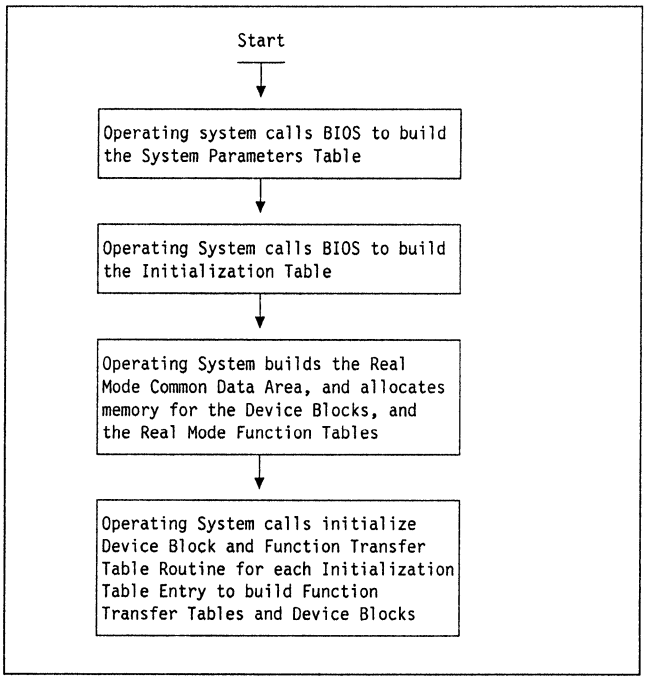


Figure 32. Flow of Real Mode Common Data Area Initialization

Build System Parameters Table

The operating system allocates a hex 20-byte area and calls BIOS to build the system parameters table. This table describes the number of devices available in the system, the ABIOS common entry points, and system stack requirements.

```
INT 15H, (AH) = 04H BUILD SYSTEM PARAMETERS TABLE

Invocation:
Software Interrupt, Operating System calls BIOS.

(ES:DI) = Pointer to caller's memory
                where System Parameters Table is
                to be built.
   (DS) = Segment with assumed 0 Offset
               to RAM extension area (points to
               a RAM extension with length=0 for
               no RAM extensions).
On Return:
   (CY) = 1 Indicates an exception error
   (AH) = Error code, 0 for no errors
   (All registers except AX and FLAGS are restored.)
```

Figure 33. Build System Parameters Table BIOS Function

Once the system parameters table information is obtained, the memory allocated for this table may be deallocated and reused by the operating system.

The system parameters table is shown below:

Field	Offset	Length
Common Start Routine Pointer	+00H	4
Common Interrupt Routine Pointer	+04H	4
Common Time Out Routine Pointer	+08H	4
Stack Required	+0CH	2
Reserved	+0EH	4
Reserved	+12H	4
Reserved	+16H	4
Reserved	+1AH	4
Number of Entries	+1EH	2

Figure 34. System Parameters Table

The system parameters table entries are described in more detail below:

Common Start Routine Pointer: This is a doubleword address pointer to the Common Start routine entry point.

Common Interrupt Routine Pointer: This is a doubleword address pointer to the Common Interrupt routine entry point.

Common Time Out Routine Pointer: This is a doubleword address pointer to the Common Time Out routine entry point.

Stack Required: This field is a word containing the amount of stack memory, in bytes, that is required for the particular ABIOS implementation.

Number of Entries: This field is a word containing the number of entries required in the initialization table.

Build Initialization Table

The initialization table defines the initialization information for each device the system supports. This information is used to initialize the device blocks and the function transfer tables.

The operating system allocates memory and calls BIOS to build the initialization table. The amount of memory required for the initialization table in bytes is hex 18 times the number of entries in the initialization table. The Number of Entries field in the system parameters table is used for this calculation. When the initialization process is complete the memory allocated for the initialization table can be deallocated and reused by the operating system.

```
INT 15H, (AH) = 05H BUILD INITIALIZATION TABLE

Invocation:
Software interrupt, operating system calls BIOS.

(ES:DI) = Pointer to caller's memory
                where initialization table is to
                be built.
(DS)    = Segment with assumed 0 offset
                to RAM extension area (points to
                a RAM extension with length=0 for
                no RAM extensions).
On Return:
   (CY) = 1 Indicates exception error
   (AH) = Error code
(All registers except AX and FLAGS are restored.)
```

Figure 35. Build Initialization Table BIOS Function

The initialization table structure, shown in the following figure, is repeated for each entry:

Field	Offset	Length
Device ID	+00H	2
Number of Logical IDs	+02H	2
Device Block Length	+04H	2
Initialize Device Block and Function Transfer Table Routine Pointer	+06H	4
Request Block Length	+0AH	2
Function Transfer Table Length	+0CH	2
Data Pointers Length	+0EH	2
Secondary Device ID	+10H	1
Revision	+11H	1
Reserved	+12H	2
Reserved	+14H	2
Reserved	+16H	2

Figure 36. Initialization Table

The Initialization Table entries are described in more detail below.

Device ID: There may be more than one entry in the initialization table with the same Device ID.

Number of Logical IDs: This is a word containing the maximum number of devices that require individual device blocks but are operated by the same code. The Number of Logical IDs field tells the Operating System the maximum number of Logical IDs that this initialization table entry allows.

Device Block Length: This is a word containing the length, in bytes, of the storage allocation required for the device block for this device.

Initialize Device Block and Function Transfer Table Routine: This is a doubleword address pointer (real mode segment:offset) to the routine to initialize the device blocks and function transfer tables for an entry in the initialization table.

Request Block Length: This is a word containing the length, in bytes, of the storage allocation required for the Request Block for this device. When making a request to ABIOS, any Request Block size greater than the size returned is valid.

Function Transfer Table Length: This is a word containing the length, in bytes, of the function transfer table.

Data Pointer Length: This is a word containing the length, in bytes, of the storage allocation required for the Data Pointer fields in the common data area.

Secondary Device ID: This is a byte used to determine the level of hardware that an ABIOS implementation supports.

Revision: This byte is used to indicate the level of the supporting code for this device.

Build Common Data Area

After the system parameters table and the initialization table are built, the Operating System has all the necessary information required to build the ABIOS common data area and its associated data structures. The size of the common data area, the size of each function transfer table, and the size of each device block can be determined from the initialization table.

The operating system builds the common data area at offset 0 within a segment, and allocates memory for each device block and each function transfer table. Memory is allocated within the common data area for the data pointers. The offset to the Data Pointer 0 field is initialized to point to the Data

Pointer Length 0 field within the common data area. The Data Pointer Count field is initialized to 0. The Count of Logical IDs field is filled in with the number of device block and function transfer table pointer pairs. Each device block pointer and each function transfer table pointer is initialized to point to the memory that has been allocated.

Logical ID numbers are assigned by their order in the initialization table. For example, if the Number of Logical IDs field is 1 for each entry in the initialization table, the first entry corresponds to Logical ID 2, the second entry corresponds to Logical ID 3, and so on. If the Number of Logical IDs field is greater than 1 for the first initialization table entry, that entry corresponds to Logical ID 2 through Logical ID 2 plus the Number of Logical IDs field - 1. The second initialization table entry corresponds to the next succeeding Logical ID.

Multiple function transfer table pointers can point to the same function transfer table. This occurs when the Number of Logical IDs field in an entry in the initialization table entry is greater than 1. The Operating System must ensure that the function transfer table pointers for the succeeding Logical IDs, corresponding to a single initialization table entry, point to the same function transfer table.

The operating system calls the Initialize Device Block and Function Transfer Table routine once for each entry in the initialization table. The Operating System passes the following parameters: the Anchor Pointer, the Starting Logical ID, and the Number of Logical IDs to Initialize.

```
        INITIALIZE DEVICE BLOCK AND
      FUNCTION TRANSFER TABLE ROUTINE

Invocation:
Call FAR, operating system calls
ABIOS on system board ROM, depending on device.

  (CX) = Number of Logical IDs to Initialize
           (up to the Number of Logical IDs
           from initialization table)
   (DX) = Starting Logical ID
   (DS) = Anchor pointer to beginning of
           common data area
On Return:
  (AL) = Exception condition, 0 for no errors
           00         Successful completion
           01-FF   Device initialization failure
All registers except AX are restored.
```

Figure 37. Initialize Device Block and Function Transfer Table Routine

Build Protected Mode Tables

For protected mode or bimodal implementations, it is necessary to build the protected mode Common Data Area and Function Transfer Tables using the information built in the real mode Common Data Area and Function Transfer Tables. The operating system must create selectors in the protected mode Common Data Area and Function Transfer Tables whose effective address is identical to their corresponding segments in the real mode Common Data Area and Function Transfer Tables.

The following diagram describes the steps necessary to build the protected mode common data area:

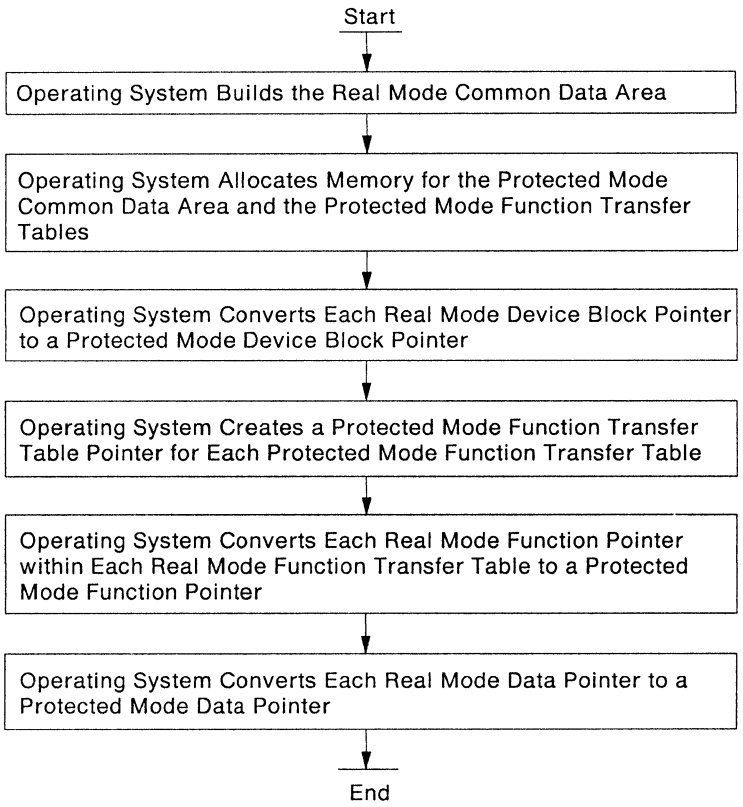


Figure 38. Flow of Protected Mode Common Data Area Initialization

Request Block

The ABIOS Request Block is a parameter block used to communicate information bidirectionally between the caller and an ABIOS service. Parameters are passed by the caller (IN) and returned by ABIOS (OUT).

The following diagram shows the Request Block and its relationship with a Common Data Area:

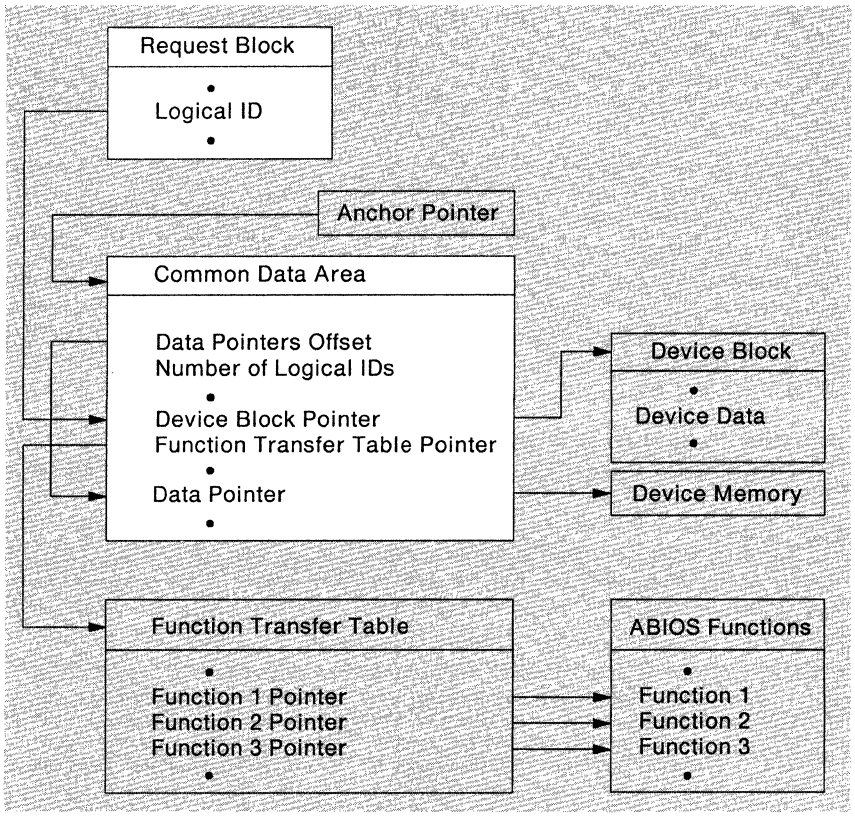


Figure 39. Flow of Request Block

All input parameters (IN) are unaltered by ABIOS throughout the duration of a request. All output parameters (OUT) and work areas need not be set to any predefined value before ABIOS is called. This allows Request Blocks to be reused after requests are completed. This requires that any work area fields containing request state information be initialized by the ABIOS Start routines to the predefined values. Only input (IN) or input/output (IN/OUT) parameters that change between requests are required to be initialized before the Request Block is re-used. All reserved input fields must be set to 0 by the caller of ABIOS. The parameters are divided into two categories: functional parameters and service specific parameters.

Functional Parameters

Functional parameters are common to all ABIOS service requests. They convey information to ABIOS about which service should be invoked on which device. Each input parameter is initialized by the caller and, once initialized, must remain unaltered until the requested operation is complete. Functional parameters include the Request Block Length field through the Time Out field.

Service Specific Parameters

Service specific parameters are specific to an ABIOS request. The details of parameters passed by the caller and parameters returned by ABIOS depend on the service requested. The service specific parameters include the Data Pointer 1 field through the Work Area field.

Request Block Structure

The structure of a Request Block containing functional parameters and service specific parameters is shown below:

Field	Offset	Length
Functional Parameters		
Request Block Length (IN)	+00H	2
Logical ID (IN)	+02H	2
Unit (IN)	+04H	2
Function (IN)	+06H	2
Reserved	+08H	2
Reserved	+0AH	2
Return Code (IN/OUT)	+0CH	2
Time Out (OUT)	+0EH	2
Service Specific Parameters		
Reserved	+10H	2
Data Pointer 1 (IN)	+12H	4
Reserved	+16H	2
Reserved	+18H	2
Data Pointer 2 (IN)	+1AH	4
.		
.		
.		
Reserved	+10H+(N-1)*8	2
Data Pointer N (IN)	+12H+(N-1)*8	4
Parameters (IN/OUT)	+18H+(N-1)*8	?
Work Area	?	?

? - undefined initial value

Figure 40. Request Block

Request Block Length (IN): The Request Block Length field contains the length, in bytes, of the Request Block including the Request Block Length field itself. The maximum specifiable length is 64KB minus 1. Request block length is a fixed value initialized by the caller for the specific Logical ID. The size of the Request Block for a Logical ID is returned at ABIOS initialization time and by the Return Logical ID Parameters function (hex 01). However, the Request Block may be larger than the returned size.

Logical ID (IN): The Logical ID field indicates the particular device that is addressed by a function request. It is analogous to a software interrupt number used by BIOS to access different device types.

Unit (IN): The Unit field is a parameter that addresses a particular unit of a device within a Logical ID. The range of valid values is limited by the number of units attached to a single controller. The maximum unit number is n-1, where n is the count of units attached to the controller. The minimum number of units is one, resulting in a Unit field equal to 0.

Function (IN): The Function field is a parameter used to request a particular category of operation. The assignment of functions is:

Function *Function Performed*

00H Default Interrupt Handler - This function is called with no service specific parameters for each Logical ID by way of the Interrupt routine. The Request Block for the default interrupt handler has a fixed length of hex 10 bytes, and the Return Code field is updated on return with hex 0000 for Operation Completed Successfully or hex 0005 for Not My Interrupt. This handler is used to process spurious interrupts in the system.

01H Return Logical ID Parameters - This function is a standard, single-staged function common to all ABIOS Device IDs. It returns information pertaining to the Logical ID and its Request Block has a fixed length of hex 20 bytes. It returns the following parameters:

Service Specific Input

Size	Offset	Description
Word	1AH	Reserved
Word	1CH	Reserved
Word	1EH	Reserved

Service Specific Output

SIZE	OFFSET	DESCRIPTION
Byte	10H	Hardware interrupt level FFH = non-interrupting Logical ID FEH = special case for NMI
Byte	11H	Arbitration level FFH = not applicable
Word	12H	Device ID
Word	14H	Count of Units
Word	16H	Logical ID flags Bit 15-4 = Reserved Bit 3 = Overlapped I/O across units 0 - Not supported 1 - Supported Bit 2 = Reserved Bits 1-0 = Function Read/Write/Additional Data Transfer Data Pointer mode 00 - No Read/Write/Additional Data Transfer Functions supported 01 - Data Pointer 1, Logical Data Pointer 2, Reserved 10 - Data Pointer 1, Reserved Data Pointer 2, Physical 11 - Data Pointer 1, Logical Data Pointer 2, Physical
Word	18H	Request Block Length For functions other than Default Interrupt Handler and Return Logical ID Parmeters. Variable by Logical ID.
Byte	1AH	Secondary Device ID
Byte	1BH	Revision
Word	1CH	Reserved
Word	1EH	Reserved

Logical ID flags contain 2 bits that indicate the mode (physical vs. logical) of the data pointer for the Read (hex 08), the Write (hex 09), and the Additional Data Transfer (hex 0A) functions. If this

parameter indicates that the pointer should be a logical pointer, then Data Pointer 1 is a logical pointer and Data Pointer 2 is reserved. If this parameter indicates that the pointer should be a physical pointer, Data Pointer 2 is a physical pointer and Data Pointer 1 is reserved. If this parameter indicates that both a logical pointer and a physical pointer are to be passed, Data Pointer 1 is the logical pointer and Data Pointer 2 is a physical pointer. If the parameter indicates neither, this Logical ID does not support functions hex 08, 09, and 0A, or these functions require no address pointers. There is no space reserved for data pointers in the Request Block in this event.

02H Reserved.

03H Read Device Parameters - device specific parameters are returned.

04H Set Device Parameters - device specific parameters are set.

05H Reset/Initialize - device is placed in a known state.

06H Enable - device is enabled for interrupts (not at interrupt controller).

07H Disable - device is disabled for interrupts (not at interrupt controller).

08H Read - data is transferred from device to memory. Data Pointer mode is determined by the function Return Logical ID Parameters.

09H Write - data is transferred from memory to device. Data Pointer mode is determined by the function Return Logical ID Parameters.

0AH Additional Data Transfer Function - Data Pointer mode is determined by the function Return Logical ID Parameters.

0BH and up Additional Functions - as necessary.

Return Code (IN/OUT): Return code is a field that contains the results of the current stage of the requested operation. For those operations that are single-staged or those that are on the final stage of a discrete multistaged operation, the Return Code field indicates the results of the entire operation.

The values for the Return Code field are shown in the following figure.

Return Code Values	Definition
0001H	Stage On Interrupt
0002H	Stage on Time
0005H	Not My Interrupt, Stage On Interrupt
0009H	Attention, Stage On Interrupt
0081H	Unexpected Interrupt Reset, Stage On Interrupt
8000H	Device in Use, Request Refused
8001-8FFFH	Service Specific Unsuccessful Operation
9000-90FFH	Device Error
9100-91FFH	Retryable Device Error
9200-9FFFH	Device Error
A000-A0FFH	Timeout Error
A100-A1FFH	Retryable Timeout Error
A200-AFFFH	Timeout Error
B000-B0FFH	Device Error with Timeout
B100-B1FFH	Retryable Device Error with Timeout
B200-BFFFH	Device Error with Timeout
C000H	Invalid Logical ID
C001H	Invalid Function
C002H	Reserved
C003H	Invalid Unit Number
C004H	Invalid Request Block Length
C005-C01FH	Invalid Service Specific Parameter
C020-FFFEH	Service Specific Unsuccessful Operation
FFFFH	Return Code Field Not Valid

Figure 41. Possible Values for Return Code Fields

The bits within the Return Code field are defined in the following figure.

Bit	Definition
0	Stage On Interrupt
1	Stage on Time
2	Not My Interrupt
3	Attention
4 to 6	Reserved
7	Unexpected Interrupt Reset
8	Retryable Error
9 to 11	Reserved
12	Device Error
13	Timeout Error
14	Parameter Error
15	Unsuccessful Operation

Notes:

1. Bits 0-7 are defined as above only when Bit 15 equals 0.
2. Bits 8-14 are defined as above only when Bit 15 equals 1.
3. If all bits equal 1, the Return Code field is not valid.

Figure 42. Return Code Field Bit Definitions

The caller of ABIOS must initialize the Return Code field to Return Code Field Not Valid (hex FFFF) before calling any ABIOS Start routine. If the Operating System has an outstanding Request Block at interrupt time, it first checks for a Return Code field equal to Return Code Field Not Valid (hex FFFF), and if it is, the Operating System considers the Return Code field as not set and does not attempt to resume this request. The ABIOS routine sets the Return Code

field to its appropriate value when the interrupt is expected.

When ABIOS is processing a request that causes a hardware interrupt, interrupts are disabled between the time of writing to the interrupt enable port and changing the Return Code field from a value of Return Code Field Not Valid (hex FFFF) to a Return Code value of Stage On Interrupt bit (bit 0) set. After changing the Return Code field, the interrupt flag is restored to the value contained prior to disabling it.

When the hardware interrupt occurs, the caller only responds to those requests that have a value of the Return Code field with the Stage On Interrupt bit (bit 0) set. In other words, the outstanding requests with a Return Code field equal to Return Code Not Valid (hex FFFF) are not called.

The caller should also maintain a flag that indicates whether or not a request has completed the Start routine to the point at which the Return Code field is interrogated. This allows for the situation when the interrupt occurs after the Return Code field is valid (not hex FFFF) but before the Return Code field is interrogated by the caller. At this point there could be a Start routine and an Interrupt routine operating on the same Request Block within different stack frames, necessitating the caller's flag.

Attention (hex 0009) and Stage on Time (hex 0002) are values of the Return Code field that need only be tested by services that require them. Attention (hex 0009) indicates that there is data available in a service specific output parameter although the function is not complete. Stage on Time (hex 0002) indicates that the operation is incomplete and must be resumed when a certain amount of time has elapsed. This amount of time is contained in a service specific output parameter depending on the service. In addition, the values of the Return Code field with Bit 15 equal to 1 are service specific.

The return code value Device In Use, Request Refused (hex 8000) is used for device serialization. If a Logical ID/Unit combination is a serially reusable device, ABIOS returns this return code value when there is an outstanding request on this device.

Time Out (OUT): The Time Out field contains the expected duration of the requested stage. This is used to detect when an operation has timed out and needs to be reset by the Time Out routine. The unit of time is 1 second, and the value occupies bits 15 through 3. Bits 2 through 0 of this field are reserved. A value of 0 indicates the operation has no timeout value. The Time Out field is valid for Stage On Interrupt (hex 0001), when the Stage on Interrupt bit (bit 0) is set.

Parameters (IN/OUT): Parameters communicate operands and, in some cases, results of ABIOS functions. Parameter requirements vary by device and function requested.

Data Pointer 1 and Data Pointer 2 (IN): Data Pointers, if required, are doubleword pointers to the I/O buffer area for this request. The effective address must be addressable in the current mode of the microprocessor in a bimodal environment. The address may be a 32-bit physical address for DMA, or segmented for programmed I/O. Return Logical ID Parameters returns a parameter that indicates the mode (physical or logical) of the data pointer for the functions Read (hex 08), Write (hex 09), and Additional Data Transfer (hex 0A). If this parameter indicates that the pointer should be a logical pointer, Data Pointer 1 is a logical pointer and Data Pointer 2 is reserved. If this parameter indicates that the pointer should be a physical pointer, Data Pointer 2 is a physical pointer and Data Pointer 1 is reserved. If this parameter indicates that both a logical pointer and a physical pointer are to be passed, Data Pointer 1 is the logical pointer and Data Pointer 2 is a physical pointer. If the parameter indicates neither, this Logical ID does not support functions hex 08, 09 and 0A, or these functions require no address pointers. No space is reserved for data pointers in the Request Block in this event.

Work Area: Work Area fields are an optional data area reserved for ABIOS. No user data may be stored here. Their contents is variable by the type of request and the particular device routine involved. These fields are not required to be initialized to any value. Their content must not be altered by the caller of ABIOS across multistaged requests. Work Area fields are those fields that are not defined as service specific input or service specific output parameters.

ABIOS Transfer Convention

The ABIOS Transfer Convention places the requirement on ABIOS to access the effective address of a particular ABIOS function. ABIOS indexes into the Common Data Area based upon the Logical ID to access the necessary pointers including the effective Function Routine pointer (Start, Interrupt, or Time Out). The ABIOS Transfer Convention is the simplest calling sequence for the operating system.

The flow of an ABIOS Transfer Convention request is shown below:

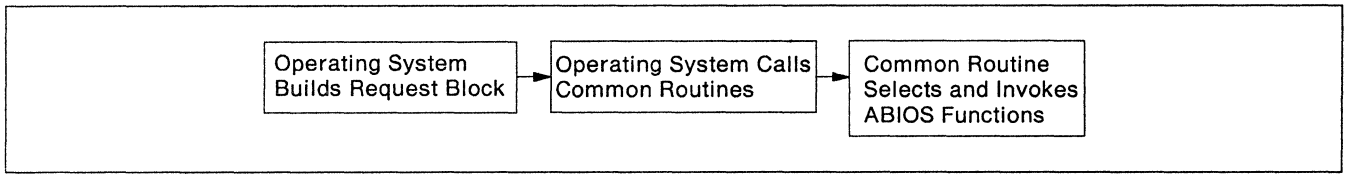


Figure 43. Flow of ABIOS Transfer Convention

For this transfer convention, there are only three routines by which the caller can transfer control to ABIOS. The pointers to these 3 routines are returned in the System Parameters Table at initialization time. They are also contained in the function transfer table for Logical ID 2. These routines are:

- **Common Start Routine** - This routine is called (using a call far indirect) to start a request. The Logical ID field within the Request Block is validated. If this Logical ID value is greater than the value of the Count of Logical IDs field in the common data area, or if this Logical ID value pertains to a null common data area entry, the Return Code field is set to Invalid Logical ID (hex C000).
- **Common Interrupt Routine** - This routine is called (using a call far indirect) to resume a multistaged request.
- **Common Time Out Routine** - This routine is called (using call far indirect) to terminate a request that fails to receive a hardware interrupt in a specified time. The Time Out routine aborts the request and leaves the hardware controller in a known, initial state.

The parameter passing convention for the ABIOS Transfer Convention is a set of two parameters, two reserved doublewords, and a return address on the stack. The first parameter is the common data area anchor pointer segment or selector with assumed 0 offset. The second parameter is the doubleword pointer to the request block. The third parameter is a reserved doubleword placeholder for the function transfer table pointer. The fourth parameter is a reserved doubleword placeholder for the Device Block pointer.

The ABIOS common routines expect the addresses from high to low (the order of pushing) as shown in the following figure:

Contents	Displacement (from Stack Pointer)
Return Address	+00H
Reserved for Device Block Pointer	+04H
Reserved for Function Transfer Table Pointer	+08H
Request Block Pointer	+0CH
Common Data Area Anchor Pointer (Segment or Selector only)	+10H

Figure 44. ABIOS Transfer Convention Stack Frame

The following pseudo code instructions are suggested:

```
PUSH    Anchor Pointer Segment or Selector
PUSH    Request Block Segment or Selector
PUSH    Request Block Offset
SUB     Stack Pointer,8
CALL    Common Start Routine
```

Pseudo Code - ABIOS Transfer Convention

The common routines use the Logical ID from the Request Block and the Anchor pointer to determine which Device Block pointer and Function Transfer Table pointer pair is to be used. These routines take this pair of pointers and place them in the stack placeholder positions allocated by the caller. Then the common routines transfer control to the Start, Interrupt or Time Out routines whose pointers are contained in the function transfer table for the requested Logical ID field. The common data area segment or selector, the Request Block pointer, the Function Transfer Table pointer and the device block pointer are passed on the stack as in the Operating System Transfer Convention. For the ABIOS Transfer Convention, it is the responsibility of the caller to remove the parameters from the stack upon return.

Operating System Transfer Convention

This convention places the requirement on the Operating System to determine the effective address of a particular routine. This method is most useful for handling interrupts from character and programmed I/O devices that call a single routine repeatedly.

There are two different methods to accomplish operating system transfers. In the first method, the Operating System indexes into the common data area based upon the Logical ID to access the necessary pointers including the effective Routine pointer (Start, Interrupt, or Time Out). The advantage of this approach over the ABIOS Transfer Convention is one of performance.

In the second method, the operating system stores the necessary pointers as it sees fit and accesses them without indexing into the common data area. An Operating System might want to use this method if that Operating System is a real-mode-only or protected-mode-only Operating System. The common data area is provided to access the necessary pointers as quickly as possible in a bimodal environment. In a single mode environment there is no advantage to accessing the pointers by indexing into the common data area. In fact there is a small performance loss.

The flow of an Operating System Transfer Convention request is shown below:

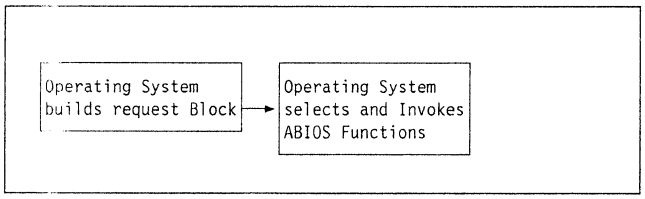


Figure 45. Flow of Operating System Transfer Convention

The parameter passing convention for the Operating System Transfer Convention is a set of four parameters and a return address on the stack. The first is the anchor pointer segment or selector of the Common Data Area with assumed 0 offset. The second is a doubleword pointer to the Request Block. The third is a doubleword pointer to the function transfer table, and the fourth is a doubleword pointer to the Device Block.

The Start, Interrupt, and Time Out routines for each Logical ID expect the addresses from high to low (the order of pushing) as shown in the following figure:

Contents	Displacement (from Stack Pointer)
Return Address	+00H
Device Block Pointer	+04H
Function Transfer Table Pointer	+08H
Request Block Pointer	+0CH
Common Data Area Anchor Pointer (Segment or Selector only)	+10H

Figure 46. Operating System Transfer Convention Stack Frame

The following pseudo code instructions are suggested:

```
PUSH    Anchor Segment or Selector
PUSH    Request Block Segment or Selector
PUSH    Request Block Offset
PUSH    Function Transfer Table Segment or Selector
PUSH    Function Transfer Table Offset
PUSH    Device Block Segment or Selector
PUSH    Device Block Offset
CALL    Logical ID Start Routine
```

Pseudo Code - Operating System Transfer Convention

For the Operating System transfer convention, it is the responsibility of the caller to remove the parameters from the stack upon return.

Interrupt Processing

The Operating System that interfaces with ABIOS provides interrupt handlers that receive control through the hardware interrupt vector. The interrupt handler is required to retain the Logical IDs of the devices that operate on the given interrupt level. ABIOS provides routines that are called by the Operating System interrupt handlers.

Each device has a Logical ID that is known to the Operating System. A Logical ID may have one or more Request Blocks active when the interrupt is processed by the interrupt handler in the Operating System. Each active Request Block of the Logical ID is processed by calling ABIOS at its interrupt entry point. The Return Code field is set by ABIOS to indicate whether or not the interrupt was associated with the Request Block.

The Operating System can call ABIOS for interrupt processing with interrupts enabled or disabled. ABIOS restores the state of the interrupt flag after any period that interrupts are required to be disabled. If there are no Request Blocks that have the Stage On Interrupt bit (bit 0) of the Return Code field set to 1, and an interrupt occurs, the default interrupt handler is provided to remove the interrupt at the device.

Interrupt Sharing

Where more than one Logical ID or Logical ID-unit combination share an interrupt level, the process is repeated for each Logical ID until all are processed or the first Logical ID with an interrupt is completely processed.

ABIOS expects the Operating System to manage the end of interrupt (EOI) processing at the interrupt controller. The method used for EOI processing is entirely up to the Operating System. ABIOS does not reset the interrupt controller. The Operating System can choose its desired strategy for resetting the interrupt controller after all outstanding Request Blocks for a particular Logical ID are processed through the Interrupt routine and at least one request responds that the interrupt was serviced. A serviced interrupt request returns from the Interrupt routine with the Return Code field having any value other than Not My Interrupt, Stage On Interrupt (hex 0005).

ABIOS Rules

The following rules are presented for programmers writing operating systems and device drivers.

Rule 1 The Device Block Pointer field, the Function Transfer Table Pointer field and all Data Pointer fields for a given Logical ID within the ABIOS Common Data Area must not be altered by the Operating System during a particular stage of a request to that Logical ID.

Rule 2 ABIOS, after being interrupted within a given stage of a request, returns to that stage in the mode that it was running at the time of interrupt.

Rule 3 ABIOS device blocks are owned by ABIOS and only the public portions are accessed by the Operating System. There is no guarantee of compatibility of the Device Block private area contents across ABIOS implementations.

Rule 4 ABIOS Request Blocks are shared by ABIOS and the operating system.

Rule 5 ABIOS must traverse the Common Data Area to retrieve necessary pointers. It must not store pointers in one request or stage of a request to be used on another request or stage of a request.

Rule 6 ABIOS function X, running in protected mode, can be interrupted, and function Y can be invoked in real mode, and vice versa. X can equal Y. After being preempted in the middle of a request stage in mode X, ABIOS can be called through the START routine in mode Y.

Rule 7 ABIOS does not change the state of the interrupt flag. ABIOS can temporarily disable the interrupt flag, but must restore it to its original state.

Rule 8 A Request Block pointer that is passed on a request is valid for the duration of that stage of the request.

Rule 9 The effective memory address of a physical address pointer must not be moved for the duration of a single request. When a function requires the data pointer to be passed as a physical address within memory it is assumed that an external process is performing the read or write to memory, therefore across stages this address cannot change.

Rule 10 The effective memory address of a logical address pointer (those pointers in the Request Block in the form segment:offset or selector:offset) can be changed or moved across stages of a request. In real mode, the segment and/or offset can be changed. In protected mode the selector and/or offset can be changed as well as the physical address located in the descriptor.

Rule 11 ABIOS does not do End of Interrupt processing. In a level-sensitive interrupt environment, the device condition causing the interrupt is reset by ABIOS.

Rule 12 The caller of ABIOS can perform EOI processing when the Return Code field is any value other than Not My Interrupt, Stage On Interrupt (hex 0005) and all Request Blocks are serviced on the Logical ID. The caller can assume that the interrupt was serviced and process the EOI.

Rule 13 The caller of ABIOS must call each outstanding request per Logical ID at interrupt time until the first Logical ID with an interrupting condition is completely processed.

Rule 14 Operating System device numbers are allocated by the Operating System based on increasing units within increasing Logical IDs. For example; the first Logical ID with

Device ID = printer, unit 0 is lpt1:, unit 1 is lpt2:. If unit 1 does not exist, the second Logical ID with Device ID = printer, unit 0 is named lpt2:, and so on.

Rule 15 ABIOS in a protected mode or bimodal implementation must have I/O privilege when operating in protected mode.

Sample Interfaces

This section describes the service specific parameters for Disk and Video ABIOS interfaces. Each interface description includes the interface functions and the values of the return code field. Programming considerations are also included where appropriate.

Parameters are passed to ABIOS functions in the request block. All input parameters are set by the caller and all output parameters are returned by ABIOS functions.

Disk

Functions

The following shows the disk functions with an explanation of each function performed.

00H - Default Interrupt Handler

01H - Return Logical ID Parameters

02H - Reserved

03H - Read Device Parameters

- This function returns disk drive information based on the unit requested and the disk device control information.
- Possible return codes = hex 0000.

Service Specific Input

Size	Offset	Description
Word	28H	Reserved

Service Specific Output

Size	Offset	Description
Word	10H	Sectors/Track associated with unit in request block
Word	12H	Size of sectors in bytes 02H - 512-byte sectors All other values are reserved
Word	14H	Device control flags Bits 15 to 13 - Reserved Bits 12, 11 - Format support (values in binary) 00 - Format not supported 01 - Format track supported 10 - Format unit supported 11 - Format track/unit supported Bit 10 - ST506 Drive 0 - Not ST506 1 - ST506 Bit 9 - Concurrent unit requests per Logical ID 0 - Not concurrent 1 - Concurrent Bit 8 - Ejecting capability 0 - Not ejectable 1 - Ejectable Bit 7 - Media organization 0 - Random 1 - Sequential Bit 6 - Locking capability 0 - Not lockable 1 - Lockable Bit 5 - Read capability 0 - Not readable 1 - Readable Bit 4 - Caching support 0 - No caching 1 - Caching Bit 3 - Write frequency 0 - Write once 1 - Write many Bit 2 - Change signal support 0 - No change signal supported 1 - Change signal supported Bits 1, 0 - Reserved
DWord	18H	Physical number of cylinders associated with unit in request block
Byte	1CH	Physical number of heads associated with unit in request block
Byte	1DH	Suggested number of software retries for retryable operations
DWord	20H	Number of relative block addresses associated with unit in request block
DWord	24H	Reserved
Word	28H	Reserved
Word	2CH	Maximum number of blocks to transfer per one call

04H - Set Device Parameters (Reserved)

05H - Reset/Initialize

- This function resets the disk system to an initial state.
- All Return Code values listed in the Disk Return Codes table are possible for this function.

Service Specific Input

Size	Offset	Description
Word	10H	Reserved

Service Specific Output

Size	Offset	Description
DWord	28H	Time to wait before resuming request in microseconds

06H - Enable (Reserved)

07H - Disable (Reserved)

08H - Read

- The Read function transfers data from the specified relative block address to the specified memory location. The Number of Blocks to Read field contains the amount of data to transfer.
- If the Number of Blocks to Read field is 0, no action is performed.
- If the Number of Blocks to Read field is greater than the maximum number of blocks, then no action is performed. The Number of Blocks to Read field contains the amount of data transferred.
- When a parameter error is returned, the Number of Blocks to Read field is not updated.
- All Return Code values listed in the Disk Return Codes Table are possible for this function.

Service Specific Input

Size	Offset	Description
Word	10H	Reserved
DWord	12H	Data pointer 1
Word	16H	Reserved
Word	18H	Reserved
DWord	1AH	Data pointer 2
Word	1EH	Reserved
DWord	20H	Relative block address
DWord	24H	Reserved
Word	2CH	Number of blocks to read
Byte	2EH	Bits 7 to 1 - Reserved (set to 0) Bit 0 - Caching 0 - Caching is OK for this request 1 - Don't cache on this request

Service Specific Output

Size	Offset	Description
DWord	28H	Time to wait before resuming request in microseconds
Word	2CH	Number of blocks read
Word	2FH	Indicates if a soft error occurred = 0 - Soft error did not occur ≠ 0 - Soft error occurred

09H - Write

- The Write function transfers data from the specified relative block address to the specified memory location. The Number of Blocks to Write field contains the amount of data to transfer.
- If the Number of Blocks to Write field is 0, no action is performed.
- If the Number of Blocks to Write field is greater than the maximum number of blocks, no action is performed.
- The Number of Blocks Written field contains the amount of data transferred.
- When a parameter error is returned, the Number of Blocks Written field is not updated.
- All Return Code values listed in the Disk Return Codes table are possible for this function.

Service Specific Input

Size	Offset	Description
Word	10H	Reserved
DWord	12H	Data pointer 1
Word	16H	Reserved
Word	18H	Reserved
DWord	1AH	Data pointer 2
Word	1EH	Reserved
DWord	20H	Relative block address
DWord	24H	Reserved
Word	2CH	Number of blocks to write
Byte	2EH	Bits 7 to 1 - Reserved (set to 0) Bit 0 - Caching 0 - Caching is OK for this request 1 - Don't cache on this request

Service Specific Output

Size	Offset	Description
DWord	28H	Time to wait before resuming request in microseconds
Word	2CH	Number of blocks written
Word	2FH	Indicates if a soft error occurred = 0 - Soft error did not occur ≠ 0 - Soft error occurred

0AH - Write Verify

- The Write Verify function operates similar to the Write function (hex 09) with the addition of a Verify function (hex 0B).
- If the Number of Blocks to Write/Verify field is 0, no action is performed.
- If the Number of Blocks to Write/Verify field is greater than the maximum number or blocks, then no action is performed.
- The Number of Blocks Written field contains the amount of data transferred.
- When a parameter error is returned, the Number of Blocks Written field is not updated.
- All Return Code values listed in the Disk Return Codes Table are possible for this function.

Service Specific Input

Size	Offset	Description
Word	10H	Reserved
DWord	12H	Data Pointer 1
Word	16H	Reserved
Word	18H	Reserved
DWord	1AH	Data Pointer 2
Word	1EH	Reserved
DWord	20H	Relative block address
DWord	24H	Reserved
Word	2CH	Number of blocks to write/verify
Byte	2EH	Bits 7 to 1 - Reserved (set to 0) Bit 0 - Caching 0 - Caching is OK for this request 1 - Don't cache on this request
Word	31H	Reserved for subfunction

Service Specific Output

Size	Offset	Description
DWord	28H	Time to wait before resuming request in microseconds
Word	2CH	Number of blocks written
Word	2FH	Indicates if a soft error occurred = 0 - Soft error did not occur ≠ 0 - Soft error occurred

0BH - Verify

- The Verify function reads from the specified relative block address without transferring any data to system memory. This function verifies the readability of the data.
- If the Number of Blocks to Verify field is 0, no action is performed.
- If the Number of Blocks to Verify field is greater than the maximum number of blocks, no action is performed.
- All Return Code values listed in the Disk Return Codes Table are possible for this function.

Service Specific Input

Size	Offset	Description
Word	16H	Reserved
Word	18H	Reserved
Word	1EH	Reserved
DWord	20H	Relative Block Address
DWord	24H	Reserved
Word	2CH	Number of blocks to verify

Service Specific Output

Size	Offset	Description
DWord	28H	Time to wait before resuming request in microseconds
Word	2FH	Indicates if a soft error occurred = 0 - Soft error did not occur ≠ 0 - Soft error occurred

0CH - Interrupt Status

- This function returns the Disk Interrupt Pending status. It does not reset the interrupt condition.
- The Interrupt Status field is associated with the Logical ID as opposed to the Unit field. This field represents the current interrupt pending state of the disk controller. The Unit field is tested for validity.
- If there is a parameter error, the Interrupt Status field is undefined.
- Possible return codes = hex 0000.

Service Specific Input

Size	Offset	Description
Word	16H	Reserved

Service Specific Output

Size	Offset	Description
Byte	+10H	Interrupt Status 00H - Interrupt not pending 01H - Interrupt pending

Return Codes

The following figure lists the Disk Return Codes:

Value	Description
0000H	Operation Completed Successfully
0001H	Stage on Interrupt
0002H	Stage on Time
0005H	Not My Interrupt, Stage on Interrupt
8000H	Device Busy, Request Refused
800FH	DMA Arbitration Level Out of Range
9001H	Bad Function
9002H	Address Mark Not Found
9004H	Record Not Found
9005H	Reset Failed
9007H	Controller Parameter Activity Failed
900AH	Defective Sector
900BH	Bad Track
900DH	Invalid Sector on Format
900EH	CAM Detected During Read or Verify
9010H	Uncorrectable ECC or CRC Error
9020H	Bad Controller
9021H	Equipment Check
9040H	Bad Seek
9080H	Device Did Not Respond
90AAH	Drive Not Ready
90BBH	Undefined Error
90CCH	Write Fault
90FFH	Incomplete Sense Operation
A000H	TimeOut Occurred - No Other Error
A001H	Bad Command
A002H	Address Mark Not Found
A004H	Record Not Found
A005H	Reset Failed
A007H	Parameter Activity Failed
A00AH	Defective Sector
A00BH	Bad Track
A00DH	Invalid Sector on Format
A00EH	CAM Detected During Read or Verify
A010H	Uncorrectable ECC or CRC Error
A011H	ECC Corrected Data Error
A020H	Bad Controller
A021H	Equipment Check
A040H	Bad Seek
A080H	Device Did Not Respond
A0AAH	Drive Not Ready
A0BBH	Undefined Error
A0CCH	Write Fault
A0FFH	Incomplete Sense Operation
B001H	Controller Bad Command
B020H	Bad Controller
B021H	Equipment Check
B080H	Device Did Not Respond
B0BBH	Undefined Error
B0FFH	Sense Failed
C000H	Invalid Logical ID (ABIOS Transfer Convention only)
C001H	Invalid Function
C003H	Invalid Unit Number
C004H	Invalid Request Block Length

Figure 47. Disk Return Codes

Programming Considerations

- The Disk ABIOS interface requires the use of the DMA ABIOS interface, therefore, if the disk ABIOS is initialized and used, the DMA ABIOS must not be initialized.
- Read Device Parameters returns the number of software retries to attempt for any one operation when an error occurs that is retryable.
- In the event of an error, ABIOS resets the disk system when required.
- For the functions Read (hex 08), Write (hex 09), and Write/Verify (hex 0A), the output parameter at hex 2C represents the number of blocks transferred as determined from the hardware. This value is supplied in the event that the request ended in error before the data transfer was complete. If no error is reported this value equals the number of blocks that were requested for transfer. It is only valid when the request is completed.
- When error recovery procedures are invoked by the adapter, and are successful, disk ABIOS attempts to determine the nature of the recovery performed. It sets the Soft Error Occurred field in the request block with the recovered error code.
- Relative block addresses begin ordering with the first block assigned the value 0. For hardware devices that do not support relative block addresses, the equivalent is Cylinder 0, Head 0, and Sector 1. In the formulas below, sectors per track, sector ID, heads, and cylinders refer to physical (1-based) entities. Cylinder and head refer to ID values as actually sent to the controller (0 based). Disk ABIOS returns physical values for number of sectors per track, number of heads, number of cylinders on the Read Device Parameters function (hex 03), which should be used for relative block address calculations. ABIOS uses the following to break down the Relative Block Address (RBA):

  Sector ID = (RBA MOD Sectors Per Track) + 1

  Head = (RBA \ Sectors Per Track) MOD Heads

  Cylinder = (RBA \ Sectors Per Track) \ Heads

  The RBA may be calculated by the following:

  RBA = (Sectors Per Track * Heads * Cylinder) + (Sectors Per Track * Head) + Sector ID - 1

  The number of RBAs is:

  RBAs = Cylinders * Heads * Sectors Per Track

  This is the value returned by Read Device Parameters.

  The maximum allowable RBA is:

  Max RBA = cylinders * heads * sectors per track - 1.

- When issuing Disk ABIOS and Disk BIOS requests, the following rules should be followed:
  - Do not attempt an ABIOS call while there is an outstanding BIOS call.
  - Do not attempt a BIOS call while there is an outstanding ABIOS call.
  - The Reset/Initialize function (hex 05) must be the first ABIOS request immediately following a BIOS request
  - The Reset Disk System BIOS function (Interrupt 13H, (AH) = 00H) must be the first BIOS request immediately following an ABIOS request.
  - The Reset/Initialize function (hex 05) must be issued after ABIOS initialization has been completed.

Video

Functions

The following shows the video functions with an explanation of each function performed.

00H - Default Interrupt Handler

01H - Return Logical ID Parameters

02H - Reserved

03H - Read Device Parameters

- This function returns parameters that indicate the current video state.
- The Character Block Specifier field returns the active character generator blocks in map 2. The Character Block Select A field specifies the block used to generate alpha characters when bit 3 of the Attribute byte is a 1. The Character Block Select B field specifies the block used to generate alpha characters when attribute bit 3 of the Attribute byte is a 0. When the Character Block Select A field is equal to the Character Block Select B field, the Character Select function is disabled and bit 3 of the Attribute byte determines the foreground intensity state (1 = on, 0 = off).
- The Save/Restore Header Size, Hardware State Size, Device Block State Size, and DAC State Size fields are used in calculating the size of the Save buffer for the Save Environment function (hex 0C). Refer to the Save Environment function (hex 0C), on page 67 for more information.
- The possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+28H	Word	Reserved

Service Specific Output

OFFSET	SIZE	DESCRIPTION
+1CH	Byte	Number of scan lines on the screen 00H - 200 scan lines 01H - 350 scan lines 02H - 400 scan lines 03H - 480 scan lines 04H to 0FFH - Reserved
+1EH	Word	Video mode setting
+20H	Word	Bits 15 to 1 - Reserved Bit 0 - Type of monitor attached 0 - Color monitor 1 - Monochrome monitor
+22H	Word	Character height (bytes/character)
+24H	Word	Character block specifier Bits 15 to 12 - Reserved Bits 11 to 8 - Character Block Select A Bits 7 to 4 - Reserved Bits 3 to 0 - Character Block Select B
+2AH	Word	Size of ROM fonts. Data buffer required for the Return ROM Fonts function
+2EH	Word	Size of the save/restore buffer header in bytes
+30H	Word	Size of the save/restore hardware state in bytes
+32H	Word	Size of the save/restore device block state in bytes
+34H	Word	Size of the save/restore DAC state in bytes

04H - Set Device Parameters (Reserved)

05H - Reset/Initialize

- This function initializes the video controller to the requested mode. (See page 71).
- The Character Blocks to Load field tells which character blocks will be loaded with the default ROM character font for the requested mode and scan lines. This parameter is only required when setting an alpha mode (0, 1, 2, 3, 7).
- Scan lines are only specified when setting an alpha mode (0, 1, 2, 3, 7).
- The Character Blocks Specifier field is only specified when setting an alpha mode (0, 1, 2, 3, 7).
- The Device Control Summing flag is only required when a color monitor is attached. Summing is done automatically for monochrome monitors.
- When using a monochrome display in a color mode, the colors are displayed as shades of gray. There are 16 of 64 gray shades available in all modes except mode 13H, where all 64 gray shades are available.
- For the Character Blocks Specifier field, the Character Block Select A field specifies the block used to generate alpha characters when bit 3 of the Attribute byte is a 1. The Character Block Select B field specifies the block used to generate alpha characters when bit 3 of the Attribute byte is 0. When the Character Block Select A field is equal to the Character Block Select B field, the Character Select function is disabled and bit 3 of the Attribute byte determines the foreground intensity state (1 = on, 0 = off).

- Modes 0, 2 and 5 are identical to modes 1, 3 and 4 respectively.
- The possible return code = hex 0000.

Service Specific Input

OFFSET	SIZE	DESCRIPTION
+1AH	Word	Video device control flag Bits 15 to 3 - Reserved Bit 2 - Summing 0 - Summing disabled 1 - Summing enabled Bit 1 - Initialize digital-to-analog converter (DAC) to default 0 - Do not initialize DAC to default 1 - Initialize DAC to default Bit 0 - Regenerative buffer flag 0 - Don't clear buffer 1 - Clear buffer
+1CH	Byte	Requested number of scan lines 00H - 200 scan lines (modes 0, 1, 2, 3) 01H - 350 scan lines (modes 0, 1, 2, 3, 7) 02H - 400 scan lines (modes 0, 1, 2, 3, 7) 03H to FFH - Reserved
+1EH	Word	Video mode to set
+24H	Word	Character block specifier Bits 15 to 12 - Reserved Bits 11 to 8 - Character Block Select A Bits 7 to 4 - Reserved Bits 3 to 0 - Character Block Select B
+26H	Word	Character blocks to load with default ROM font Bit 'n' - Block 'n' flag 0 - Don't update font 1 - Update font
+28H	Word	Reserved

Service Specific Output

Offset	Size	Description
None		

06H - Enable (Reserved)

07H - Disable (Reserved)

08H - Read (Reserved)

09H - Write (Reserved)

0AH - Additional Data Transfer Function (Reserved)

0BH - Return ROM Fonts Information

- This function returns the following information about each of the ROM fonts: the pointer to the ROM font, the size of character (row and column), whether it is a total or partial font, and if a partial font, which font it relates to.
- There are 12 bytes of information per ROM font. They are stored sequentially in the specified data area.
- The following shows the format of a ROM font entry:

```
 Word - Reserved
DWord - Pointer to ROM font
 Word - Reserved
 Byte - Size of character (number of columns)
 Byte - Size of character (number of rows)
 Byte - Total/partial font indicator
        00H - Total font
        01H - Partial font
        02H to FFH - Reserved
 Byte - Related font
        If this is a partial font, this byte contains a
        number to indicate which font this font goes with.
        The font number is based on the place a particular
        font occupies in the ROM font entries.
```

- The Read Device Parameters function (hex 03) should be issued before issuing this function to find the size of the buffer required to save the ROM fonts information to be returned.
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+10H	Word	Reserved
+12H	DWord	Pointer to buffer to store ROM fonts information
+16H	Word	Reserved

Service Specific Output

Offset	Size	Description
None		

0CH - Save Environment

- This function stores the caller's requested video states in the buffer.
- The video environment consists of the following states:
  - Hardware state
  - Device block state
  - DAC state
- To calculate the size of the save buffer that is required, the Read Device Parameters function (hex 03) must be issued. It gives the individual sizes of the possible states to be saved and the size of the save/restore header. Then:

  Save Buffer Size = (A+B+C+D)

  where:

  A = Size of the Save/Restore header.
  B = Environment(bit 0) * (size of hardware state).
  C = Environment(bit 1) * (size of device block state).
  D = Environment(bit 2) * (size of DAC state).
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+10H	Word	Reserved
+12H	DWord	Pointer to environment save area
+16H	Word	Reserved
+2CH	Word	Video environment states to be saved Bits 15 to 3 - Reserved and should be set to 0 Bit 2 - DAC state Bit 1 - Device block state Bit 0 - Hardware state

Service Specific Output

Offset	Size	Description
None		

0DH - Restore Environment

- This function restores the video environment from the given buffer location. Refer to the Save Environment function (hex 0C) for more information on the contents and structure of the video environment.
- Unexpected results may occur if you restore a particular state not previously saved.
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+10H	Word	Reserved
+12H	DWord	Pointer to environment restore area
+16H	Word	Reserved
+1AH	Word	Device Control flag Bits 15 to 1 - Reserved Bit 0 - Regenerative buffer flag 0 - Don't clear buffer 1 - Clear buffer
+2CH	Word	Video environment states to be restored Bits 15 to 3 - Reserved Bit 0 - Hardware state Bit 2 - DAC state Bit 1 - Device block state Bit 0 - Hardware state

Service Specific Output

Offset	Size	Description
None		

0EH - Select Character Generator Block

- This function selects up to two character generator blocks.
- For the Character Block Specifier field, the Character Block Select A field specifies the block used to generate alpha characters when bit 3 of the Attribute byte is a 1. For the Character Blocks to Make Active field, the Character Block Select B field specifies the block used to generate alpha characters when bit 3 of the Attribute byte is a 0. When the Character Block Select A field is equal to the Character Block Select B field, the Character Select function is disabled and bit 3 of the Attribute byte determines the foreground intensity state (1 = on, 0 = off).
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+16H	Word	Reserved
+24H	Word	Character block specifier Bits 15 to 12 - Reserved Bits 11 to 8 - Character Block Select A Bits 7 to 4 - Reserved Bits 3 to 0 - Character Block Select B

Service Specific Output

Offset	Size	Description
None		

0FH - Alpha Load

- This function loads the requested character generator or part of one to the specified character blocks.
- This function does not update the hardware registers. Refer to the Enhanced Alpha Load function (hex 10) if hardware updating is required.
- When loading any of the ROM character generators (the Character Generator Type field is equal to 1, 2 or 3), the full set of characters (hex 100) is loaded. Thus, the only parameters required to invoke this function are the Character Generator Type field and the Character Block Specifier field.
- When loading a user font (the Character Generator Type field is equal to 0) all parameters are required.
- When loading a user font, if the Count of Characters field is equal to 0, no character will be loaded and the Return Code field is set to Operation Completed Successfully (hex 0000).
- When loading a user font, the sum of the Count of Characters field and the Character Offset field should not exceed the maximum number of characters in a set (hex 100). If it does, the Return Code field is set to Invalid Video Parameter (hex C005).
- Possible return codes = hex 0000 and C005.

Service Specific Input

OFFSET	SIZE	DESCRIPTION
+10H	Word	Reserved
+12H	DWord	Pointer to user font
+16H	Word	Reserved
+18H	Word	Count of characters 1 - 100H - Valid count of characters
+1DH	Byte	Character generator type 00H - User's alphanumerics font 01H - 8 x 8 alphanumerics ROM font 02H - 8 x 14 alphanumerics ROM font 03H - 8 x 16 alphanumerics ROM font 04H to FFH - Reserved
+22H	Word	Character height (bytes/character)
+24H	Word	Character block to load Bits 15 to 12 - Reserved Bits 11 to 8 - Character Block Select A Bits 7 to 4 - Reserved Bits 3 to 0 - Character Block Select B
+28H	Word	Character offset into the table

Service Specific Output

Offset	Size	Description
None		

10H - Enhanced Alpha Load

- This function loads the requested character generator or part of one to the specified character block and updates the hardware registers.
- When loading any of the ROM character generators (the Character Generator Type field is equal to 1, 2, or 3), the full set of characters (hex 100) are loaded. Thus, the only parameters required to invoke this function are the Character Generator Type field and the Character Blocks to Load field.
- When loading a user font (the Character Generator Type field is equal to 0) all parameters are required.
- When loading a user font, if Count of Characters field is equal to 0, no character is loaded and the Return Code field is set to Operation Completed Successfully (hex 0000).
- When loading a user font, the sum of the Count of Characters field and the Character Offset field should not exceed the maximum number of characters in a set (hex 100). If it does the Return Code field is set to Invalid Video Parameter (hex C005).
- Possible return codes = hex 0000 and C005.

Service Specific Input

Offset	Size	Description
+10H	Word	Reserved
+12H	DWord	Pointer to user font
+16H	Word	Reserved
+18H	Word	Count of characters 1 - 100H - Valid count of characters
+1DH	Byte	Character generator type 00H - User's alphanumerics font 01H - 8 x 8 alphanumerics ROM font 02H - 8 x 14 alphanumerics ROM font 03H - 8 x 16 alphanumerics ROM font 04H to FFH - Reserved
+22H	Word	Character height (bytes/character)
+24H	Word	Character block to load 00H to 07H - Valid character blocks to load values 08H to FFFFH - Reserved
+28H	Word	Character offset into the table

Service Specific Output

Offset	Size	Description
None		

11H - Read Palette Register

- This function reads a palette register.
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+16H	Word	Reserved
+32H	Word	Palette register read 00H to 0FH - Valid palette register to read values 10H to FFFFH - Reserved

Service Specific Output

Offset	Size	Description
+34H	Word	Palette value read.

12H - Write Palette Register

- This function writes a value to a palette register.
- Executing this function when the mode is set to mode hex 13 is not allowed. It is a hardware requirement to have these registers remain programmed as set by the Reset/Initialize function (hex 05). Changing these registers can cause unpredictable results.
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+16H	Word	Reserved
+32H	Word	Palette register to write 00H to 0FH - Valid pallette register to write values 10H to FFFFH - Reserved
+34H	Word	Palette value to load 00H to 3FH - Valid 40H to FFFFH - Reserved

Service Specific Output

Offset	Size	Description
None		

13H - Read Color Register

- This function reads the red, green, and blue values of a color register from the video Digital-to-Analog Converter (DAC).
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+16H	Word	Reserved
+2AH	Word	Color register to read 00H to FFH - Valid color register to read values 100H to FFFFH - Reserved

Service Specific Output

Offset	Size	Description
+2CH	Word	Red value read
+2EH	Word	Green value read
+30H	Word	Blue value read

14H - Write Color Register

- This function loads a DAC color register with the specified red, green and blue values.
- For the Device Control Flags field, the summing flag is disregarded when a monochrome display is attached. Summing always occurs with a monochrome display when in color modes.
- Possible return code = hex 0000.

Service Specific Input

Offset	Size	Description
+16H	Word	Reserved
+1AH	Word	Device control flags Bits 15 to 3 - Reserved Bit 2 - Summing 0 - Summing disabled 1 - Summing enabled Bit 1 to 0 - Reserved
+2AH	Word	Color register to write 00H to FFH - Valid color registers to write 100H to FFFFH - Reserved
+2CH	Word	Red value to write 00H to 3FH - Valid red value to write 40H to FFFFH - Reserved
+2EH	Word	Green value to write 00H to 3FH - Valid green value to write 40H - FFFFH - Reserved
+30H	Word	Blue value to write 00H to 3FH - Valid blue value to write 40H - FFFFH - Reserved

Service Specific Output

Offset	Size	Description
None		

15H - Read Block of Color Registers

- This function reads a block of DAC color registers into the specified save area beginning at the requested color register.
- The format of the data returned is (red value, green value, blue value), (red value, green value, blue value), ..., (red value, green value, blue value).
- The range for the red, green, or blue values is hex 00 to 3F.
- If the Count of Color Registers to Read field equals 0, no action is performed and the Return Code field is set to Operation Completed Successfully (hex 0000).
- If the First Color Register to Read field plus the Count of Color Registers to Read field is greater than the maximum number of color registers, no action is performed and the Return Code field is set to Invalid Video Parameter (hex C005).
- Possible return codes = hex 0000 and C005.

Service Specific Input

Offset	Size	Description
+10H	Word	Reserved
+12H	DWord	Pointer to read save area
+16H	Word	Reserved
+18H	Word	Count of color registers to read
+2AH	Word	First color register to read 00H to FFH - Valid first color register to read values 1000H to FFFFH - Reserved

Service Specific Output

Offset	Size	Description
None		

16H - Write Block of Color Registers

- This function loads a block of DAC color registers with the requested values beginning with the requested color register.
- The format of the data to be written is (red value, green value, blue value), (red value, green value, blue value), ..., (red value, green value, blue value).
- If the Count of Color Registers to Write field equals zero, no action is performed and the

Return Code field is set to Operation Completed Successfully (hex 0000).

- If the first Color Register to Write field plus the Count of Color Registers field is greater than the maximum number of color registers, no action is performed and the Return Code field is set to Invalid Video Parameter (hex C005).
- For the Device Control Flags field, the summing flag is disregarded when a monochrome display is attached. Summing will always occur with a monochrome display when in color modes.
- Possible return codes = hex 0000 and C005.

Service Specific Input

Offset	Size	Description
+10H	Word	Reserved
+12H	DWord	Pointer to write save area
+16H	Word	Reserved
+18H	Word	Number of color registers to write
+1AH	Word	Device control flags Bits 15 to 3 - Reserved Bit 2 - Summing 0 - Summing disable 1 - Summing enabled Bits 1, 0 - Reserved
+2AH	Word	First color register to write 00H to FFH - Valid first color register to write 100H to FFFFH - Reserved

Service Specific Output

Offset	Size	Description
None		

Return Codes

The following figure lists the Video Return Codes:

Value	Description
0000H	Operation Completed Successfully
C000H	Invalid Logical ID (ABIOS Transfer Convention only)
C001H	Invalid Function
C003H	Invalid Unit Number
C004H	Invalid Request Block Length
C005H	Invalid Video Parameter

Figure 48. Video Return Codes

The following figure shows the supported Video Modes:

Mode #	Type	Max Colors	Alpha Format	Buffer Start	Box Size	Max Pages	Display Pel Dimensions
0	A/N	16/256K	40X25	B8000	8x8	8	320X200
0	A/N	16/256K	40X25	B8000	8x14	8	320X350
0	A/N	16/256K	40X25	B8000	9x16	8	360X400
1	A/N	16/256K	40X25	B8000	8x8	8	320X200
1	A/N	16/256K	40X25	B8000	8x14	8	320X350
1	A/N	16/256K	40X25	B8000	9x16	8	360X400
2	A/N	16/256K	80X25	B8000	8x8	8	640X200
2	A/N	16/256K	80X25	B8000	8x14	8	640X350
2	A/N	16/256K	80X25	B8000	9x16	8	720X400
3	A/N	16/256K	80X25	B8000	8x8	8	640X200
3	A/N	16/256K	80X25	B8000	8x14	8	640X350
3	A/N	16/256K	80X25	B8000	9x16	8	720X400
4	APA	4/256K	40X25	B8000	8x8	1	320X200
5	APA	4/256K	40X25	B8000	8x8	1	320X200
6	APA	2/256K	80X25	B8000	8x8	1	640X200
7	A/N	MONO	80X25	B0000	9x14	8	720X350
7	A/N	MONO	80X25	B0000	9x16	8	720X400
08-0C	RESERVED						
0D	APA	16/256K	40X25	A0000	8x8	8	320X200
0E	APA	16/256K	80X25	A0000	8x8	4	640X200
0F	APA	MONO	80X25	A0000	8x14	2	640X350
10	APA	16/256K	80X25	A0000	8x14	2	640X350
11	APA	2/256K	80X30	A0000	8x16	1	640X480
12	APA	16/256K	80X30	A0000	8x16	1	640x480
13	APA	256/256K	40X25	A0000	8x8	1	320x200

Figure 49. Video Mode Table

IBM Personal System/2 Seminar Proceedings

Publication Number	Vol.	Topic
G360-2653	V5.1	IBM Personal System/2 Model 30 IBM Personal Computer DOS, Version 3.30
G360-2678	V5.2	IBM Personal System/2 Displays and Display Adapters
G360-2637	V5.3	IBM Personal System/2 Models 50, 60, 80 Micro Channel™ Architecture, Hardware Features and Design Considerations
G360-2747	V5.4	IBM Personal System/2 Models 50, 60, 80 VGA, BIOS and Programming Considerations

Notes

G360-2747

IBM Corporation
Editor, IBM Personal System/2 Seminar Proceedings
Internal Zip 3636
Post Office Box 1328
Boca Raton, Fl 33429-1328