Chapter 10

Accessing and Programming the Video Cards

This chapter explains methods of programming the most popular video cards on the PC market. Even though the video cards mentioned here differ in their capabilities, they are all based on the same basic principle. High level languages such as BASIC, Pascal or C often have their own specific keywords and commands for controlling screen display. However, many of these commands merely call BIOS or DOS functions, which are both slow and inflexible in execution.

Direct access

Direct access to the video card is the alternative. Applications from Lotus 1-2-3® to dBASE® use direct video access coding, to guarantee both speed and that element of extra control over the video display. The main disadvantage: Programming in assembly language is required, since the communication here occurs at the system level. This chapter examines the programming needed for the best known video cards on the market:

- Monochrome Display Adapter (MDA), also called a *monochrome card*
- Color Graphics Adapter (CGA), also called a *color card*
- Hercules Graphic Card (HGC)
- Enhanced Graphic Adapter (EGA)
- Video Graphics Array (VGA)

Most of the graphic cards on the market are compatible with one of the cards mentioned in this chapter, and the descriptions stated here should apply to those cards.

Video Graphics Array (VGA)

This also applies to the newest generation of video cards, the VGA card. Designed in conjunction with the IBM PS/2 system, the VGA card is now available to the general public as an add-on card. This chapter demonstrates some general features of the EGA and VGA, as well as a few programming techniques.

What's needed

Before a video card can display a character or graphic pixel on a monitor screen or *CRT* (cathode ray tube), the card must know the following:

- which character or graphic pixel to display
- The color of the character or pixel
- The location on the screen at which it should be displayed.

PC video cards include RAM which collects information about every CRT screen pixel or screen location. This RAM memory is called *video RAM* and interfaces with the PC's RAM, allowing direct access from the microprocessor.

Speed

Rapid screen changes are important in word processing programs and other PC applications. For example, if you are paging through a word processing document at high speed, a 25-line, 80-column screen requires the transmission of 2,000 characters through the video card at one time. Fast data transfer is even more important for high-resolution graphics. For example, the 200x640-pixel IBM Color Graphics Adapter transmits 128,000 pixels of graphic information at a time.

Display modes

Each type of video card can have more than one display mode. Text and graphics display may be very different from one another. The monitor cannot distinguish between the two modes; it just processes the graphic information sent by the video card (or *video controller*). For the programmer and the video card, the modes require completely different programming techniques.

Graphic mode and text mode

Graphic mode stores the color of a screen pixel in one or more bits, then transmits the contents of video RAM more or less directly to the screen. Text mode uses a different method. The ASCII code of a character is stored in video RAM for each screen location. When the video controller displays the screen, it obtains the character pattern of the ASCII code from the ROM chip on the video card, then converts the code into a character matrix of pixels. This pattern then passes to the monitor and appears on the screen.

PC text mode uses the 256-character extended character set (see Appendix I). Since these characters are numbered sequentially from 0 to 255, one byte is enough for each screen position to display the character at the proper position.

Attribute bytes

Every screen position has an *attribute byte* which indicates the color or display attribute of the character (underlined, blinking, inverse video, etc.). This means that two bytes are needed for each position on the screen. Therefore, a total of 4000 bytes are required for a 25-line, 80-column screen. This appears to be a lot of memory at first glance, but is fairly small when compared to the memory requirements for bit-mapped graphic screen. In graphic mode, each dot is represented by one or more bits. A resolution of 640x200 pixels requires 128,000 bits (16K).

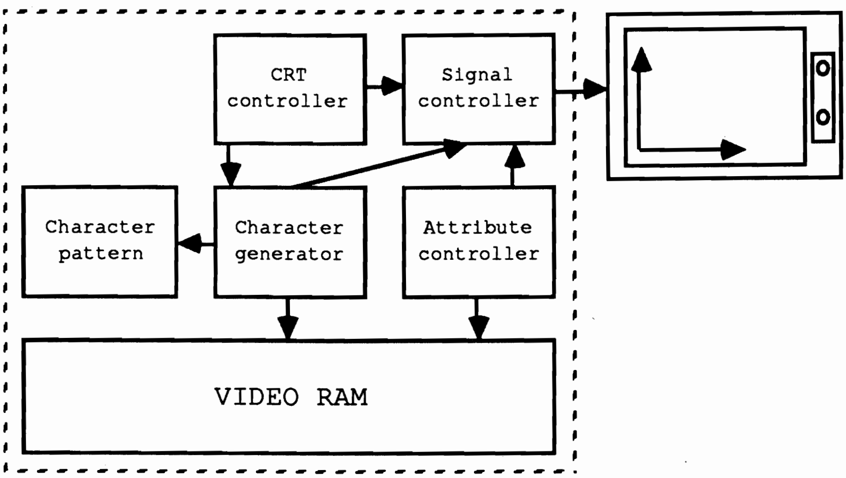
Another advantage of text mode is the simplicity in exchanging one character for another on the screen. The bit-map mode has its own advantages. Besides graphic displays, text can be displayed as individual dots whose pattern is derived from a character table in RAM installed by the user. This means that the user can design his own fonts (character sets).

10.1 Anatomy of a Video Card

The figure below shows the individual hardware components of a video card. The starting point for creating the picture is always the video RAM. This video RAM contains information about the characters to be displayed, and their display attributes (color, style, etc.).

Getting to the screen

The character generator first accesses video RAM, reading the characters one by one, and uses a character pattern table to construct the bit-map that will later form the character on the screen. The attribute controller also gets information about the display attributes (color, underlining, reverse, etc.) of the character from the video RAM. Both modules prepare this information and send it to the signal controller, which converts it to appropriate signals to be sent to the monitor. The signal controller itself is controlled by the CRT controller, which is the central point of video card operations. Besides the monitor and the video RAM, this CRT controller is one of the most important components of a video system. We will examine all these components in greater detail.



Block diagram of a video card

The monitor

The monitor is the device on which the video data is displayed. Unlike the video card, the monitor is a "dumb" device. This means it has no memory and cannot be programmed. All monitors used with PCs are *raster-scan devices*, in which the picture is made up of many small dots arranged in a rectangular pattern or raster.

When forming the picture, the electron beam of the picture tube touches each individual dot and illuminates it if it is supposed to be visible on the screen. This is done by switching on the electron beam as it passes over this dot, causing a phosphor particle on the picture tube to light up.

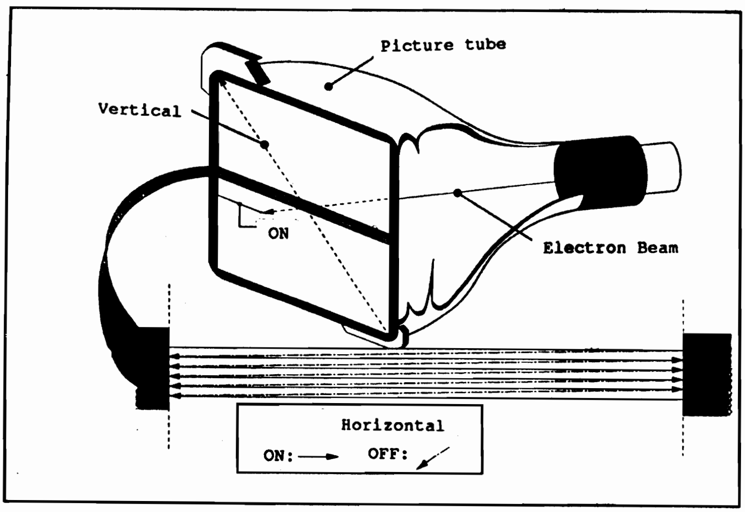
Color monitors

While monochrome monitors need only one electron beam to create a picture, color monitors use three beams which scan the screen simultaneously. Here a screen pixel consists of three phosphor particles in the basic colors of light: red, green, and blue. Each color has a matching electron beam. Any color in the spectrum can be created by combining these three colors and varying their intensities.

But since an ionized phosphor particle emits light for only a very brief period of time, the entire screen must be scanned many times per second to create the illusion of a stationary picture. PC monitors perform this task between 50 and 70 times per second. This repeated re-scanning is called the *refresh rate*. One rule of thumb for this rate: The faster the refresh rate, the better quality the picture.

Each new screen image begins in the upper left corner of the screen. From there the electron beam moves to the right along the first raster line. When it reaches the end of this line, the electron beam moves back to the start of the next line down, similar to pressing the <Return> key on a typewriter. The electron beam then scans the second raster line, at the end of which it moves to the start of the next raster line, and so on. Once it reaches the bottom of the screen, the electron beam returns to the upper left corner of the screen and the process starts over again. The illustration below shows the path of the electron beam.

Remember that the movement of the electron beam is controlled by the video card, not by the monitor itself.



Electron beam scan movement

The resolution of the monitor naturally controls the number of raster lines and columns which the electron beam scans when creating a display. Thus, a monitor which has only 200 raster lines of 640 raster columns each clearly cannot handle the high resolutions of an EGA card at 640x350 pixels. The four monitor types used with a PC generally have the following resolutions:

Resolutions of different monitors		
Monitor	Vertical	Horizontal
Monochrome	350	720
Color	200	640
EGA	350	640
Multisync	varies, up to 600	varies, up to 800

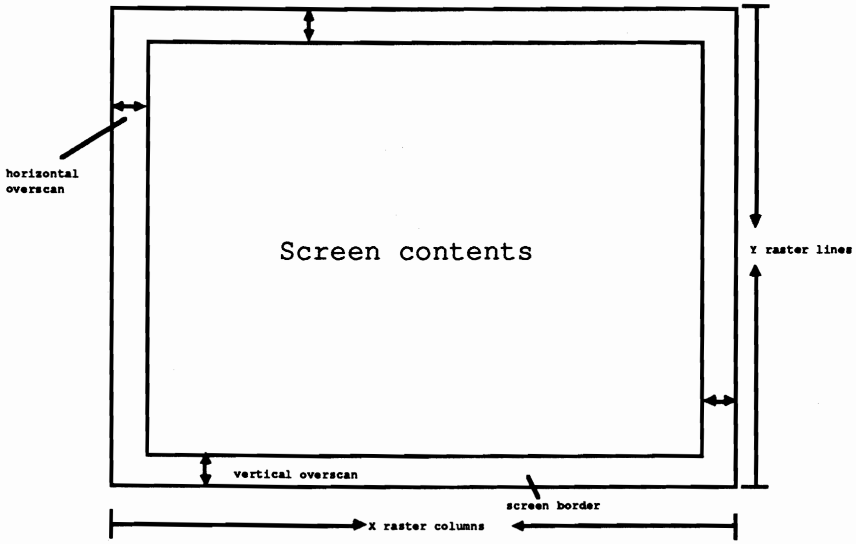
The CRT controller

The CRT Controller or CRTC is the heart of a video card. It controls the operation of the video card and generates the signals the monitor needs to create the picture. Its tasks also include controlling light pens, generating the cursor and controlling the video RAM.

To inform the monitor of the next raster line, the CRTC sends a display enable signal at the start of each line, which activates the electron beam. While the beam moves from left to right over each raster column of the line, the CRTC controls the individual signals for the electron beam(s) so that the pixels appear on the screen as desired. At the end of the line, the CRTC disables the display enable signal so that the electron beam's return to the next raster line doesn't make a visible line on the screen. The electron beam is directed to the left edge of the following raster line by the output of a horizontal synchronization signal. The display enable signal is again enabled at the start of the next raster line, and the generation of the next line begins.

Overscan

Since the time that the electron beam needs to return to the start of the next line is less than the time the CRTC needs to get and prepare new information from the video RAM, there is a short pause. But the electron beam cannot be stopped, so we get something called *overscan*, which is visible as the left and right borders of the actual screen contents. Although this is an undesirable side effect in one sense, it is useful because it prevents the edges of the screen contents from being hidden by the edge of the monitor. If the electron beam is enabled while it is traveling over this border, a color screen border can be created.



Rasters and overscan on a screen

Once the electron beam reaches the end of the last raster line, the display enable signal is disabled, and a vertical synchronization signal is sent. The electron beam returns to the upper left corner of the screen. Again the display enable signal is re-enabled and scanning again begins.

Pause and overscan

As with the horizontal electron beam return, a pause results which is displayed in the form of overscan, creating a vertical screen border.

Signal timing

The timing of individual signals varies from video mode to video mode. For this reason, the CRTC has a number of registers which describe the signal outputs and their timing. The structure of these registers and how they are programmed will be discussed in the remainder of this section. Many of these registers come from the registers of the 6845 video controller from Motorola. This controller is used in the MDA, CGA, and Hercules graphics cards. The EGA and VGA cards use a special VLSI (very large scale integration) chip as a CRTC, and its registers are somewhat more complicated. The techniques described here are intended as general descriptions for all video cards.

Registers of the 6845 video controller from Motorola		
Reg.	Meaning	Access
00H	Total horizontal character	Write
01H	Display horizontal character	Write
02H	Horizontal synchronization signal after ...char	Write
03H	Duration of horizontal synchronization signal in char.	Write
04H	Total vertical character	Write
05H	Adjust vertical character	Write
06H	Display vertical character	Write
07H	Vertical synchronization signal after ...char	Write
08H	Interlace mode	Write
09H	Number of scan lines per screen line	Write
0AH	Starting line of screen cursor	Write
0BH	Ending line of screen cursor	Write

These registers, like all of the other registers on the video card, are accessed via I/O ports with the assembly language instructions IN and OUT. The registers of the CRTC are accessed through a special address register, rather than directly from the address space of the processor. The number of the desired CRTC register is written to the port corresponding to this address register. Then the contents of this register can be read into a special data register with the IN assembly language instruction. If a value is to be written to the addressed register, it must be transferred to the data register with the OUT instruction. Then the CRTC automatically places it in the desired register. These two registers are actually found at successive port addresses, but these addresses vary from video card to video card.

We will include tables throughout the chapter to describe the contents of individual CRTC registers under the various video modes. Here's an example which shows how the contents of these registers are calculated and how the individual registers are related to each other. If you try some of these calculations with your calculator or PC, you will notice that some of them do not work out evenly. But since the registers of the CRTC hold only integer values, they will be rounded up or down.

The basis for the various calculations are the bandwidth and the horizontal and vertical scan rates of a monitor.

Bandwidth and scan rates of different video cards				
Video system rate	Resolution	Bandwidth	Vert. scan rate	Horiz. scan
MDA		720 x 350	16.257 MHz 50 Hz *	18.43 KHz*
CGA		640 x 200	14.318 MHz 60 Hz	15.75 KHz
HGC		640 x 200	14.318 MHz 50 Hz	18.43 KHz
EGA		640 x 350 640 x 200 720 x 350	16.257 MHz 60 Hz 14.318 MHz 60 Hz 16.257 MHz 50 Hz	21.85 KHz 15.75 KHz 18.43 KHz
(*MHz=Megahertz, KHz=Kilohertz, Hz=Hertz				

The bandwidths in the figure above specify the number of points which the electron beam scans per second, and is therefore also called the point or dot rate. The vertical scan rate specifies the number of screen refreshes per second, while the horizontal scan rate refers to the number of raster lines which the electron beam scans per second.

Starting with these values, let's practice calculating the individual CRTC register values for the 80x25 character text mode on a CGA card.

Dividing the bandwidth by the horizontal scan rate we get the number of pixels (screen dots) per raster line.

```
     Bandwidth                  14.318 MHz
+    Horizontal scan rate       15.570 KHz
---------------------------------------------
     Pixels per line          919
```

Since the CRTC registers generally refer to the number of characters rather than pixels, this value must be converted to the number of characters per line. This is done by dividing the number of pixels per line by the width of the character matrix. On the CGA card this is eight pixels.

```
     Pixels per line          919
+    Pixels per character       8
--------------------------------------------
     Characters per line      114
```

This value, decremented by one, is placed in the first register of the CRTC and specifies the total number of characters per line. In the second register we load the number of characters that will actually be displayed per line. The 80x25 character text mode usually offers 80 characters.

The difference between the total and the number of characters actually displayed per line is the number of characters which can be displayed between the horizontal return and the overscan. The difference in this case is 34 characters.

The duration of the horizontal beam return must be entered in the fourth register of the CRTC. This register stores the number of characters which could be displayed during this time, rather than the actual time duration. The monitor specifications define this instead of the video card itself. As a rule this number is between 5% and 15% of the total number of characters per line. A color monitor uses exactly ten characters.

This leaves 24 characters for the overscan (the horizontal screen border). The third CRTC register specifies how these characters are divided between the left and right screen borders. This register specifies the number of character positions which will be scanned before the horizontal beam return occurs. The BIOS specifies the value 90 here, or after ten characters have been displayed for the screen borders. The remaining 14 characters are placed at the start of the next line and form the left screen border.

The calculations for the vertical data, the number of vertical lines, the position of the vertical synchronization signal, etc., follow a similar scheme. The first calculation is the number of raster lines per screen. This results from the division of the number of lines displayed per second by the number of screen refreshes per second:

```
        Pixels per line         919
   +    Pixels per character      8
   ---------------------------------------------
        Characters per line     114

        Horizontal scan rate     15.750 KHz
   +    Screen refreshes         60       Hz
   ---------------------------------------------
        Raster lines            262
```

Since the characters in CGA text mode are eight pixels high by eight pixels wide, we again divide by eight to get the number of text lines per screen:

```
        Raster lines            262
   +    Pixels per character      8
   ---------------------------------------------
        Lines per screen         32
```

This result must be decremented by one and then loaded into the fifth register of the CRTC. The number of displayed lines is loaded into the seventh register. Since seven fewer lines are displayed than are actually available, these extra lines are used for the vertical beam return and overscan, whereby the vertical beam return begins after the 28th line.

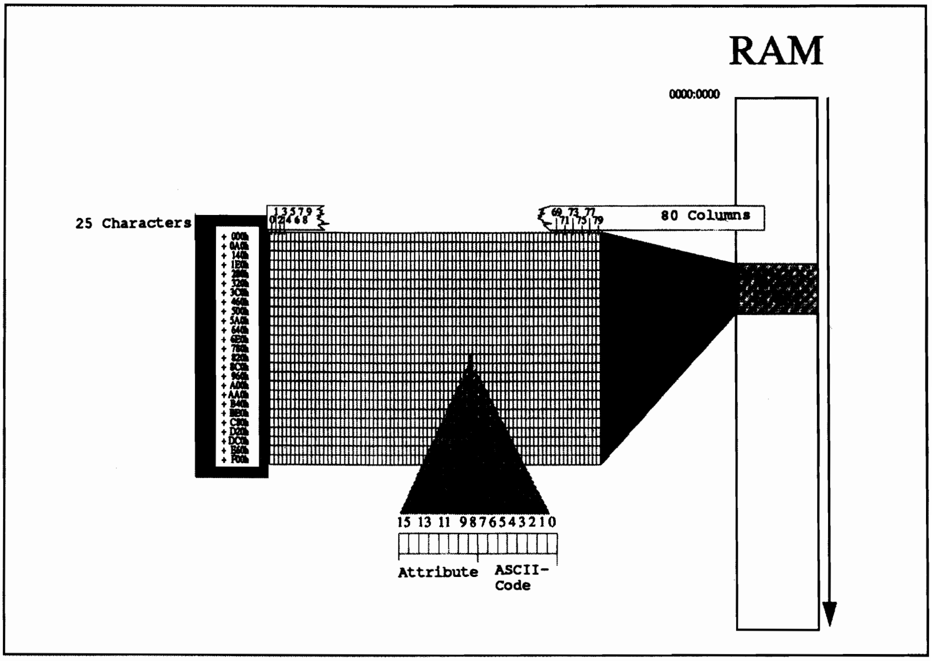
The character height must be decremented by one and loaded into CRTC register nine. The decrement results is 7 in this example. This value also determines the range for the values loaded into register ten and eleven. They specify the first and last raster lines of the screen cursor. The cursor position is determined by the contents of registers 14 and 15. They refer to the distance of the character from the upper left corner of the screen, instead of line and column. This value is calculated by multiplying the cursor line by the number of columns per line and then adding the cursor column. The high byte of the result must be loaded into register 14 and the low byte in register 15.

The video RAM area

The contents of registers 12 and 13 determine the area of video RAM displayed on the screen. To understand these registers, we first need to know something about the way video RAM is organized.

The third component of the video system determines what will eventually be displayed on the screen. In text mode, the video RAM contains the ASCII codes of the characters to be displayed and their attributes. While the organization of video RAM in this mode is identical for all of the video cards discussed here, the organization for graphic mode varies from card to card. The description of each card discusses the way video RAM organizes graphic modes (more on this later).

As the illustration below shows, each screen position occupies two bytes in video RAM. The ASCII code of the character to be displayed is placed in the first of these two bytes, the one with the even address. By using eight bits per character code, a maximum of 256 different characters can be displayed.



Normal text mode structure in video RAM

After the ASCII code, and always at an odd offset address, follows the attribute byte, which defines the appearance of the character on the screen. The attribute controller divides it into two nibbles, whereby the upper nibble (bits four to seven) describes the character background, and the lower nibble (bits zero to three) describes the character foreground. This results in two values between zero and fifteen which are interpreted depending on the type of monitor attached. With a color monitor (and a CGA or EGA card) both values select one of 16 possible colors. Each character on the screen can thus have its own foreground and background colors.

A monochrome monitor cannot display colors, regardless of the adapter. Here the attribute controls whether the character is displayed at high or low intensity, inverse, or underlined.

Character organization in video RAM

To access video RAM, you must know how the individual characters are organized within this memory. This organization is similar to character display on the screen.

The first character on the screen (the character in the upper left corner) is also the first character in video RAM, located at offset position 0000H. The next character to the right is located at offset position 0002H. All 80 characters of the first screen line follow in this manner. Since each screen character takes two bytes of memory, each line occupies 160 bytes of RAM. The first character of the second screen line follows the last character of the first line, and so on.

Finding character locations in video RAM

You can easily find the starting address of a line within video RAM by multiplying the line number (starting with zero) by 160. To get from the beginning of the line to a character within the line, the distance of the character from the start of the line must be added to this value. Since each character takes two bytes, this is done simply by multiplying the column number (also starting at zero) by two. Adding both products together yields the offset position of the character in the video RAM. These calculations can be combined into a single formula:

```
Offset_position(row, column) = row * 160 + column * 2
```

Note: Since only 40 characters per line are displayed in 40-column video modes, the factor 80 must replace the original 160.

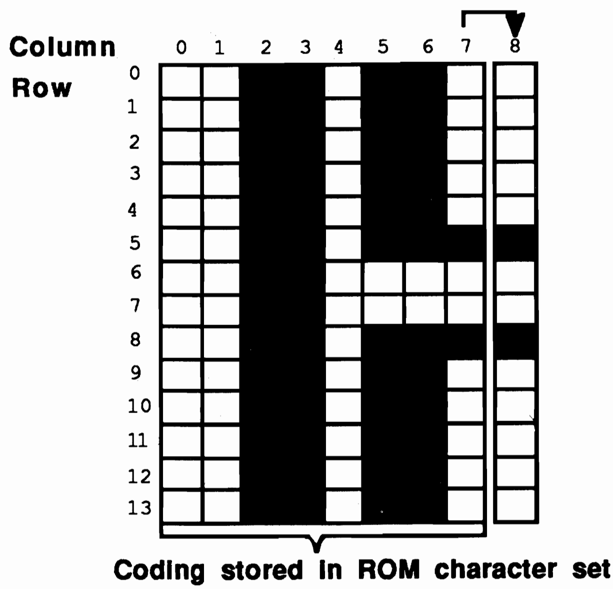
The RAM memory of the video card is integrated into the normal RAM of the PC system, so you can use normal memory access commands to access video RAM. You must know the segment address of video RAM, which is used together with the formula above to find the offset position. Section 10.7 shows how this can be done easily in assembly language, BASIC, Pascal, and C.

Now that we have discussed the most important similarities between the four video cards, the following four sections describe the capabilities of these cards. In addition, these sections explain how these capabilities can be used for optimal screen output.

10.2 The IBM Monochrome Card

The IBM Monochrome Display Adapter, or MDA, is probably the oldest of the video cards. This card is based on the Motorola 6845 video controller, which is an intelligent peripheral chip. The 6845 controller constructs a display by generating the proper signals for the monitor from video RAM.

This card is excellent for text display. This is achieved with a 9x14 character matrix, which permits high-resolution character display. The format of this matrix is unusual since a character generator containing the bit pattern of each character can only produce characters 8 pixels wide. Characters from the IBM character set may not connect with each other (e.g., using box characters to draw a box). A circuit on the graphics card sidesteps this disadvantage by copying the eighth pixel of the line into the ninth pixel for any characters whose ASCII codes are between B0H and DFH. This allows the horizontal box drawing characters to connect.



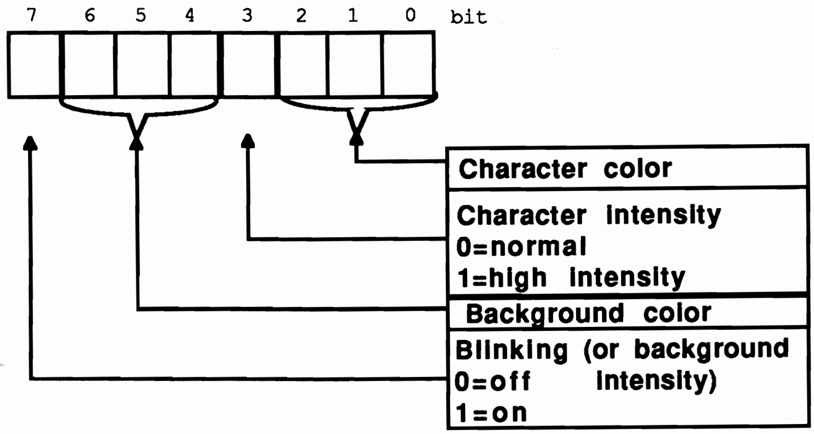
Monochrome display adapter—9x14 character matrix

The character generator requires one byte for each screen line: one bit per pixel, eight bits per line. Each character requires 14 bytes. The complete character set has a memory requirement of almost 4K, stored in a ROM chip on the card. For some reason the card has an 8K ROM, leaving the second bank of 4K unused.

Video RAM on the MDA

The video RAM of the card starts at address B000:0000 and extends over 4K (4,096 bytes). Since the screen display only has space for 2,000 characters and requires only 4,000 bytes of memory for those characters, the unused 96 bytes at the end of video RAM are available for other applications.

The following figure shows the meanings of the different values representing the attribute byte:



Attribute byte values—IBM monochrome display adapter

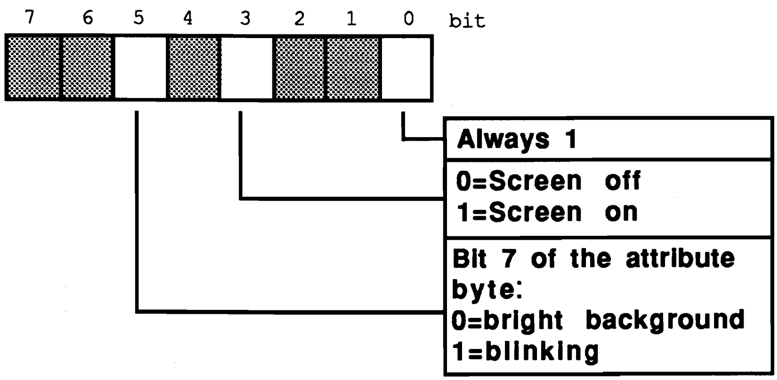
Any combination of bits can be loaded into this byte. However, the MDA only accepts the following combinations:

7	6	5	4	3	2	1	0	
?	0	0	0	?	0	0	0	No character (black on black)
?	0	0	0	?	0	0	1	underlined character (white on black)
?	0	0	0	?	1	1	1	White character on black
?	1	1	1	?	0	0	0	Black character on white (inverse)
?	1	1	1	?	1	1	1	No character (white on white)

Byte combinations—IBM monochrome display adapter

Besides these bit combinations, bits 3 and 7 of the attribute byte can be set or unset. Bit 3 defines the intensity of the foreground display. When this bit is set, the characters appear in higher intensity. Bit 7's purpose varies with the contents of the control registers (more on this later). For now, all you need to know is that bit 7 can either enable blinking characters, or enable an intensity matching the background color.

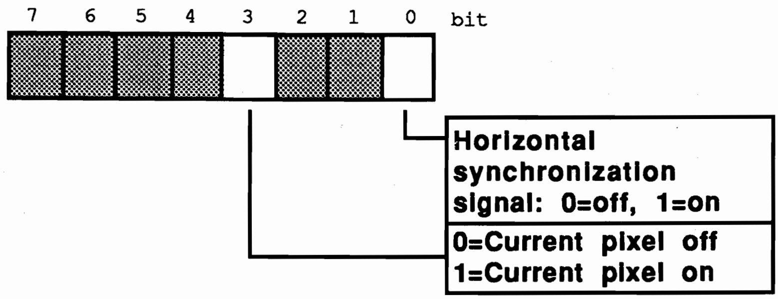
Monochrome cards have two more registers available: the control register and the status register.



Control register

MDA control register

The control register located at port 3B8H controls the monochrome display adapter's different functions. As the figure below shows, only bits 0, 3 and 5 are of importance. Bit 0 controls the resolution on the card. Although the card only supports one resolution (80x25 characters), this bit must be set to 1 during system initialization. Otherwise the computer goes into an infinite wait loop. Bit 3 controls the creation of a visible display on the monitor. If bit 3 is set to 0, the screen is black and the blinking cursor disappears. If bit 3 is set to 1, the display returns to the screen. Bit 5 has a similar function: If bit 7 in the attribute byte of the character is set to 1, it enables blinking characters. If bit 7 contains the value 0, the character appears, unblinking, in front of a light background color. This means that bit 7 of the attribute byte acts as an intensity bit for the background. This register can only be written. This makes it impossible for a program to determine whether the display is turned on or off. The normal value for this register is 29H, meaning that all three relevant bits default to 1.



Status registers (3BAH)

MDA status register

Only bits 0 and 3 are used in the status register; all the other bits must contain the value 1. Unlike the control register, programs can read this register, but register contents cannot be changed by program code.

Horizontal synchronization

Bit 0 indicates if a horizontal synchronization signal is being sent to the display screen. The video card sends this signal after creating a screen line (not to be confused with a text line, which consists of 14 screen lines) on the screen. This signal informs the electron gun, which "draws" the picture on the screen, that it should return to the left border of the current screen line. In this case the bit has the value 1. Bit 3 contains the value of the pixel where the electron beam is currently located. A 1 signals that the pixel is visible on the screen and 0 means that the screen remains black at this location.

MDA internal registers

Besides the two registers directly connected to the hardware of the monochrome display adapter, the 6845 video processor contains a series of internal registers. These 18 registers are open to user access through the 6845 index register and data register. The index register is connected to port address 3B4H, the data register at port address 3B5H. You can only write to the 6845 registers—you cannot read data from them.

When you enter a value into one of the 18 registers, the number of the register (0-17) passes first into the index register. Then the value which is transmitted to the register passes into the data register. The 6845 then transmits the indicated value to the proper register. Most of these 18 registers should not be modified, since they contain important data about the screen structure (e.g., synchronization signals) and incorrect values in these registers can damage the monitor. The following table shows the meanings of the individual registers and the values which ensure a correct display.

Registers of the CRTC register in 80x25 text mode on the Monochrome Display Adapter (MDA)		
Reg.	Meaning	Content
00H	Total horizontal character	97
01H	Display horizontal character	80
02H	Horizontal synchronization signal after ...char	82
03H	Duration of horizontal synchronization signal in char.	15
04H	Total vertical character	25
05H	Adjust vertical character	6
06H	Display vertical character	25
07H	Vertical synchronization signal after ...char	25
08H	Interlace mode	2
09H	Number of scan lines per screen line	13
0AH	Starting line of blinking screen cursor	11
0BH	Ending line of blinking screen cursor	12
0CH	Starting address of displayed screen page (low byte)	0
0DH	Starting address of displayed screen page (high byte)	0
0EH	Character address of blinking screen cursor (high byte)	0
0FH	Character address of blinking screen cursor (low byte)	0
10H	Light pen position (high byte)	*
11H	Light pen position (low byte)	*
	*not available on MDA	

The following program makes full use of the monochrome display adapter's capabilities. It was written in assembly language. The individual routines are fully documented and require no additional explanation. The demonstration program built into the listing shows practical application of the individual routines.

Assembler listing: VMONO.ASM

```
;**********************************************************************;
;*                              V M O N O                             *;
;*--------------------------------------------------------------------*;
;*    Task           : makes some elementary functions available for *;
;*                     access to the monochrome display screen        *;
;*--------------------------------------------------------------------*;
;*    Info           : all functions subdivide the screen             *;
;*                     into columns 0 to 79 and lines 0 to 24         *;
;*--------------------------------------------------------------------*;
;*    Author         : MICHAEL TISCHER                                *;
;*    Developed on   : 8/11/87                                        *;
;*    Last Update    : 6/14/89                                        *;
;*--------------------------------------------------------------------*;
;*    assembly       : MASM VMONO;                                    *;
;*                     LINK VMONO;                                    *;
;*--------------------------------------------------------------------*;
;*    Call           : VMONO                                          *;
;**********************************************************************;

;== Constants ==========================================================

CONTROL_REG   = 03B8h             ;Control register port address
ADDRESS_6845  = 04B4h             ;6845 address register
DATA_6845     = 03B5h             ;6845 data register
VIO_SEG       = 0B000h            ;Segment address of video RAM
CUR_START     = 10                ;Register # CRTC: Starting cursor line
CUR_END       = 11                ;Register # CRTC: Ending cursor line
CURPOS_HI     = 14                ;Register # CRTC: Cursor pos. hi byte
CURPOS_LO     = 15                ;Register # CRTC: Cursor pos. lo byte

DELAY         = 20000             ;Counter for delay loop
```

```
;== Stack ============================================================

stack     segment para stack       ;Definition of stack segment

          dw 256 dup (?)           ;256-word stack

stack     ends                     ;End of stack segment

;== Data =============================================================

data    segment para 'DATA'      ;Define data segment

;== the Data for the Demo-Program ====================================

str1    db "a" ,0
str2    db " >PC SYSTEM PROGRAMMING< ",0
str3    db "   window 1   ",0
str4    db "   window 2   ",0
str5    db "                 the program is stopped by "
        db " pressing a Key....                  ",0

initm   db 13,10,"VMONO (c) 1987 by Michael Tischer",13,10,13,10
        db "This demonstration program only runs with "
        db " a monochrome",13,10,"display card. If your PC "
        db "has another type of display card,",13,10
        db "please enter <s> to stop the "
        db " program.",13,10,"Otherwise press any "
        db "key to start ",13,10
        db "the program ...",13,10,"$"

;== Data =============================================================

linen     dw  0*160,1*160,2*160 ;Start addresses of the lines as
          dw  3*160,4*160,5*160 ;offset addresses in the video RAM
          dw  6*160,7*160,8*160
          dw  9*160,10*160,11*160,12*160,13*160,14*160,15*160,16*160
          dw 17*160,18*160,19*160,20*160,21*160,22*160,23*160,24*160

data      ends                    ;End of data segment

;== Code ================================================

code      segment para 'CODE'     ;Definition of the CODE segment

          assume cs:code, ds:data, es:data, ss:stack

;== this is the Demo-Program =====================================

demo      proc far

          mov ax,data             ;Get segment address of data segment
          mov ds,ax               ;and load into DS
          mov es,ax               ;as well as ES

          ;-- Display initial msg./wait for input ----------------

          mov  ah,9               ;String output function
          mov  dx,offset initm    ;Address of initial message
          int  21h                ;Call DOS interrupt 21H

          xor  ah,ah              ;Get function number for key
          int  16h                ;Call BIOS keyboard interrupt
          cmp  al,"s"             ;was <s> entered?
          je   ende               ;YES --> end program
          cmp  al,"S"             ;was <S> entered?
          jne  startdemo          ;NO --> start demo

ende:     mov  ax,4c00h           ;Function number for program end
          int  21h                ;Call DOS interrupt 21H
```

```
startdemo label near
          mov  cx,0d00h           ;Enable full cursor
          call cdef
          call cls                ;Clear screen

          ;-- Fill screen with ASCII characters -------------

          xor  di,di              ;Start in upper left corner
          mov  si,offset str1     ;Offset address of string1
          mov  cx,2000            ;2,000 characters fit on the screen
          mov  al,07h             ;white letters on black background
demo1:    call print              ;Display string
          inc  str1               ;Increment character in test string
          jne  demo2              ;NUL code suppressed
          inc  str1
demo2:    loop demo1              ;Repeat output

          ;-- Create window 1 and window 2 ----------

          mov  bx,0508h           ;Upper left corner of window 1
          mov  dx,1316h           ;Lower right corner of window 1
          mov  ah,07h             ;White letters, black background
          call clear              ;Clear window 1
          mov  bx,3C02h           ;Upper left corner of window 2
          mov  dx,4A10h           ;Lower right corner window 2
          call clear              ;Clear window 2
          mov  bx,0508h           ;Upper left corner of window 1
          call calo               ;Convert to offset address
          mov  si,offset str3     ;Offset address string 3
          mov  ah,70h             ;Black characters, white background
          call print              ;Display string 3
          mov  bx,3C02h           ;Upper left corner of window 2
          call calo               ;Convert to offset address
          mov  si,offset str4     ;Offset address string 4
          call print              ;Display string 4
          xor  di,di              ;Upper left display corner
          mov  si,offset str5     ;Offset address string 5
          call print              ;Display string 5

          ;-- Display program logo ------------------------------

          mov  bx,1E0Ch           ;Column 30, line 12
          call calo               ;Convert offset address
          mov  si,offset str2     ;Offset address string 2
          mov  ah,0F0h            ;Inverse blinking
          call print              ;Display string 2

          ;-- Fill window with arrows ----------------------------

          xor  ch,ch              ;Hi-byte of the counter to 0
arrow:    mov  bl,1               ;Asterisk
arrow0:   push bx                 ;Push BX on the stack
          mov  di,offset str3     ;Draw arrow line in string 3
          mov  cl,15              ;Total of 15 characters in a line
          sub  cl,bl              ;Calculate number of spaces
          shr  cl,1               ;Divide by 2 (for left half)
          or   cl,cl              ;No blanks ?
          je   arrow1             ;YES --> ARROW1
          mov  al," "
          rep  stosb              ;Draw blanks in string 3
arrow1:   mov  cl,bl              ;Number of asterisks in counter
          mov  al,"*"
          rep  stosb              ;Draw stars in string 3
          mov  cl,15              ;Total of 15 characters in a line
          sub  cl,bl              ;Calculate number of blanks
          shr  cl,1               ;Divide by 2 (for right half)
          or   cl,cl              ;No blanks?
          je   arrow2             ;YES --> ARROW2
          mov  al," "
```

```
          rep  stosb              ;Draw blanks in string 3
arrow2:   mov  bx,0509h           ;below the first line of window 1
          call calo               ;Convert to offset address
          mov  si,offset str3     ;Offset address string 3
          mov  ah,07h             ;White characters, black background
          call print              ;Display string 3
          mov  bx,3C10h           ;into the lowest line of window 2
          call calo               ;Convert offset address
          call print              ;Display string 3

          ;-- Brief pause ------------------------------------

          mov  cx,DELAY           ;Loop counter
waitlp:   loop waitlp             ;Count loop to 0

          ;-- Scroll window 1 line down -------------------------

          mov  bx,0509h           ;Upper left corner of window 1
          mov  dx,1316h           ;Lower right corner window 1
          mov  cl,1               ;Scroll down
          call scrolldn           ;one line

          ;-- Scroll window 2 one line up -----------------------

          mov  bx,3C03h           ;Upper left corner window 2
          mov  dx,4A10h           ;Lower right corner window 2
          call scrollup           ;Scroll up

          ;-- Was a key pressed? (end program) -------------------

          mov  ah,1               ;Function number for testing key
          int  16h                ;Call BIOS keyboard interrupt
          jne  end_it             ;Keypress -> goto end of program

          ;-- NO, display next arrow ----------------------------

          pop  bx                 ;Pop BX from stack again
          add  bl,2               ;2 more stars in next line
          cmp  bl,17              ;Reached 17 ?
          jne  arrow0             ;NO --> next arrow
          jmp  arrow              ;No key --> next arrow

          ;-- Get ready to end program

end_it:   xor  ah,ah              ;Get function number for key
          int  16h                ;Call BIOS-keyboard-interrupt
          mov  cx,0D0Ch           ;Restore normal cursor
          call cdef
          call cls                ;Clear screen
          jmp  ende               ;Go to end of program

demo      endp

;== Functions ================================================

;-- SOFF: switches the display off  ----------------------
;-- Input    : none
;-- Output   : none
;-- register : AX and DX are changed

SOFF      proc near

          mov  dx,CONTROL_REG     ;Address of display control register
          in   al,dx              ;read its content
          and  al,11110111b       ;bit 3 = 0: display off
          out  dx,al              ;set new value (display off)

          ret                     ;back to caller

SOFF      endp
```

```
;-- SON: switches the display on ------------------------
;-- Input    : none
;-- Output   : none
;-- register : AX and DX are changed

SON       proc near

          mov  dx,CONTROL_REG      ;Address of display control register
          in   al,dx               ;Read its content
          or   al,8                ;Bit 3 = 1: display on
          out  dx,al               ;Set new value (display on)
          ret                      ;Back to caller

SON       endp

;-- CDEF: sets the start and end line of the cursor ------------
;-- Input    : CL = Start line
;--            CH = End line
;-- Output   : none
;-- register : AX and DX are changed
cdef      proc near

          mov  al,CUR_START        ;Register 10: start line
          mov  ah,cl               ;Start line to AH
          call setvk               ;Transmit to video controller
          mov  al,CUR_END          ;Register 11: end line
          mov  ah,ch               ;End line to AH
          jmp  short setvk         ;Transmit to video controller

cdef      endp

;-- SETBLINK: sets the blinking display cursor -------------------
;-- Input    : DI = offset address of the cursor
;-- Output   : none
;-- register : BX, AX and DX are changed

setblink  proc near

          mov  bx,di               ;Transmit offset to BX
          mov  al,CURPOS_HI        ;Register 15:Hi-byte of cursor offset
          mov  ah,bh               ;HI-byte of the offset
          call setvk               ;Transmit to video controller
          mov  al,CURPOS_LO        ;Register 15:Lo-byte of cursor offset
          mov  ah,bl               ;Lo-byte of the offset

          ;-- SETVK is called automatically -----------------------

setblink  endp

;-SETVK: sets a byte in one of the registers of the video controller --
;-- Input    :  AL = number of the register
;--             AH = new content of the register
;-- Output   :  none
;-- register : DX and AL are changed

setvk     proc near

          mov  dx,ADDRESS_6845     ;Address of the index register
          out  dx,al               ;Send number of the register
          jmp  short $+2           ;Small I/O pause
          inc  dx                  ;Address of the index register
          mov  al,ah               ;Content to AL
          out  dx,al               ;Set new content
          ret                      ;Back to caller

setvk     endp

;-- GETVK: reads a byte from one register of the video controllers -
;-- Input     : AL = number of the register
```

```
;-- Output   : AL = content of the register
;-- register : DX and AL are changed

getvk     proc near

          mov  dx,ADDRESS_6845     ;Address of the index register
          out  dx,al               ;Send number of the register
          jmp short $+2
          inc  dx                  ;Address of the index register
          in   al,dx               ;Read content to AL
          ret                      ;Back to caller

getvk     endp

;-- SCROLLUP: scrolls a window up by N lines ----------------
;-- Input  :   BL = line upper left
;--            BH = column upper left
;--            DL = line lower right
;--            DH = column lower right
;--            CL = number of lines to scroll
;-- Output  : none
;-- register : only FLAGS are changed
;-- Info     : the display lines released are erased

scrollup  proc near

          cld                      ;Increment on string instructions

          push ax                  ;Push all changed registers on the
          push bx                  ;stack
          push di                  ;In this case the sequence
          push si                  ;must be observed!

          push bx                  ;These three registers are restored
          push cx                  ;from the stack before ending
          push dx
          sub  dl,bl               ;Calculate the number of lines
          inc  dl
          sub  dl,cl               ;Deduct number of lines scrolled
          sub  dh,bh               ;Calculate number of columns
          inc  dh
          call calo                ;Convert upper left in offset
          mov  si,di               ;Record Address in SI
          add  bl,cl               ;First line in scrolled window
          call calo                ;Convert first line to offset
          xchg si,di               ;Exchange SI and DI
          push ds                  ;Store segment register on
          push es                  ;the stack
          mov  ax,VIO_SEG          ;Segment address of the video RAM
          mov  ds,ax               ;to DS
          mov  es,ax               ;and ES
supl:     mov  ax,di               ;Record DI in AX
          mov  bx,si               ;Record SI in BX
          mov  cl,dh               ;Number of column in counter
          rep movsw                ;Move a line
          mov  di,ax               ;Restore DI from AX
          mov  si,bx               ;Restore SI from BX
          add  di,160              ;Set next line
          add  si,160
          dec  dl                  ;Processed all lines ?
          jne  supl                ;NO --> move another line
          pop  es                  ;Get segment register from
          pop  ds                  ;stack
          pop  dx                  ;Get lower right corner
          pop  cx                  ;Read number of lines
          pop  bx                  ;Get upper left corner
          mov  bl,dl               ;Lower line to BL
          sub  bl,cl               ;Deduct number of lines
          inc  bl
          mov  ah,07h              ;Color : black on white
```

```
          call clear              ;Erase lines freed

          pop  si                 ;CX and DX have already
          pop  di                 ;been read
          pop  bx
          pop  ax

          ret                     ;Back to caller

scrollup  endp

;-- SCROLLDN: scrolls a window down N lines ---------------
;-- Input   :  BL = line upper left
;--             BH = column upper left
;--             DL = line lower right
;--             DH = column lower right
;--             CL = number of lines to scroll
;-- Output   : none
;-- register : only FLAGS are changed
;-- Info     : display lines released are erased

scrolldn  proc near

          cld                     ;Increment on string instructions

          push ax                 ;Store all changed registers on the
          push bx                 ;stack
          push di                 ;In this case the sequence
          push si                 ;must be observed !

          push bx                 ;These three registers are returned
          push cx                 ;from the stack before the end
          push dx                 ;of the routine

          sub  dh,bh              ;Calculate the number of the column
          inc  dh
          mov  al,bl              ;Record line upper left in AL
          mov  bl,dl              ;Line upper right to line upper left
          call calo               ;Convert upper left into offset
          mov  si,di              ;Record address in SI
          sub  bl,cl              ;Deduct number of lines to scroll
          call calo               ;Convert upper left in offset
          xchg si,di              ;Exchange SI and DI
          sub  dl,al              ;Calculate number of lines
          inc  dl                 ;Deduct number
          sub  dl,cl              ;of lines to be scrolled
          push ds                 ;Push segment register onto stack
          push es
          mov  ax,VIO_SEG         ;Segment address of video RAM
          mov  ds,ax              ;to DS
          mov  es,ax              ;and ES
sdn1:     mov  ax,di              ;Move DI to AX
          mov  bx,si              ;Move SI to BX
          mov  cl,dh              ;Number column in counter
          rep movsw               ;Scroll one line
          mov  di,ax              ;Get DI from AX
          mov  si,bx              ;Restore SI from BX
          sub  di,160             ;Set next line
          sub  si,160
          dec  dl                 ;All lines processed ?
          jne  sdn1               ;NO --> scroll another line
          pop  es                 ;Get segment register from
          pop  ds                 ;stack
          pop  dx                 ;Return lower right corner
          pop  cx                 ;Return number of lines
          pop  bx                 ;Return upper left corner
          mov  dl,bl              ;Upper line to DL
          add  dl,cl              ;Add number of lines
          dec  dl
          mov  ah,07h             ;Color : black on white
```

```
          call clear              ;Erase lines which were released

          pop  si                 ;CX and DX are
          pop  di                 ; already returned
          pop  bx
          pop  ax

          ret                     ;Back to caller

scrolldn  endp

;-- CLS: Clear the complete screen ------------------------------
;-- Input   : none
;-- Output   : none
;-- register : only FLAGS are changed

cls       proc near

          mov  ah,07h             ;Color is white on black
          xor  bx,bx              ;Upper left is (0/0)
          mov  dx,4F18h           ;Lower right is (79/24)

          ;-- Execute Clear ------------------------------------------

cls       endp

;-- CLEAR: fills a designated display with space characters ----
;-- Input   : AH = Attribute/color
;--              BL = line upper left
;--              BH = column upper left
;--              DL = line lower right
;--              DH = column lower right
;-- Output   : none
;-- register : only FLAGS are changed

clear     proc near

          cld                     ;Increment on string instructions
          push cx                 ;Store all registes which
          push dx                 ;are changed on the stack
          push si
          push di
          push es
          sub  dl,bl              ;Calculate number of lines
          inc  dl
          sub  dh,bh              ;Calculate number of columns
          inc  dh
          call calo               ;Offset address of upper left corner
          mov  cx,VIO_SEG         ;Segment address of the video RAM
          mov  es,cx              ;to ES
          xor  ch,ch              ;Hi-bytes of the counter to 0
          mov  al," "             ;Space character
clearl:   mov  si,di              ;Move DI to SI
          mov  cl,dh              ;Number of column in counter
          rep stosw               ;Store space character
          mov  di,si              ;Restore DI from SI
          add  di,160             ;Set in next line
          dec  dl                 ;All lines processed ?
          jne  clearl             ;NO --> erase another line

          pop  es                 ;Restore registers from
          pop  di                 ;stack
          pop  si
          pop  dx
          pop  cx
          ret                     ;Back to caller

clear     endp

;-- PRINT: outputs a string on the Display -------------------
```

```
;-- Input    : AH = Attribute/color
;--            DI = offset address of the first character
;--            SI = offset address of the string to DS
;-- Output   : DI points behind the last character output
;-- register : AL, DI and FLAGS are changed
;-- Info     : the string must be terminated with a NUL-character.
;--            other control characters are not recognized

print     proc near

          cld                      ;Increment on string instructions
          push si                  ;Store SI, DX and ES on the stack
          push es
          push dx
          mov  dx,VIO_SEG          ;Segment address of the video RAM
          mov  es,dx               ;First to DX and then to ES
          jmp  print1              ;YES --> Output finished

print0:   stosw                    ;Store attribute and color in V-RAM
print1:   lodsb                    ;Get next character from the string
          or   al,al               ;Is it NUL
          jne  print0              ;NO --> output

printe:   pop  dx                  ;Get SI, DX and ES back from stack
          pop  es
          pop  si
          ret                      ;Back to caller

print     endp

;- CALO: converts line and column into offset address ----------------
;-- Input    : BL = line
;--            BH = column
;-- Output   : DI = the offset address
;-- Registers: DI and FLAGS are changed

calo      proc near

          push ax                  ;Store AX on the stack
          push bx                  ;Store BX on the stack

          shl  bx,1                ;Column and line times 2
          mov  al,bh               ;Column to AL
          xor  bh,bh               ;Get Hi-byte
          mov  di,[linen+bx]       ;Offset address of the line
          xor  ah,ah               ;HI-byte for column offset
          add  di,ax               ;Add line- and column offset

          pop  bx                  ;Get BX from stack again
          pop  ax                  ;Get AX from stack again
          ret                      ;Back to caller

calo      endp

;== End ===================================================

code      ends                     ;End of the CODE segment
          end  demo                ;Start program execution w/ demo
```

10.3 The Hercules Graphic Card

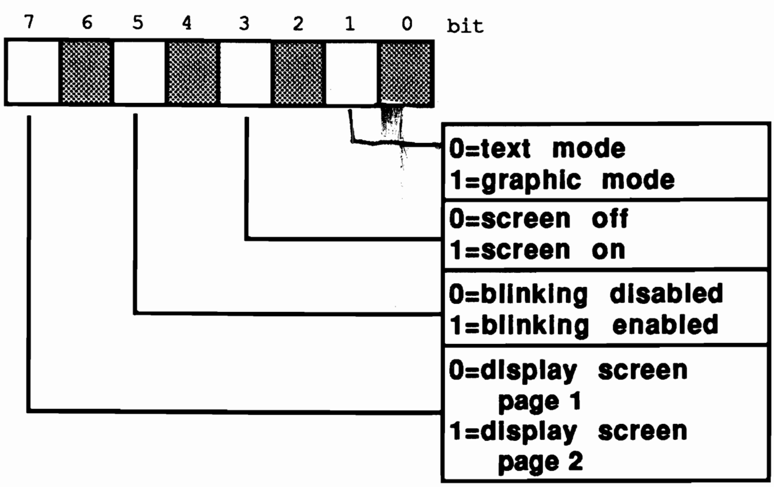
The Hercules display adapter displays text in both text mode and graphics mode, with a graphic resolution of 720x348 pixels. This card contains enough RAM for two display pages. Each display page is 32K, so video RAM can accept a 4K text page and a graphic page. The first display page extends from address B000:0000 to B000:7FFF. The second screen page goes from B000:8000 to B000:FFFF.

Hercules video RAM

The Hercules card's video RAM in text mode has the same cursor character and port addresses as the IBM monochrome display adapter. With the graphic capabilities, only a few bits in the status and control register are different from the monochrome card. An additional configuration register can be addressed from 3BFH. You can write to this register only. Only bits 0 and 1 are of interest to the programmer. The former indicates whether the graphic mode can be switched on (1) or not (0). Bit 1 determines whether the second display page can be used. Bit 1 contains the value 1 if the second page is usable.

To avoid conflicts with other video cards (especially color cards), both bits are set to 0 at the start of the system so that neither graphic mode nor the second display page are accessible at first. Application programs must configure the Hercules display adapter through the configuration register if the programs require graphic mode or the second screen page.

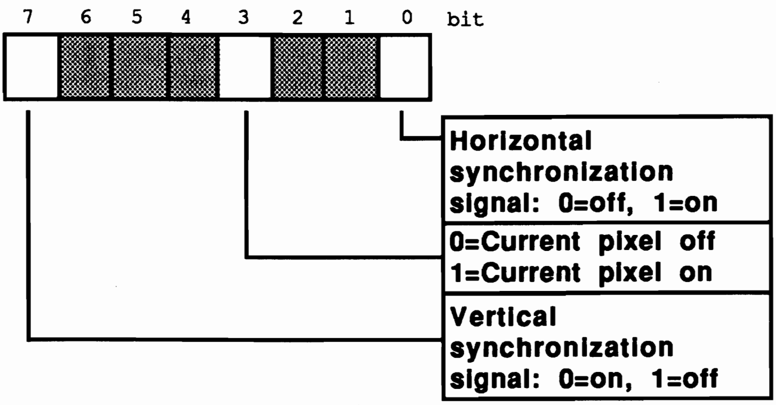
The control register of the Hercules graphic card has some differences from that of the MDA discussed in the preceding section.



The Hercules control register (3B8H)

Unlike the IBM monochrome display adapter, bit 0 is unused and doesn't have to be set to 1 during the system boot. Bit 1 determines text or graphic mode: a 0 in bit 1 enables text mode, while a 1 in bit 1 enables graphic mode. As you shall see in the following examples, changing these bits isn't enough to switch between text and graphic modes. The internal registers of the 6845 must be reset as well. During this process, the screen display must be switched off to prevent the 6845 from creating garbage during its reprogramming.

The Hercules card has a seventh bit in this register. Its contents determine which of the two screen pages appear on the monitor screen. If this bit is 0, the first screen page appears; a 1 calls the second screen page on the screen. Independent of each other, the user can write to or read from either page at any time. You can only write to this register; attempts to read this register return the value FFH. Because of this, it is impossible to switch off the display simply by reading the contents of the status register and erasing bit 3, regardless of the display mode and the screen page selected.



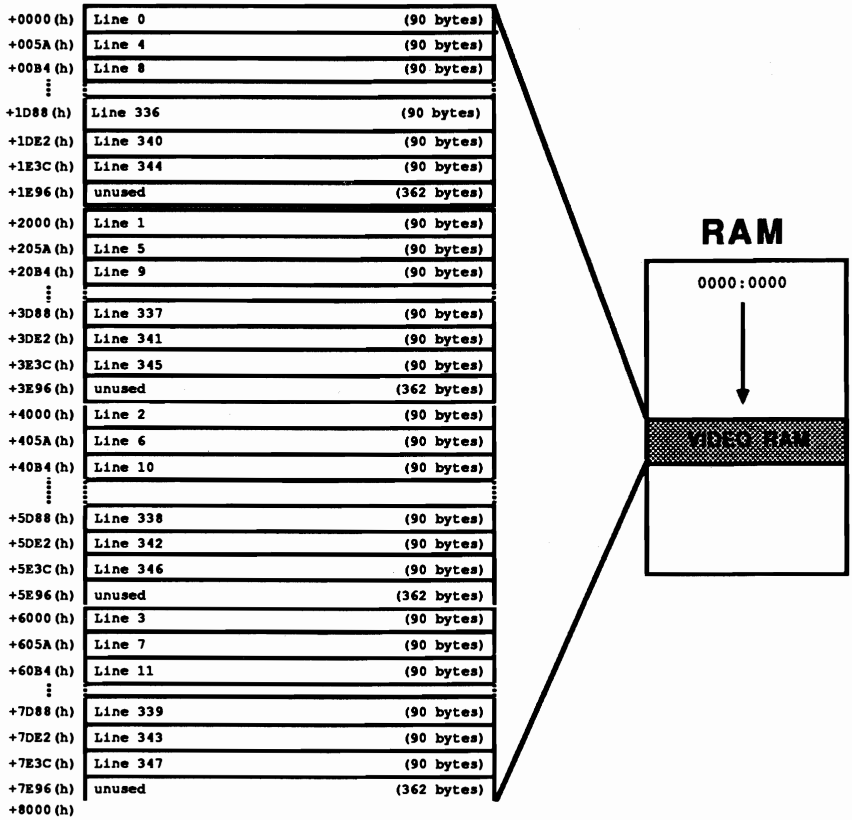
Hercules status register (3BAH)

Only the significance of bit 7 makes this register different from the IBM monochrome card. It's always set to 0 when the 6845 sends a vertical synchronization signal to the display. This signal is always sent when the last screen line has been constructed. The electron beam, which constructs the display, then jumps to the first line of the screen to start constructing a new screen.

Since the Hercules card uses the same processor as the IBM card, the internal registers of the 6845 and their meaning are identical to the IBM card. The index register and data register are also located at the same address. The following values must be assigned to the various registers in the text and graphic modes respectively:

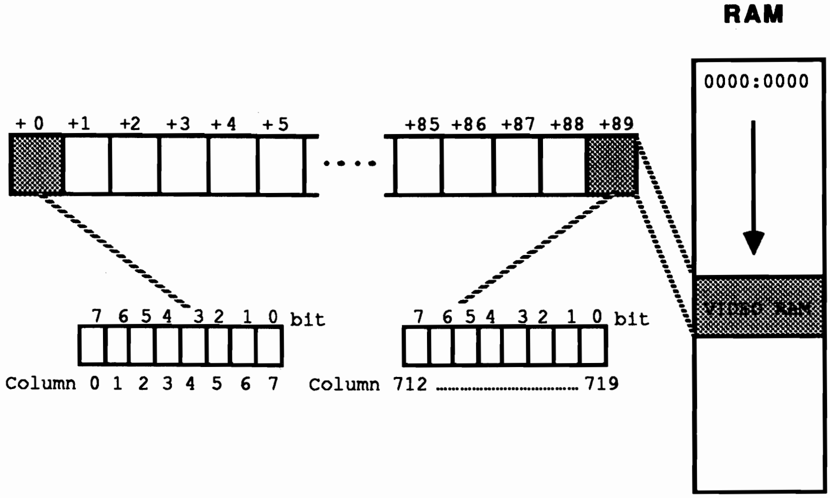
No.	Meaning	Text	Graphic
0	Horizontal character seeded	97	53
1	Horizontal character displayed	80	45
2	Horiz. synchronization signal after...character	82	46
3	Horiz. synchronization signal width	15	7
4	Vertical character seeded	25	91
5	Vertical character justified	6	2
6	Vertical character displayed	25	87
7	Vert. synchronization signal after...character	25	87
8	Interlace mode	2	2
9	Number of ccan-lines per line	13	3
10	Starting line of blinking cursor	11	0
11	Ending line of the blinking cursors	12	0
12	High byte of screen page starting address	0	0
13	Low byte of screen page starting address	0	0
14	High byte of blinking cursor char. address	0	0
15	Low byte of blinking cursor char. address	0	0
16	Reserved		
17	Reserved		

As mentioned earlier, the Hercules card in graphic mode provides 348x720 resolution. Every pixel on the screen corresponds to one bit in the video RAM. If the corresponding bit contains the value 1, the dot is visible on the display, otherwise it remains dark. The following figure shows the construction of the video RAM in the graphic mode.



Video RAM and the screen under construction

The bit patterns of the individual lines in the video RAM aren't arranged sequentially, as you might have assumed. The 32K of video RAM is divided into four 8K blocks. The first block contains the bit pattern for any lines divisible by 4 (0, 4, 8, 12, etc.). The second block contains the bit patterns for lines 1, 5, 9, 13 etc. The third block contains the bit patterns for lines 2, 6, 10, 14, etc., while the last block contains lines 3, 7, 11, 15 etc. When the 6845 generates a display, it obtains information for screen line zero from the first data block, screen line one from the second data block, etc. After it has obtained the contents of the third screen line from the fourth data block, it accesses the first data block again for the structure of the fourth line. Each line requires 90 bytes within the individual data blocks—every pixel requires a bit, and 720 pixels divided by 8 bits (per byte) equals 90. The first 90 bytes in the first memory area provide the bit pattern for screen line zero, and the 90 bytes following provide the bit pattern for the fourth screen line. The zero byte of one of these 90-byte sets represents the first eight columns of a screen line (columns 0-8). The first byte represents columns 8-15, etc. Within one of these bytes, bit 7 corresponds to the left screen pixel and bit 0 corresponds to the right screen pixel.



Relationship between 90-line bytes and screen display

If the screen pixels of a line (0 to 719) and the screen pixels of a column (0 to 347) are sequentially numbered, an equation indicates the address of the bytes relative to the beginning of the screen page. This address contains the information for a pixel with the coordinates X/Y.

To determine the bit within the byte which represents the pixel, the following formula can be used:

```
Address = 2000H * (Y mod 4) + 90 * int(Y/4) + int(X/8)
```

The following program demonstrates the abilities of the Hercules display adapter. The individual routines within this program have some differences from the routines shown in the monochrome display adapter demo program from the previous section. The routines here enable access to both screen pages, and support the Hercules graphic mode.

Assembler listing: VHERC.ASM

```
;**********************************************************************;
;*                              V H E R C                             *;
;*--------------------------------------------------------------------*;
;*    Task           : makes a basic function available for           *;
;*                     access to the HERCULES GRAPHICS CARD           *;
;*--------------------------------------------------------------------*;
;*    Info           : all functions partition the screen display     *;
;*                     into columns  0-79 and lines 0-24 (text mode)  *;
;*                     & columns 0-719 and lines 0-347 (graphic mode)*;
;*--------------------------------------------------------------------*;
;*    Author         : MICHAEL TISCHER                                *;
;*    developed on   : 8/11/87                                        *;
```

```
;*    last update     : 6/15/89                                        *;
;*---------------------------------------------------------------------*;
;*    assembly        : MASM VHERC;                                    *;
;*                      LINK VHERC;                                    *;
;*---------------------------------------------------------------------*;
;*    call            : VHERC                                          *;
;***********************************************************************;

;== Constants ==========================================================

CONTROL_REG  = 03B8h                ;Control register port address
ADDRESS_6845 = 03B4h                ;6845 address register
DATA_6845    = 03B5h                ;6845 data register
CONFIG_REG   = 03BFh                ;Configuration register
VIO_SEG      = 0B000h               ;Video RAM segment address
CUR_START    = 10                   ;Reg. # for CRTC: Start cursor line
CUR_END      = 11                   ;Reg. # for CRTC: End cursor line
CURPOS_HI    = 14                   ;Reg. # for CRTC: Cursor pos hi byte
CURPOS_LO    = 15                   ;Reg. # for CRTC: Cursor pos lo byte

DELAY        = 20000                ;Count for delay loop

;== Macros ==============================================

setmode   macro modus             ;Set control register

          mov  dx,CONTROL_REG     ;Screen control register address
          mov  al,modus           ;Put new mode in AL register
          out  dx,al              ;Send mode to control register

          endm

setvk     macro                   ;Write value to CRTC registers
                                  ;Input: AL = register number
                                  ;       AH = Value for register

          mov  dx,ADDRESS_6845    ;Index register address
          out  dx,ax              ;Display register number and new value

          endm

;== Stack ===============================================

stack     segment para stack      ;Definition of stack segment

          dw 256 dup (?)          ;Stack is 256 words in size

stack     ends                    ;End of stack segment

;== Data ================================================

data      segment para 'DATA'     ;Define data segment

;== Data needed for demo program ========================

initm     db 13,10,"VHERC (c) 1987 by Michael Tischer",13,10,13,10
          db "This demonstration program runs only with "
          db " a HERCULES",13,10,"graphics card. If your PC "
          db "has another type of display card, ",13,10
          db "please input an >s< to stop the "
          db " program.",13,10,"Otherwise please press any "
          db "key to start the ",13,10
          db "program ...",13,10,"$"

str1      db 1,17,16,2,7,0
str2      db 2,16,17,1,7,0

domes     db 13,10
          db "This program creates a short graphic demo ",13,10
          db "and a text demo. Pressing a key during the",13,10
```

```
          db "demo ends the program.",13,10
          db "Press a key to start the program...",13,10,"$"

;== Table of line offset addresses ========================

lines     dw  0*160,1*160,2*160 ;Beginning addresses of the lines as
          dw  3*160,4*160,5*160 ;offset addresses in video RAM
          dw  6*160,7*160,8*160
          dw  9*160,10*160,11*160,12*160,13*160,14*160,15*160,16*160
          dw 17*160,18*160,19*160,20*160,21*160,22*160,23*160,24*160

grafikt   db 35h, 2Dh, 2Eh, 07h, 5Bh, 02h  ;Register values for the
          db 57h, 57h, 02h, 03h, 00h, 00h  ;graphic mode

textt     db 61h, 50h, 52h, 0Fh, 19h, 06h  ;Register values for the
          db 19h, 19h, 02h, 0Dh, 0Bh, 0ch  ;text mode

data      ends                 ;End of data segment

;== Code segment ==========================================

code      segment para 'CODE'     ;Definition of the code segment

          org 100h

          assume cs:code, ds:data, es:data, ss:stack

;== this is only the Demo-Program =========================

demo      proc far

          mov  ax,data             ;Get segment address of data segment
          mov  ds,ax               ;Load into DS
          mov  es,ax               ;and ES

          ;-- Opening msg., wait for input -------------------

          mov  ah,9                ;Output function number for string
          mov  dx,offset initm     ;address of the message
          int  21h                 ;Call DOS interrupt

          xor  ah,ah               ;Get function number for key
          int  16h                 ;Call BIOS keyboard interrupt
          cmp  al,"s"              ;Was <s> entered?
          je   ende                ;YES--> End program
          cmp  al,"S"              ;Was <S> entered?
          jne  startdemo           ;NO --> Start demo

ende:     mov  ax,4C00h            ;Function number - end program
          int  21h                 ;Call DOS interrupt 21H

startdemo label near
          mov  ah,9                ;Output function number for string
          mov  dx,offset domes     ;address of the message
          int  21h                 ;Call DOS interrupt

          xor  ah,ah               ;Get function number for key
          int  16h                 ;Call BIOS keyboard interrupt

          ;-- Initialize graphic mode -------------------------

          mov  al,11b              ;Graphic and page 2 possible
          call config              ;Configure
          xor  bp,bp               ;Access display page 0
          call grafik              ;Switch to graphic mode
          xor  al,al
          call cgr                 ;Erase graphic page 0
          xor  bx,bx               ;Begin in the upper left
          xor  dx,dx               ;Display corner
          mov  ax,347              ;Vertical pixels
```

```
        mov  cx,719              ;Horizontal pixels
grl:    push cx                  ;Push horizontal pixels on stack
        mov  cx,ax               ;Vertical pixels in counter
        push ax                  ;Push vertical pixels on stack
gr2:    call spix                ;Set pixel
        inc  dx                  ;Increment line
        loop gr2                 ;Draw line
        pop  ax                  ;Get vert. pixels from stack
        sub  ax,3                ;next line 3 pixels less
        pop  cx                  ;Get horiz. pixels from stack
        push cx                  ;Store horizontal pixels
        push ax                  ;Push vertical pixels on stack
gr3:    call spix                ;Set pixel
        inc  bx                  ;Increment column
        loop gr3                 ;Draw line
        pop  ax                  ;Get vertical pixels from stack
        pop  cx                  ;Get horizontal pixels from stack
        sub  cx,6                ;Next line 6 pixels less
        push cx                  ;Record horizontal pixels
        mov  cx,ax               ;Vertical pixels in counter
        push ax                  ;Note vertical pixels on stack
gr4:    call spix                ;Set pixel
        dec  dx                  ;Decrement line
        loop gr4                 ;Draw line
        pop  ax                  ;Get vertical pixels from stack
        sub  ax,3                ;Next line 3 pixels less
        pop  cx                  ;Get horizontal pixels from stack
        push cx                  ;Record horizontal pixels
        push ax                  ;Record vertical pixels on stack
gr5:    call spix                ;Set pixel
        dec  bx                  ;Increment column
        loop gr5                 ;Draw line
        pop  ax                  ;Get vertical pixels from stack
        pop  cx                  ;Get horizontal pixels from stack
        sub  cx,6                ;Next line 6 pixels less
        cmp  ax,5                ;Is the vertical line longer than 5
        ja   grl                 ;YES --> continue

        xor  ah,ah               ;Wait for function nr. for key
        int  16h                 ;Call BIOS keyboard interrupt

        ;-- Initialize text mode -----------------------------

        call text                ;Switch on text mode
        mov  cx,0d00h            ;Switch on full cursor
        call cdef
        call cls                 ;Clear screen

        ;-- Display strings in display page 0 -----------------

        xor  bx,bx               ;Start in upper left display corner
        call calo                ;Convert to offset address
        mov  si,offset strl      ;Offset address of stringl
        mov  cx,16*25            ;The string is 5 characters long
demol:  call print               ;Output string
        loop demol

        ;-- Display strings in display page 1 -----------------

        inc  bp                  ;Process display page 1
        xor  bx,bx               ;Start in the upper left corner
        call calo                ;Convert to offset address
        mov  si,offset str2      ;Offset address of stringl
        mov  cx,16*25            ;string is 5 characters long
demo2:  call print               ;Output string
        loop demo2

demo3:  setmode 10001000b        ;Display text page 1

        ;-- short Pause ---------------------------------------------
```

```
        mov  cx,DELAY           ;Load counter
pause:  loop pause              ;Count to 65,536

        setmode 00001000b       ;Display page 0

        ;-- short pause ------------------------------------------
        mov  cx,DELAY           ;Load counter
pause1: loop pause1             ;Count to 65,536

        mov  ah,1               ;Test function nr. for key
        int  16h                ;Call BIOS-keyboard-Interrupt
        je   demo3              ;No key --> continue

        xor  ah,ah              ;Get function number for key
        int  16h                ;Call BIOS-keyboard-Interrupt

        mov  bp,0               ;Display page 1
        call cls                ;Clear screen
        mov  cx,0D0ch           ;Restore normal cursor
        call cdef
        call cls                ;Clear screen
        jmp  ende               ;End program

demo    endp

;== The actual functions follow ==========================

;-- CONFIG: configures the HERCULES card -----------------------------
;-- Input    : AL : bit 0 = 0 : Only text presentation possible
;--                         1 : also graphic presentation possible
;--                 bit 1 = 0 : RAM for display page 2 off
;--                         1 : RAM for display page 2 on
;-- Output   : none
;-- Register : AX and DX are changed

config  proc near

        mov  dx,CONFIG_REG      ;Address of configuration register
        out  dx,al              ;Set new value
        ret                     ;Back to caller

config  endp

;-- TEXT: switches the text presentation on --------------------------
;-- Input   : none
;-- Output   : none
;-- Register : AX and DX are changed

text    proc near

        mov  si,offset textt    ;Offset address of the register-table
        mov  bl,00100000b       ;Display page 0,text mode,blinking
        jmp  short vcprog       ;Program video-controller again

text    endp

;-- GRAFIK: switches on the graphic mode ------- ----------------------
;-- Input   : none
;-- Output   : none
;-- Register : AX and DX are changed

grafik  proc near

        mov  si,offset grafikt  ;Offset address of the register-table
        mov  bl,00000010b       ;Display page 0, graphic mode

grafik  endp

;-- VCPROG: programs the video controller -----------------------------
;-- Input   :    SI = address of a register-table
```

```
;--           BL = value for display-control-register
;-- Output   : none
;-- register : AX, SI, BH, DX and FLAGS are changed

vcprog    proc near

          setmode bl             ;Bit 3 = 0: display aus

          mov  cx,12             ;12 registers are set
          xor  bh,bh             ;Start with register 0
vcpl:     lodsb                  ;Get register value from the table
          mov  ah,al             ;Register value to AH
          mov  al,bh             ;Number of the register to AL
          setvk                  ;Transmit value to the controller
          inc  bh                ;Address next register
          loop vcpl              ;Set additional registers

          or   bl,8              ;Bit 3 = 1: display on
          setmode bl             ;Set new mode
          ret                    ;Back to caller

vcprog    endp

;-- cDEF: sets the start and end line of the cursor--------------------
;-- Input    : cL = start line
;--            cH = end line
;-- Output   : none
;-- register : AX and DX are changed

cdef      proc near

          mov  al,CUR_START      ;Register 10: start line
          mov  ah,cl             ;Start line to AH
          setvk                  ;Transmit to video-controller
          mov  al,CUR_END        ;Register 11: Endline
          mov  ah,ch             ;End line to AH
          setvk                  ;Transmit to video-controller
          ret

cdef      endp

;-- SETBLINK : sets the blinking display cursor -----------------------
;-- Input    : DI = offset address of the cursor
;-- Output   : none
;-- register : BX, AX and DX are changed

setblink  proc near

          mov  bx,di             ;Transmit offset to BX
          mov  al,CURPOS_HI      ;Register 15:Hi Byte of cursor offset
          mov  ah,bh             ;HI byte of the offset
          setvk                  ;Transmit to video-controller
          mov  al,CURPOS_LO      ;Register 15:Lo-Byte of cursor offset
          mov  ah,bl             ;Lo byte of the offset
          setvk                  ;Transmit to CRTC
          ret

setblink  endp

;-- GETVK    : reads a byte from one register of the video-controller -
;-- Input    : AL = number of the register
;-- Output   : AL = content of the register
;-- register : DX and AL are changed

getvk     proc near

          mov  dx,ADDRESS_6845   ;Address of the index register
          out  dx,al             ;Send number of the register
          jmp $+2                ;Short io pause
          inc  dx                ;Address of the index register
```

```
        in   al,dx                  ;Read content to AL
        ret                         ;Back to caller

getvk   endp

;-- SCROLLUp: scrolls a window by N lines upward ----------------------
;-- Input  :  BL = line upper left
;--           BH = column upper left
;--           DL = line lower right
;--           DH = column lower right
;--           CL = number of the lines to be scrolled
;--         : BP = number of the display page (0 or 1)
;-- Output  : none
;-- register : only FLAGS are changed
;-- Info     : the display lines released are erased

scrollup proc near

        cld                         ;Increment for string instructions
        push ax                     ;Store all changed registers
        push bx                     ;on the stack
        push di                     ;In this case the sequence
        push si                     ;must be followed !

        push bx                     ;These three registers are returned
        push cx                     ;from the stack before
        push dx                     ;the end of the routine
        sub  dl,bl                  ;Calculate number of lines
        inc  dl                     ;Deduct number
        sub  dl,cl                  ;of lines to be scrolled
        sub  dh,bh                  ;Calculate number of columns
        inc  dh
        call calo                   ;Convert upper left in offset
        mov  si,di                  ;Note address in SI
        add  bl,cl                  ;First line in scrolled window
        call calo                   ;Convert first line in offset
        xchg si,di                  ;Exchange SI and DI
        push ds                     ;Store segment register
        push es                     ;on the stack
        mov  ax,VIO_SEG             ;Segment address of the video RAM
        mov  ds,ax                  ;to DS
        mov  es,ax                  ;and ES
supl:   mov  ax,di                  ;Note DI in AX
        mov  bx,si                  ;Note SI in BX
        mov  cl,dh                  ;Number of columns in counter
        rep movsw                   ;Move a line
        mov  di,ax                  ;Restore DI from AX
        mov  si,bx                  ;Restore SI from BX
        add  di,160                 ;Set next line
        add  si,160
        dec  dl                     ;Processed all lines ?
        jne  supl                   ;NO --> move another line
        pop  es                     ;Get segment register from
        pop  ds                     ;stack
        pop  dx                     ;Get lower right corner
        pop  cx                     ;Get number of lines
        pop  bx                     ;Get upper left corner
        mov  bl,dl                  ;Lower line to BL
        sub  bl,cl                  ;Deduct number of lines
        inc  bl
        mov  ah,07h                 ;Color : black on white
        call clear                  ;Erase liberated lines

        pop  si                     ;CX and DX have been brought back
        pop  di                     ;already
        pop  bx
        pop  ax

        ret                         ;Back to caller
```

```
scrollup  endp

;-- SCROLLDN: scroll a Window by N lines upwards ----------------------
;-- Input    : BL = line upper left
;--            BH = column upper left
;--            DL = line lower right
;--            DH = column lower right
;--            CL = number of the lines to be scrolled
;--          : BP = number of the display page (0 or 1)
;-- Output   : none
;-- register : only FLAGS are changed
;-- Info     : released lines are deleted

scrolldn  proc near

          cld                    ;Increment on string instructions

          push ax                ;Secure all changed registers on the
          push bx                ;stack
          push di                ;In this case the sequence must
          push si                ;be followed!

          push bx                ;These three registers are
          push cx                ;returned from the stack before the
          push dx                ;end of the routine

          sub  dh,bh             ;Calculate number of columns
          inc  dh
          mov  al,bl             ;Record line upper left in AL
          mov  bl,dl             ;Line lower right top lower left
          call calo              ;Convert upper left in offset
          mov  si,di             ;Note address in SI
          sub  bl,cl             ;Deduct number of chars to scroll
          call calo              ;Convert upper left in offset
          xchg si,di             ;Exchange SI and DI
          sub  dl,al             ;Calculate number of lines
          inc  dl
          sub  dl,cl             ;Deduct number of lines to scroll
          push ds                ;Store segment register on the
          push es                ;stack
          mov  ax,VIO_SEG        ;Segment address of the video RAM
          mov  ds,ax             ;to DS
          mov  es,ax             ;and ES
sdnl:     mov  ax,di             ;Record DI in AX
          mov  bx,si             ;Record SI in BX
          mov  cl,dh             ;Number of columns in counter
          rep movsw              ;Move a line
          mov  di,ax             ;Restore DI from AX
          mov  si,bx             ;Restore SI from BX
          sub  di,160            ;Set next line
          sub  si,160
          dec  dl                ;All lines processed ?
          jne  sdnl              ;NO --> move another line
          pop  es                ;Get segment register from
          pop  ds                ;stack
          pop  dx                ;Get lower right corner
          pop  cx                ;Get number of lines
          pop  bx                ;Get upper left corner
          mov  dl,bl             ;Upper line to DL
          add  dl,cl             ;Add number of lines
          dec  dl
          mov  ah,07h            ;Color : black on white
          call clear             ;Erase liberated lines

          pop  si                ;CX and DX have already
          pop  di                ;been read
          pop  bx
          pop  ax

          ret                    ;Back to caller
```

```
scrolldn  endp

;-- cLS: clear the whole screen ------------------------------------
;-- Input    : BP = number of the display page (0 or 1)
;-- Output   : none
;-- register : only FLAGS are changed

cls       proc near

          mov  ah,07h             ;Color is white on black
          xor  bx,bx              ;Upper left is (0/0)
          mov  dx,4F18h           ;Lower right is (79/24)

          ;-- perform clear -------------------------------------------

cls       endp

;-- CLEAR: fills a designated display area with space character -------
;-- Input    : AH = Attribute/color
;--            BL = line upper left
;--            BH = column upper left
;--            DL = line lower right
;--            DH = column lower right
;--            BP = number of the display page (0 or 1)
;-- Output   : none
;-- register : only FLAGS are changed

clear     proc near

          cld                     ;Increment on string instructions
          push cx                 ;Secure all changed
          push dx                 ;registers on the stack
          push si
          push di
          push es
          sub  dl,bl              ;Calculate number of lines
          inc  dl
          sub  dh,bh              ;Calculate number of columns
          inc  dh
          call calo               ;Offset address of upper left corner
          mov  cx,VIO_SEG         ;Segment address of the video RAM
          mov  es,cx              ;to ES
          xor  ch,ch              ;Hi byte of the counter to 0
          mov  al," "             ;Space character
clearl:   mov  si,di              ;Note DI in SI
          mov  cl,dh              ;Number of columns in counter
          rep stosw               ;Store space character
          mov  di,si              ;Restore DI from SI
          add  di,160             ;Set next line
          dec  dl                 ;All lines processed ?
          jne  clearl             ;NO --> erase another line

          pop  es                 ;Get secured registers
          pop  di                 ;from the stack
          pop  si
          pop  dx
          pop  cx
          ret                     ;Back to caller

clear     endp

;-- PRINT: outputs a string on the display --------------------------
;-- Input    : AH = attribute/color
;--            DI = offset address of the first character
;--            SI = offset address of the strings to DS
;--            BP = number of the display page (0 or 1)
;-- Output   : DI points behind the last character to be output
;-- register : AL, DI and FLAGS are changed
;-- Info     : the string must ne terminated with NUL-character.
```

```
;--           other control characters are not recognized

print     proc near

          cld                        ;Increment on string instructions
          push si                    ;SI, DX and ES to the stack
          push es
          push dx
          mov  dx,VIO_SEG            ;First segment address of video RAM
          mov  es,dx                 ;to DX and then to ES
          jmp  print1                ;Get first character from string
print0:   stosw                      ;Store attribute and color in V-RAM
print1:   lodsb                      ;Get next character from the string
          or   al,al                 ;Is it NUL
          jne  print0                ;NO --> output

printe:   pop  dx                    ;Get SI, DX and ES from stack again
          pop  es
          pop  si
          ret                        ;Back to caller

print     endp

;-- cALO: converts line and column into offset address ----------------
;-- Input    : BL = line
;--            BH = column
;--            Bp = number of the display page (0 or 1)
;-- Output   : DI = offset address
;-- register : DI and FLAGS are changed

calo      proc near

          push ax                    ;Record AX on the stack
          push bx                    ;Record BX on the stack

          shl  bx,1                  ;Column and line times 2
          mov  al,bh                 ;Column to AL
          xor  bh,bh                 ;Hi byte
          mov  di,[lines+bx]         ;Get offset address of the line
          xor  ah,ah                 ;Hi byte for column offset
          add  di,ax                 ;Add lines- and column offset
          or   bp,bp                 ;Display page 0?
          je   caloe                 ;YES --> address ok

          add  di,8000h              ;Add 32 KB for display page 1

caloe:    pop  bx                    ;Get BX from stack again
          pop  ax                    ;Get AX from the stack again
          ret                        ;Back to caller

calo      endp

;-- CGR: clear the complete graphic screen ----------------------------
;-- Input    : BP = number of the display page (0 or 1)
;--            AL = 00H : erase all pixels
;--                 FFH : set all pixels
;-- Output   : none
;-- register : AH, BX, cX, DI and FLAGS are changed

cgr       proc near

          push es                    ;Record ES on the stack
          cbw                        ;Expand AL to AH
          xor  di,di                 ;Offset address in video RAM
          mov  bx,VIO_SEG            ;Segment address display page 0
          or   bp,bp                 ;Erase page 1?
          je   cgr1                  ;NO --> erase page 0

          add  bx,0800h              ;Segment address display page 1
```

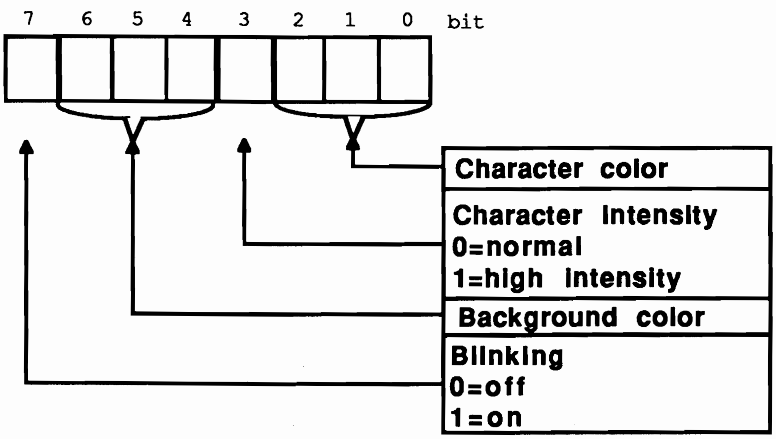
```
cgrl:   mov  es,bx              ;Segment address to segment register
        mov  cx,4000h           ;A page is 16K-words
        rep stosw               ;Fill page
        pop  es                 ;Get ES from stack
        ret                     ;Back to caller

cgr     endp

;-- SPIX: sets a pixel in the graphic display ---------------------------
;-- Input    : BP = number of the display page (0 or 1)
;--            BX = column (0 to 719)
;--            DX = line   (0 to 347)
;-- Output   : none
;-- register : AX, DI and FLAGS are changed

spix    proc near

        push es                 ;Store ES on the stack
        push bx                 ;Store BX on the stack
        push cx                 ;Store cX on the stack
        push dx                 ;Store DX on the stack

        xor  di,di              ;Offset address in video RAM
        mov  cx,VIO_SEG         ;Segment address display page 0
        or   bp,bp              ;Access page 1 ?
        je   spix1              ;NO --> access page 0

        mov  cx,0800h           ;Segment address display page 1

spix1:  mov  es,cx              ;Segment address in segment register
        mov  ax,dx              ;Move line to AX
        shr  ax,1               ;Shift line right 2 times
        shr  ax,1               ;This divides by four
        mov  cl,90              ;The factor is 90
        mul  cl                 ;Multiply line by 90
        and  dx,11b             ;AND all bits except for  0 and 1
        mov  cl,3               ;3 shifts
        ror  dx,cl              ;Rotate right  (* 2000H)
        mov  di,bx              ;Column to DI
        mov  cl,3               ;3 shifts
        shr  di,cl              ;divide by 8
        add  di,ax              ;+ 90 * int(line/4)
        add  di,dx              ;+ 2000H * (line mod 4)
        mov  cl,7               ;Maximum of 7 moves
        and  bx,7               ;Column mod 8
        sub  cl,bl              ;7 - column mod 8
        mov  ah,1               ;Determine bit value of the pixels
        shl  ah,cl
        mov  al,es:[di]         ;Get 8 pixels
        or   al,ah              ;Set pixel
        mov  es:[di],al         ;Write 8 pixels ;

        pop  dx                 ;Get DX from stack
        pop  cx                 ;Get cX from stack
        pop  bx                 ;Get BX from stack
        pop  es                 ;Get ES from stack
        ret                     ;Back to caller

spix    endp

;== End ==============================================================

code    ends                    ;End of the code segment
        end  demo
```

10.4 The IBM Color Card

The IBM Color/Graphics Adapter (CGA) supports two text modes and three different graphic modes. Like the other two cards, the CGA is based on a 6845 video processor and is equipped with 16K of video RAM which begins at address B800:0000.

Text modes

Besides the normal text mode of 25 lines and 80 columns, the CGA also has a text mode consisting of 25 lines and 40 columns. This 40-column mode displays characters twice as wide as normal 80-column mode. CGA characters are displayed in an 8x8 matrix, which results in a less distinct display than monochrome display adapter text. The CGA's video RAM assignment is almost identical to that of the monochrome card. The attribute byte is different from that of the monochrome display adapter.



Color/Graphics Adapter attribute byte

The lower four bits of the attribute byte indicate one of the 16 available colors. The meanings of the upper four bits depend on whether blinking is active. If it is active, bits 4 to 6 indicate the background color (taken from one of the first eight colors of the color palette), while bit 7 determines whether or not the characters blink. If blinking is disabled, bits 4 to 7 indicate the background color (taken from one of the 16 available colors).

Decimal	Hexadecimal	Binary	Color
0	0	0000	Black
1	1	0001	Blue
2	2	0010	Green
3	3	0011	Cyan
4	4	0100	Red
5	5	0101	Magenta
6	6	0110	Brown
7	7	0111	Light gray
8	8	1000	Dark gray
9	9	1001	Light blue
10	A	1010	Light green
11	B	1011	Light cyan
12	C	1100	Light red
13	D	1101	Light magenta
14	E	1110	Yellow
15	F	1111	White

Color/Graphics Adapter color palette

Each 80x25 text page requires 4,000 bytes of video RAM. 16K allows a total of four text pages. The first display page starts at address B800:0000, the second at B800:1000, the third at B800:2000 and the last at B800:3000. The 40x25 mode allows storage of eight display pages, because each display page only requires 2,000 bytes in this mode. The first display page starts at address B800:0000, the second at B800:0800, the third at B800:1000, etc.

Graphic modes

The CGA supports three different graphic modes, of which only two are usually used. The *color-suppressed* mode displays 160x100 pixels with 16 colors. The 6845 supports this resolution, but the rest of the hardware doesn't offer color-suppressed mode support. The remaining two graphic modes have resolutions of 320x200 and 640x200 respectively. The 320x200 resolution permits four-color graphics, while 640x200 resolution only allows two colors.

320x200 resolution

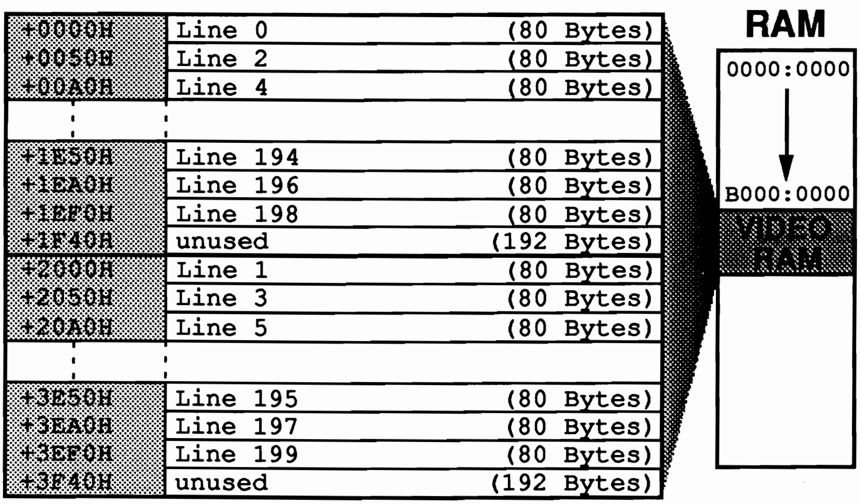
The CGA uses up all 16K of its video RAM for displaying a graphic in 320x200 resolution with four colors. This limits the user to one graphic page at a time. Of the four colors permitted, the background can be selected from the 16 available colors. The other three colors originate from one of the two user-selected color palettes, which contain three colors each.

Palette 1:	Color 1: Cyan	Palette 2:	Color 1: Green
	Color 2: Violet		Color 2: Red
	Color 3: White		Color 3: Yellow

Since a total of four colors are available, each screen pixel requires two bits. Four bits can represent the color numbers (0 to 3). The following values correspond to the various colors:

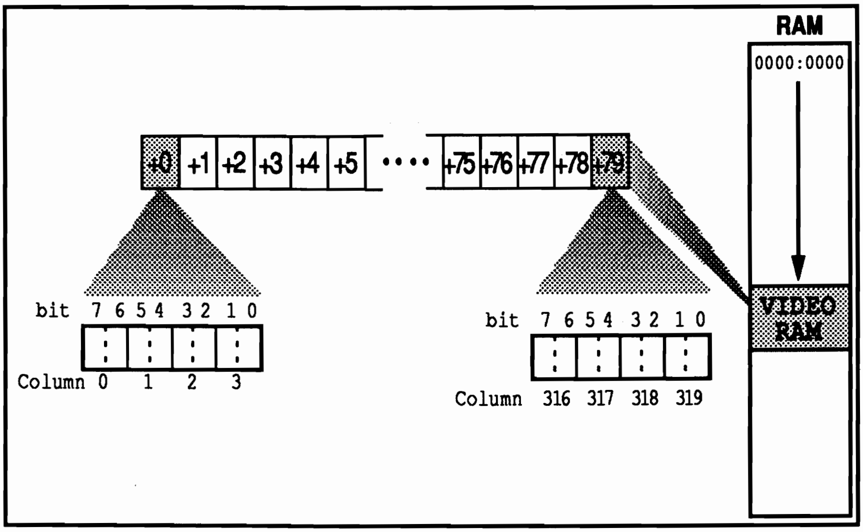
0 00(b) = freely selectable background color
1 01(b) = color 1 of the selected palette
2 10(b) = color 2 of the selected palette
3 11(b) = color 3 of the selected palette

The video RAM assignment in this mode is similar to that of the Hercules card during graphic display. The individual graphic lines are stored in two different blocks of memory. The first block, which begins at address B800:0000, contains the even lines (0, 2, 4...); the second block, which begins at B800:2000, contains odd lines (1,3,5).



Video RAM assignment in graphic mode (blocking)

Each graphic line within the two blocks requires 80 bytes, since the 320 pixels in a line are coded into four pixels to a byte. The first byte in a graphic line (an 80-byte series) corresponds to the first four dots of the graphic on the screen. Bits 7 and 8 contain the color information for the leftmost pixel, while bits 0 and 1 contain the color information for the rightmost pixel of the byte.



Graphic line coding in 320x200 resolution

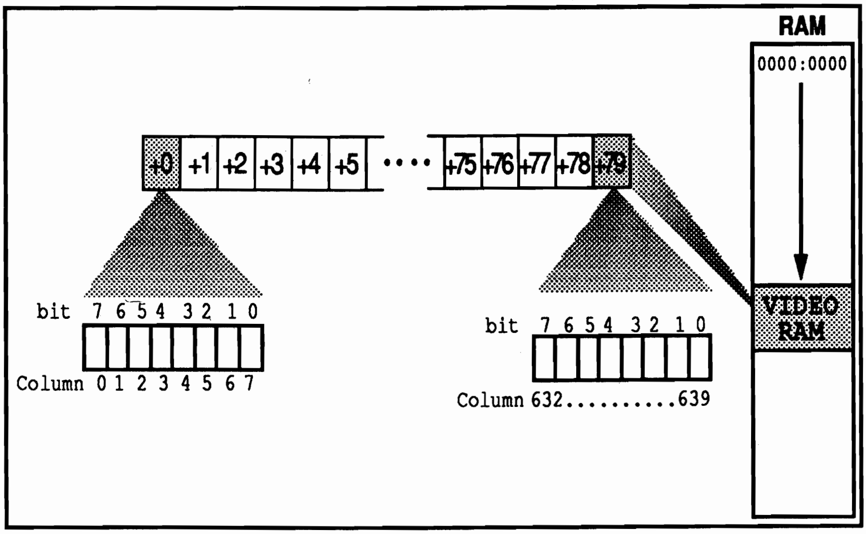
A formula can be derived with the help of this information to determine the byte in video RAM, similar to the Hercules card. This byte is relative to the starting address of the screen page, which contains the color information for a pixel. The screen column (0—319) is designated as X and the screen line (0—199) as Y:

```
Address = 2000H * (Y mod 2) + 80 * int(Y/2) + int(X/4)
```

To determine the number of the two bits within this byte which represents the pixel, use the following formula:

```
Bit number = 6 - 2 * (X mod 4)
```

For example, if this formula returns 4, this means that the color information for the dot is coded into bits 4 and 5.



Graphic line coding in 640x200 resolution

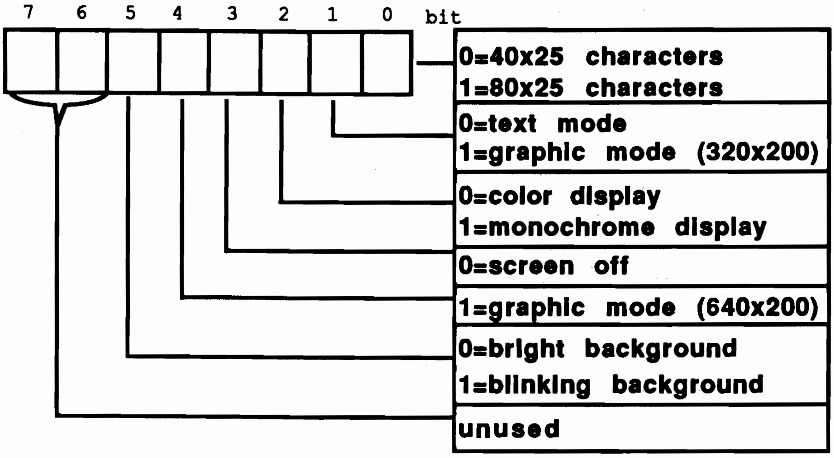
640x200 resolution

High-resolution mode with a resolution of 640x200 dots only allows the use of two colors. The video RAM assignment in this mode is similar to 320x200 mode. Each line displays twice as many pixels, with one bit encoding the line instead of 2 bits. Because of this, one screen line requires 880 bytes. Therefore the formulas for access to a screen pixel are similar.

```
Address = 2000H * (Y mod 2) + 80 * int(Y/2) + int(X/8)

Bit number = 7 - (X mod 8)
```

CGA registers

The CGA has a mode selection register at address 3D8H which is comparable with the control register of the monochrome display adapter. You can write to this register but not read it.



Mode selection register

Bit layout

Bit 0 of this register determines the text mode display of 80 or 40 columns per line. A 1 in bit 0 displays 80 columns, while a 0 in bit 0 displays 40 columns.

The status of bit 1 switches the CGA from text mode to the 320x200 bit-mapped graphic mode. A 1 in this register selects graphic mode, while a 0 selects text mode.

Bit 2 should be of interest to any users who want to operate their CGA with a monochrome monitor. If this bit contains the value 1, the 6845 suppresses the color signal, displaying monochrome mode only.

Bit 3 is responsible for creating screens. If it contains the value 0, the screen remains black. This suppression is useful when changing between display modes; it prevents sudden signals from reaching the monitor which could cause damage.

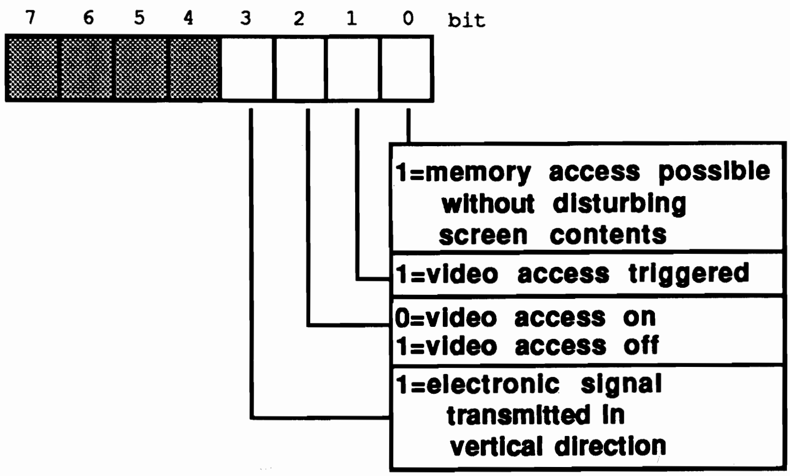
Bit 4 enables and disables 640x200 bitmapped graphic mode. A 1 in bit 4 enables this mode, while a 0 disables it.

Bit 5 has the same significance as in the monochrome card. If it contains a 0, blinking stops and bit 7 returns one of the 16 available background colors. This bit contains a default value of 1, which causes blinking characters.

The various text or graphic modes and the color or monochrome display can be selected in these modes with this register. Bits 0, 1, 2 and 4 are used for this. The following table shows how these bits must be programmed to obtain certain modes:

Bit 4	Bit 2	Bit 1	Bit 0	Result
0	1	0	0	40x25 text monochrome
0	0	0	0	40x25 text color
0	1	0	1	80x25 text monochrome
0	0	0	1	80x25 text color
0	1	1	0	320x200 graphic monochrome
0	0	1	0	320x200 graphic color
1	1	1	0	640x200 graphic monochrome

The CGA also has a status register similar to the status register in the monochrome display adapter. The following figure shows the construction of this register, which can be found at address 3DAH. It is a read-only register.



Status register structure

Bit 0 of this register always contains the value 1 when the 6845 sends a horizontal synchronization signal to the monitor. This signal is transmitted when the creation of a line ends and the CRT's electron beam reaches the end of the screen line. The electron beam then jumps back to the left corner of the screen line. The bit gets its significance from the condition that the CGA doesn't always allow data reading or writing within video RAM.

Flickering and the CGA

This problem occurs because the 6845 must continuously access video RAM to read its contents for screen display. If a program tries to transmit data to video RAM, problems can arise when the 6845 accesses video RAM at the same time. The result of this memory collision is an occasional flickering on the screen.

To avoid this problem, you should only access video RAM when the 6845 is not accessing it. This only occurs when a horizontal synchronization signal travels to the screen, because it requires a moment of time until the electron beam has carried out this instruction. For this reason, the status register must be read before every video RAM access on a CGA. This process must be repeated until bit 0 contains the value 1. When this happens, a maximum of two bytes can then be transmitted to video RAM.

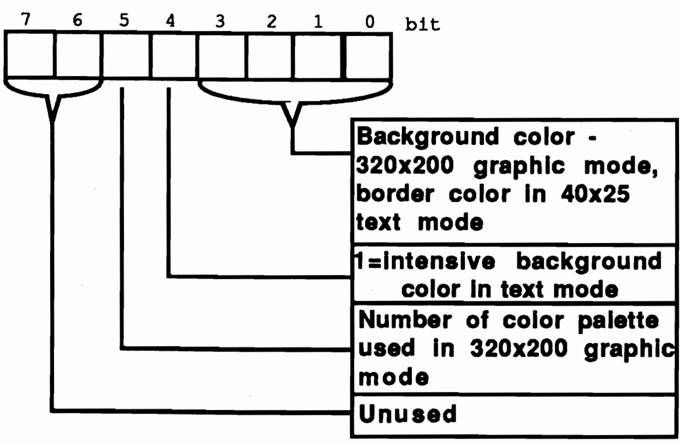
Demonstration program

The program at the end of this section demonstrates how this process functions. This delay in video RAM access doesn't occur with monochrome cards because they are equipped with special hardware logic and fast RAM chips. This is also true of most of the newer model color cards. Before waiting for the horizontal synchronization signal, which results in an enormous delay of the display output, the user should try direct access to video RAM to test his color card's reaction time.

If many accesses to video RAM occur within a short period of time (e.g., scrolling the screen), the electron beam doesn't respond fast enough. The screen should be switched off using bit 3 of the mode selection register. This prevents the 6845 from accessing video RAM, permitting unlimited user access to video RAM. When data transfer ends, the screen can be switched on again. BIOS uses this method during scrolling, which results in the flickering "silent movie effect."

Color selection register

The color selection register is located at address 3D9H. This register is write-only (cannot be read).



Color selection register

The meanings of individual bits in this register depend on the display mode. Text mode uses the lowest four bits for assigning the background color from the 16 available colors. In 320x200 graphic mode, these four bits indicate the color of all pixels represented by the bit combination 00(b) (background color).

Bit 5 selects the color palette for 320x200 mode. If this bit contains the value 1, the first color palette (cyan, violet, white) is selected. A value of 0 selects the second color palette (green, yellow, red).

Internal registers

The 18 internal registers of the 6845 on this card are accessed exactly like the monochrome card. The only difference is that the index and the data register are located at 3D4H and 3D5H. The following table shows the contents which the register must have for various display modes.

No.	Meaning	Text1	Text2	Graphics
0	Horiz. characters seeded	56	113	56
1	Horiz. characters displayed	40	80	40
2	Horiz. synchronization signal to .... Characters	45	90	45
3	Horiz. synchronization signal in characters	10	10	10
4	Vert. characters seeded	31	31	127
5	Vert. characters justified	6	6	6
6	Vert. characters displayed	25	25	100
7	Vert. synchronization signal to ... characters	28	28	112
8	Interlace mode	2	2	2
9	Number of scan-lines per line	7	7	1
10	Starting line of blinking cursor	6	6	6
11	Ending line of blinking cursor	7	7	7
12	Display page starting address (high byte)	0	0	0
13	Display page starting address (low byte)	0	0	0
14	Cusrsor character address (high byte)	0	0	0
15	Cursor character address (low byte)	0	0	0
16	Reserved			
17	Reserved			

These registers are of interest to the user since they define the position and appearance of the cursor on the screen. Section 10.1 described programming these registers. The CGA adds registers 12 and 13. They indicate the start of the video page which must be displayed on the screen, as offset of the beginning of the 16K RAM on the card (B800:0000), divided by 2. Register 12 contains the most significant 8 bits of this offset, while register 13 contains the least significant 8 bits. Normally both registers contain the value 0, displaying the first screen page (beginning at the address B800:0000) on the screen. For display of the first screen page, which begins at location B800:1000 in the 80x25 text mode, the value 1000H divided by 2 (800H) must be entered in both registers.

The last of the three programs in this chapter accesses the color/graphics adapter. The only significant difference between the two preceding programs lies in the fact that the video controller can synchronize video RAM access and screen construction. This is necessary on all video cards where direct access to video RAM causes a flickering on the screen. The WAIT constant, defined directly after the program header, switches synchronization on or off. Its contents decide during the assembly of the program, whether to assemble the program lines for synchronization listed in the source listing. These lines would slow down the screen considerably, and should only be included if it is absolutely necessary.

Assembler listing: VCOL.ASM

```
;**********************************************************************;
;*                           V C O L                                 *;
;*------------      ---------------------------------------------------*;
;*    Task          : Makes some basic functions available for        *;
;*                    access to the Color Graphics Adapter (CGA)      *;
;*--------------------------------------------------------------------*;
;*    Info          : All functions subdivide the screen              *;
;*                    into  columns 0 to 79 and lines 0 to 24         *;
;*                    in text mode and into  columns 0 to 719 and     *;
;*                    the lines 0 to 347 in graphic mode.             *;
;*                    the 40 column text mode is not supported !      *;
;*                    A high resolution graphic screen should appear*;
;*                    first, followed by a text screen. If the high *;
;*                    res screen doesn't appear, try running the      *;
;*                    program a few times in succession.              *;
;*--------------------------------------------------------------------*;
;*    Author        : MICHAEL TISCHER                                 *;
;*    Developed on  : 8/13/87                                         *;
;*    Last update   : 6/16/89                                         *;
;*--------------------------------------------------------------------*;
;*    assembly      : MASM VCOL (program will assemble with one       *;
;*                              warning - it WILL link & run)         *;
;*                    LINK VCOL;                                      *;
;*--------------------------------------------------------------------*;
;*    Call          : VCOL                                            *;
;**********************************************************************;

;== Constants =========================================================

CONTROL_REG  = 03D8h              ;Control register port address
CCHOICE_REG  = 03D9h              ;Color select register port address
ADDRESS_6845 = 03D4h              ;6845 address register
DATA_6845    = 03D5h              ;6845 data register
VIO_SEG      = 0B800h             ;Video RAM segment address
CUR_START    = 10                 ;Reg # for CRTC: Cursor start line
CUR_END      = 11                 ;Reg # for CTRC: Cursor end line
CURPG_HI     = 12                 ;Page address (high byte)
CURPG_LO     = 13                 ;Page address (low byte)
CURPOS_HI    = 14                 ;Reg # for CRTC: Cursor pos high byte
CURPOS_LO    = 15                 ;Reg # for CRTC: Cursor pos low byte
DELAY        = 20000              ;Counter for delay loop

;== Macros ============================================================

;-- SETMODE : Macro for configuring screen control register ---------

setmode  macro modus

         mov  dx,CONTROL_REG       ;Address of the display control register
         mov  al,modus             ;New mode into the AL register
         out  dx,al                ;Send mode to control register

         endm

;-- WAITRET: waits until display is completed  -----------------------

waitret  macro
local    wrl                       ;Local label

         mov  dx,3DAh              ;Address of the display status register
wrl:     in   al,dx                ;Get content
```

```
local     wrl                     ;Local label

          mov  dx,3DAh            ;Address of the display status register
wrl:      in   al,dx              ;Get content
          test al,8               ;Vertical retrace?
          je   wrl                ;NO --> wait

          endm

;== Stack ============================================================

stack     segment para stack      ;Definition of stack segment

          dw 256 dup (?)          ;256-word stack

stack     ends                    ;End of stack segment

;== Data =============================================================

data      segment para 'DATA'     ;Definition of data segment

;== Data required for demo program ===================================

initm    db 13,10
         db "VCOL (c) 1988,1989 by Michael Tischer "
         db 13,10,13,10
         db "This demo program only runs with a Color/Graphics",13,10
         db "Adapter ( CGA ).  If your PC uses another type of",13,10
         db "video card press the <s> key to stop the program.",13,10
         db "Press any other key to start the program...",13,10,"$"

strl      db 1,0

;== Table of offset addresses of line beginnings =====================
lines    dw  0*160, 1*160, 2*160 ;start addresses of the lines as
         dw  3*160, 4*160, 5*160 ;offset addresses in the video RAM
         dw  6*160, 7*160, 8*160
         dw  9*160,10*160,11*160,12*160,13*160,14*160,15*160,16*160
         dw 17*160,18*160,19*160,20*160,21*160,22*160,23*160,24*160

graphict  db 38h, 28h, 2Dh, 0Ah, 7Fh, 06h  ;register values for the
          db 64h, 70h, 02h, 01h, 06h, 07h  ;graphic-modes

textt     db 71h, 50h, 5Ah, 0Ah, 1Fh, 06h  ;register-values for the
          db 19h, 1Ch, 02h, 07h, 06h, 07h  ;graphic-modes

wait      db 0                    ;TRUE (<>0) when caller uses the
                                  ;/F switch

data      ends                    ;End of data segment

;== Code =========================================================

code      segment para 'CODE'     ;Definition of the CODE segment

          assume cs:code, ds:data, es:data, ss:stack

;== This is only the Demo-Program ====================================

demo      proc far

          ;-- Look for /F from DOS prompt ------------------------

          mov  cl,ds:128          ;Get number of bytes from prompt
          or   cl,cl              ;No parameters given?
          je   switch1            ;NO --> Ignore
          mov  bx,129             ;BX points to first byte in prompt
          mov  ch,bh              ;Set loop high byte to 0

switch:   cmp  [bx],"F/"          ;Switch in this position?
```

```
             je   switch1              ;YES --> Switch found
             cmp  [bx],"f/"            ;Switch in this position?
             je   switch1              ;YES --> Switch found
             inc  bl                   ;Set BX to next character
             loop switch               ;Check next character

switch1:     mov  ax,data              ;Get segment addr. of data segment
             mov  ds,ax                ;and load into DS
             mov es,ax                 ;and ES

             mov  wait,cl              ;Set WAIT flag

             ;-- Display init message and wait for input ------------

             mov  ah,9                 ;Function number for string display
             mov  dx,offset initm      ;Address of intial message
             int  21h                  ;Call DOS interrupt 21H

             xor  ah,ah                ;Function number: get key
             int  16h                  ;Call BIOS keyboard interrupt
             cmp  al,"s"               ;<s> key pressed?
             je   ende                 ;YES --> End program
             cmp  al,"S"               ;<S> key pressed?
             jne  startdemo            ;NO --> Start demo

ende:        mov  ax,4C00h             ;Function number: End program
             int  21h                  ;Call DOS interrupt 21H

startdemo label near
             call grafhi               ;switch on 320*200 pixel graphic
             xor  al,al
             call cgr                  ;Clear graphic display

             xor  bx,bx                ;Column 0
             xor  dx,dx                ;Line 0
             mov  ax,199               ;Pixels-vertical
             mov  cx,639               ;Pixels-horizontal
gr1:         push cx                   ;Record horizontal pixels
             mov  cx,ax                ;Vertical pixels to counter
             push ax                   ;Record vertical pixels on the stack
             mov  al,1
gr2:         call pixhi                ;Set pixel
             inc  dx                   ;Increment line
             loop gr2                  ;Draw line
             pop  ax                   ;Get vertical pixels from the stack
             sub  ax,3                 ;Next line 3 pixels less
             pop  cx                   ;Get horizontal pixels from the stack
             push cx                   ;Record horizontal pixels
             push ax                   ;Record vertical pixels on the stack
             mov  al,1
gr3:         call pixhi                ;Set pixel
             inc  bx                   ;Increment column
             loop gr3                  ;Draw line
             pop  ax                   ;Get vertical pixels from stack
             pop  cx                   ;Get horizontal pixels from stack
             sub  cx,6                 ;Next line 6 pixels less
             push cx                   ;Record horizontal pixels
             mov  cx,ax                ;Vertical pixels to counter
             push ax                   ;Record vertical pixels on the stack
             mov  al,1
gr4:         call pixhi                ;Set pixel
             dec  dx                   ;Decrement line
             loop gr4                  ;Draw line
             pop  ax                   ;Get vertical pixels from stack
             sub  ax,3                 ;Next line 3 pixels less
             pop  cx                   ;Get horizontal pixels from stack
             push cx                   ;Record horizontal pixels
             push ax                   ;Record vertical pixels on the stack
             mov  al,1
gr5:         call pixhi                ;Set pixel
```

```
        dec  bx                 ;Increment column
        loop gr5                ;Draw line
        pop  ax                 ;Get vertical pixels from the stack
        pop  cx                 ;Get horizontal pixels from the stack
        sub  cx,6               ;Next line 6 pixels less
        cmp  ax,5               ;Is the vertical line longer than 5
        ja   gr1                ;YES--> continue

        xor  ah,ah              ;Wait for function number of key wait
        int  16h                ;Call BIOS keyboard interrupt

        call text               ;Switch on 80x25 character text mode
        xor  bp,bp              ;Process screen page 0 first
demo1:  mov  al,30h             ;ASCII code "0"
        or   ax,bp              ;Convert page number to ASCII
        mov  str1,al            ;Store in string
        call setcol             ;Set color
        call setpage            ;Activate screen page in BP
        call cls                ;Clear screen page
        xor  bx,bx              ;Begin in the upper left
        call calo               ;Screen corner with output
        mov  cx,2000            ;A page contains 2,000 characters
        xor  ah,ah              ;Start with color code 0
        mov  si,offset str1     ;Offset address of string 1
demo2:  inc  ah                 ;Increment color value
        call print              ;Output string 1
        loop demo2              ;Repeat until screen is full

        xor  ah,ah              ;Wait for key
        int  16h                ;Call BIOS-Keyboard-Interrupt
        inc  bp                 ;Increment page number
        cmp  bp,4               ;All 4 pages processed ?
        jne  demo1              ;NO --> then next page

        xor  bp,bp              ;Activate page 0 again
        call setpage
        jmp  ende
demo    endp                    ;Goto program end

;== The actual functions follow ==========================

;-- TEXT: switches the text display on -------------------------------
;-- Input     : none
;-- Output    : none
;-- Register : AX, SI, BH, DX and FLAGS are changed

text    proc near

        mov  si,offset textt    ;Offset address of the register-table
        mov  bl,00100001b       ;80x25 text mode,blinking
        jmp  short vcprog       ;Program video controller again

text    endp

;-- GRAFHI: switches the 640*200 pixel graphic mode on -----------------
;-- Input    : none
;-- Output   : none
;-- Register : AX, SI, BH, DX and FLAGS are changed

grafhi  proc near

        mov  bl,00010010b        ;Graphic mode with 640*200 pixels
        jmp  short graphic       ;Program video controller again

grafhi  endp

;-- GRAFLO: switches the 320*200 pixel graphic mode on -----------------
;-- Input    : none
;-- Output   : none
;-- Register : AX, SI, BH, DX and FLAGS are changed
```

```
grafl0   proc near

         mov  bl,00100010b         ;Graphic mode with 320*200 pixels
graphic:  mov  si,offset graphict  ;Offset address of the register table

graflo   endp

;-- VCPROG: programs the video controller ----------------------------
;-- Input    : SI = Address of a register table
;--            BL = Value for display control register
;-- Output   : none
;-- Register : AX, SI, BH, DX and FLAGS are changed

vcprog   proc near

         setmode bl               ;Bit 3 = 0: screen off

         mov  cx,12               ;12 registers are set
         xor  bh,bh               ;Start with register 0
vcpl:    lodsb                    ;Get register value from table
         mov  ah,al               ;Register value to AH
         mov  al,bh               ;Number of the register to AL
         call setvk               ;Transmit value to controller
         inc  bh                  ;Address next register
         loop vcpl                ;Set additional registers

         or   bl,8                ;Bit 3 = 1: screen on
         setmode bl               ;Set new mode
         ret                      ;Back to caller

vcprog   endp

;-- SETCOL   : Sets the color of the display frame and Background -----
;-- Input    : AL = color value
;-- Output   : none
;-- register : AX and DX are changed
;-- Info     : in text mode the lowest 4 bits indicate the frame color
;--             in graphic mode the lowest 4 bits indicate the frame
;--             and background color, bit 5 selects the color palette

setcol   proc near

         mov  dx,CCHOICE_REG      ;Address of the color selection register
         out  dx,al               ;Output color value
         ret                      ;Back to caller

setcol   endp

;-- CDEF     : sets the start  and end line of the cursor -------------
;-- Input    : CL = start line
;--            CH = end line
;-- Output   : none
;-- register : AX and DX are changed

cdef     proc near

         mov  al,CUR_START        ;Register 10: start line
         mov  ah,cl               ;Start line to AH
         call setvk               ;Transmit to video controller
         mov  al,CUR_END          ;Register 11: end line
         mov  ah,ch               ;End line to AH
         jmp  short setvk         ;Transmit to video controller

cdef     endp

;-- SETPAGE  : sets the screen page ----------------------------------
;-- Input    : BP = Number of the screen page (0 to 3)
;-- Output   : none
;-- register : BX, AX, CX and DX are changed
```

```
;-- Info     : in the Graphic modes the first screen page has the
;--            number 0, the second the number 2

setpage     proc near

         mov  bx,bp              ;Screen page to BX
         mov  cl,5               ;Multiply by 2,048
         ror  bx,cl
         mov  al,CURPG_HI        ;Register 12: Hi byte page address
         mov  ah,bh              ;Hi byte of the screen page to AH
         call setvk              ;Transmit to video controller
         mov  al,CURPG_LO        ;Register 13: Lo byte page address
         mov  ah,bl              ;Lo byte of the screen page to AH
         jmp  short setvk        ;Transmit to video controller

setpage     endp

;-- SETBLINK : sets the blinking cursor ------------------------------
;-- Input    : DI = Offset address of the cursor
;-- Output   : none
;-- register : BX, AX and DX are changed

setblink proc near

         mov  bx,di            ;Move offset to BX
         mov  al,CURPOS_HI     ;Hi byte of the cursor offset
         mov  ah,bh            ;HI byte of the offset
         call setvk            ;Transmit to video controller
         mov  al,CURPOS_LO     ;Lo byte of the cursor offset
         mov  ah,bl            ;Lo byte of the offset

         ;-- SETVK is called automatically --------------------------

setblink endp

;-- SETVK    : sets a byte in one register of the video controller ----
;-- Input    : AL = Number of the register
;--            AH = new content of the register
;-- Output   : none
;-- register : DX and AL are changed

setvk    proc near

         mov  dx,ADDRESS_6845    ;Address of the index register
         out  dx,al              ;Send number of the register
         jmp  short $+2          ;Short I/O pause
         inc  dx                 ;Address of the index register
         mov  al,ah              ;Content to AL
         out  dx,al              ;Set new content
         ret                     ;Back to caller

setvk    endp

;-- GETVK    : gets a byte from one register of the video controller -
;-- Input    : AL = Number of the register
;-- Output   : AL = Contents of register
;-- register : DX and AL are changed

getvk    proc near

         mov  dx,ADDRESS_6845    ;Address of the index register
         out  dx,al              ;Send number of the register
         inc  dx                 ;Index register address
         jmp  short $+2          ;Short io pause
         in   al,dx              ;Set new contents
         ret                     ;Back to caller

getvk    endp

;-- SCROLLUP: scrolls a window N lines upward ------------------------
```

```
;-- Input    : BL = line upper left
;--            BH = column upper left
;--            DL = line below right
;--            DH = column below right
;--            CL = Number of lines, to be scrolled
;--          : BP = Number of the screen page (0 to 3)
;-- Output   : none
;-- register : only FLAGS are changed
;-- Info     : the display lines liberated are cleared

scrollup proc near

         cld                    ;On string commands count up

         push ax                ;All changed registers to the
         push bx                ;Secure stack
         push di                ;In this case the sequence
         push si                ;must be observed !

         push bx                ;These three registers are returned
         push cx                ;before the end of the routine
         push dx                ;From the stack
         sub  dl,bl             ;Calculate the number of lines
         inc  dl
         sub  dl,cl             ;Subtract number of lines to be scrolled
         sub  bh,dh             ;Calculate number of columns
         inc  dh
         call calo              ;Convert upper left in offset
         mov  si,di             ;Record address in SI
         add  bl,cl             ;First line in scrolled window
         call calo              ;Convert first line in offset
         xchg si,di             ;Exchange SI and DI

         cmp  wait,0            ;Flicker suppressed?
         je   sup0              ;NO --> SUP0

         waitret                ;YES -->Wait for retrace
         setmode 00100101b      ;Disable screen

sup0:    push ds                ;Store segment register
         push es                ;On the stack
         mov  ax,VIO_SEG        ;Segment address of the video RAM
         mov  ds,ax             ;To DS
         mov  es,ax             ;And ES

sup1:    mov  ax,di             ;Record DI in AX
         mov  bx,si             ;Record SI in BX
         mov  cl,dh             ;Number of columns in counter
         rep movsw              ;Move a line
         mov  di,ax             ;Restore DI from AX
         mov  si,bx             ;Restore SI from BX
         add  di,160            ;Set next line
         add  si,160
         dec  dl                ;processed all lines ?
         jne  sup1              ;NO --> move another line

         pop  es                ;Get segment register from
         pop  ds                ;Stack

         cmp  wait,0            ;Flickering suppressed?
         je   sup2              ;NO --> SUP2

         setmode 00101101b      ;YES --> Enable screen

sup2:    pop  dx                ;Get lower right corner back
         pop  cx                ;Return number of lines
         pop  bx                ;Return upper left corner
         mov  bl,dl             ;Lower line to BL
         sub  bl,cl             ;Subtract number of lines
         inc  bl
```

```
          mov  ah,07h           ;Color : black on white
          call clear            ;Clear lines

          pop  si               ;CX and DX have already been
          pop  di               ;Restored
          pop  bx
          pop  ax

          ret                   ;Back to caller

scrollup  endp

;-- SCROLLDN: scrolls a window N lines down --------------------------
;-- Input    : BL = line upper left
;--            BH = column upper left
;--            DL = line below right
;--            DH = column below right
;--            CL = number of lines to be scrolled
;--          : BP = number of the screen page (0 to 3)
;-- Output   : none
;-- register : only FLAGS are changed
;-- Info     : the display lines liberated are cleared

scrolldn  proc near

          cld                   ;On string commands count up

          push ax               ;Record all changed registers
          push bx               ;On the stack
          push di               ;In this case the sequence
          push si               ;Must be observed !

          push bx               ;These three registers are returned
          push cx               ;From the stack before the end
          push dx               ;Of the routine

          sub  dh,bh            ;Calculate the number of columns
          inc  dh
          mov  al,bl            ;Record line upper left in AL
          mov  bl,dl            ;Line below right to line below left
          call calo             ;Convert upper left in offset
          mov  si,di            ;Record address in SI
          sub  bl,cl            ;Subtract number of characters to scroll
          call calo             ;Convert upper left in offset
          xchg si,di            ;Exchange SI and DI
          sub  dl,al            ;Calculate number of lines
          inc  dl
          sub  dl,cl            ;Subtract number of lines to be scrolled

          cmp  wait,0           ;Flicker suppressed?
          je   sdn0             ;NO --> SDNO

          waitret               ;YES --> Wait for retrace
          setmode 00100101b     ;Disable screen

sdn0:     push ds               ;Store segment register on the
          push es               ;Stack
          mov  ax,VIO_SEG       ;Segment address of the video RAM
          mov  ds,ax            ;To DS
          mov  es,ax            ;and ES

sdn1:     mov  ax,di            ;Record DI in AX
          mov  bx,si            ;Record SI in BX
          mov  cl,dh            ;Number of columns in counter
          rep movsw             ;Move a line
          mov  di,ax            ;Restore DI from AX
          mov  si,bx            ;Restore SI from BX
          sub  di,160           ;Set into next line
          sub  si,160
          dec  dl               ;processed all lines ?
```

```
          jne  sdnl                  ;NO --> move another line

          pop  es                    ;Return segment register from
          pop  ds                    ;Stack

          cmp  wait,0                ;Flicker suppressed?
          je   sdn2                  ;NO --> SDN2

          setmode 00101101b          ;YES --> Enable screen


sdn2:     pop  dx                    ;Get lower right corner
          pop  cx                    ;Return number of lines
          pop  bx                    ;Return upper left corner
          mov  dl,bl                 ;upper line to DL
          add  dl,cl                 ;Add number of lines
          dec  dl
          mov  ah,07h                ;Color : black on white
          call clear                 ;Erase liberated lines

          pop  si                    ;CX and DX have already been
          pop  di                    ;Returned
          pop  bx
          pop  ax

          ret                        ;Back to caller

scrolldn  endp

;-- CLS: Clear the screen completely ---------------------------------
;-- Input   : BP = number of the screen page (0 or 1)
;-- Output   : none
;-- register : only FLAGS are changed

cls       proc near

          mov  ah,07h                ;Color is white on black
          xor  bx,bx                 ;upper left is (0/0)
          mov  dx,4F18h              ;Lower right is (79/24)

          ;-- Execute Clear -----------------------------------------

cls       endp

;-- CLEAR: fills a designated display area with space characters ------
;-- Input      : AH = attribute/color
;--              BL = line upper left
;--              BH = column upper left
;--              DL = line below right
;--              DH = column below right
;--              BP = number of the screen page (0 to 3)
;-- Output    : none
;-- register : only FLAGS are changed

clear     proc near

          cld                        ;On string commands count up
          push cx                    ;Store all register which are
          push dx                    ;Changed on the stack
          push si
          push di
          push es
          sub  dl,bl                 ;Calculate number of lines
          inc  dl
          sub  dh,bh                 ;Calculate number of columns
          inc  dh
          call calo                  ;Offset address of the upper left corner
          mov  cx,VIO_SEG            ;Segment address of the video RAM
          mov  es,cx                 ;To ES
          xor  ch,ch                 ;Hi bytes of the counter to 0
```

```
         mov  al," "              ;Space character

         cmp  wait,0              ;Flickering suppressed?
         je   clear1              ;NO --> CLEAR1

         push dx                  ;Store DX on the stack
         waitret                  ;Retrace wait
         setmode 00100101b        ;Switch screen off
         pop  dx                  ;Return DX from the stack

clear1:  mov  si,di               ;Record DI in SI
         mov  cl,dh               ;Number columns in counter
         rep stosw                ;Store space character
         mov  di,si               ;Return DI from SI
         add  di,160              ;Set in next line
         dec  dl                  ;All lines processed ?
         jne  clear1              ;NO --> erase another line

         cmp  wait,0              ;Flicker suppressed?
         je   clear2              ;NO --> CLEAR2

         setmode 00101101b        ;Enable screen

clear2:  pop  es                  ;Get registers from
         pop  di                  ;Stack again
         pop  si
         pop  dx
         pop  cx
         ret                      ;Back to caller

clear    endp

;-- PRINT: outputs a string on the screen ----------------------------
;-- Input   :  AH = attribute/color
;--            DI = offset address of the first character
;--            SI = offset address of the strings to DS
;--            BP = number of the screen page (0 to 3)
;-- Output   : DI points behind the last character output
;-- register : AL, DI and FLAGS are changed
;-- Info     : the string must be terminated by a NUL-character.
;--            other control characters are not recognized

print    proc near

         cld                      ;On string commands count up
         push si                  ;Store SI, DX and ES on the stack
         push es
         push cx
         push dx
         mov  dx,VIO_SEG          ;Segment address of the video RAM
         mov  cl,wait             ;Get WAIT flag
         mov  es,dx               ;First to DX and then to ES

         jmp  short print3        ;Get character and display it

print1   label near

         or   cl,cl               ;Flicker suppressed?
         je   print2              ;NO --> PRINT2

         push ax                  ;Record characters and color
         mov  dx,3DAh             ;Address of the display-status-register
hr1:     in   al,dx               ;Get content
         test al,1                ;Horizontal retrace?
         jne  hr1                 ;NO --> wait
         cli                      ;permit no further interrupts
hr2:     in   al,dx               ;Get content
         test al,1                ;Horizontal retrace?
         je   hr2                 ;YES --> wait
         pop  ax                  ;Restore characters and color
```

```
         sti                        ;Do not suppress Interrupts any more

print2:  stosw                      ;Store attribute and color in V-RAM
print3:  lodsb                      ;Get next character from the string
         or   al,al                 ;Is it NUL
         jne  print1                ;NO --> output

printe:  pop  dx                    ;Get SI, DX, CX and ES from stack
         pop  cx
         pop  es
         pop  si
         ret                        ;Back to caller

print    endp

;-- CALO: Converts line and column into offset address ----------------
;-- Input    : BL = line
;--            BH = column
;--            BP = number of the screen page (0 to 3)
;-- Output   : DI = the offset address
;-- register : DI and FLAGS are changed

calo     proc near

         push ax                    ;Secure AX on the stack
         push bx                    ;Secure BX on the stack

         shl  bx,1                  ;Column and line times 2
         mov  al,bh                 ;Column to AL
         xor  bh,bh                 ;Hi byte
         mov  di,[lines+bx]         ;Get offset address of the line
         xor  ah,ah                 ;HI byte for column offset
         add  di,ax                 ;Add line and column offset
         mov  bx,bp                 ;Screen page to BX
         mov  cl,4                  ;Multiply by 4,096
         ror  bx,cl
         add  di,bx                 ;Add beginning of screen page to offset
         pop  bx                    ;Restore BX from stack
         pop  ax                    ;Restore AX from stack
         ret                        ;Back to caller

calo     endp

;-- CGR: Erase the complete Graphic display ---------------------------
;-- Input   : AL = 00H : erase all pixels
;--                FFH : set all pixels
;-- Output   : none
;-- register : AH, BX, CX, DI and FLAGS are changed
;-- Info     : this Function erases the Graphic display in both
;--            Graphic modes

cgr      proc near

         push es                    ;Store ES on the stack
         cbw                        ;Expand AL to AH
         xor  di,di                 ;Offset address in video RAM
         mov  bx,VIO_SEG            ;Segment address screen page
         mov  es,bx                 ;Segment address into segment register
         mov  cx,2000h              ;One page is 8KB words
         rep stosw                  ;Fill page
         pop  es                    ;Return ES from stack
         ret                        ;Back to caller

cgr      endp

;-- PIXLO: sets a pixel in the 320*200 pixel graphic mode ----------------
;-- Input  :   BP = number of the screen page (0 or 1)
; -            BX = column (0 to 319)
;--            DX = line   (0 to 199)
;--            AL = color of the pixels (0 to 3)
```

```
;-- Output   : none
;-- register : AX, DI and FLAGS are changed

pixlo     proc near

          push ax                  ;Secure AX on the stack
          push bx                  ;Note BX on the stack
          push cx                  ;Store CX on the stack
          mov  cl,7
          mov  ah,bl               ;Transmit column to AH
          and  ah,11b              ;Column mod 4
          shl  ah,1                ;Column * 2
          sub  cl,ah               ;7 - 2 * (column mod 4)
          mov  ah,11               ;Bit value
          shl  ax,cl               ;Move to pixel position
          not  ah                  ;Reverse AH
          shr  bx,1                ;Divide BX by 4 by shifting
          shr  bx,1                ;Right twice
          jmp  short spix          ;Set pixel

pixlo     endp

;-- PIXHI: sets a pixel in the 640*200 pixel graphic mode -----------------
;-- Input  :   BP = number of the screen page (0 or 1)
;--            BX = column (0 to 639)
;--            DX = line   (0 to 199)
;--            AL = color of the pixels (0 or 1)
;-- Output   : none
;-- register : AX, DI and FLAGS are changed

pixhi     proc near

          push ax                  ;Store AX on the stack
          push bx                  ;Note BX on the stack
          push cx                  ;Note CX on the stack
          mov  cl,7
          mov  ah,bl               ;Transmit column to AH
          and  ah,111b             ;Column mod 8
          sub  cl,ah               ;7 - column mod 8
          mov  ah,1                ;Bit value
          shl  ax,cl               ;Move pixel position
          not  ah                  ;Reverse AH
          mov  cl,3                ;3 shifts
          shr  bx,cl               ;Divide BX by 8

          ;-- set pixel ------------------------------------------------

pixhi          endp

;-- SPIX: sets a pixel in the graphic display ---------------------------
;-- Input     : BX = column offset
;--             DX = line   (0 to 199)
;--             AH = Value to cancel old Bits
;--             AL = new Bit value
;-- Output    : none
;-- register : AX, DI and FLAGS are changed

spix      proc near

          push es                  ;Secure ES on the stack
          push dx                  ;Secure DX on the stack
          push ax                  ;Secure AX on the stack

          xor  di,di               ;Offset address in video RAM
          mov  cx,VIO_SEG          ;Segment address screen page
          mov  es,cx               ;Segment address into segment register
          mov  ax,dx               ;Move line to AX
          shr  ax,1                ;Divide line by 2
          mov  cl,80               ;The factor is 90
          mul  cl                  ;Multiply line by 80
```

```
        and  dx,1                ;Line mod 2
        mov  cl,3                ;3 shifts
        ror  dx,cl               ;Rotate right (* 2000H)
        mov  di,ax               ;80 * int(line/2)
        add  di,dx               ;+ 2000H * (line mod 4)
        add  di,bx               ;Add column offset
        pop  ax                  ;Return AX from stack
        mov  bl,es:[di]          ;Get pixel
        and  bl,ah               ;Erase Bits
        or   bl,al               ;Add pixel
        mov  es:[di],bl          ;write pixel back

        pop  dx                  ;Return DX from stack
        pop  es                  ;Return ES from stack
        pop  cx                  ;Return CX from stack
        pop  bx                  ;Return BX from stack
        pop  ax                  ;Return AX from stack

        ret                      ;Back to caller

spix    endp

;== end ==============================================================

code    ends                     ;End of the code segment
        end  demo
```

10.5 EGA and VGA Cards

The EGA and VGA cards far exceed their predecessors in both graphics and in text display capabilities. Other computers have had EGA and VGA capabilities for some time (e.g., work stations, CAD/CAM applications), but these video cards are now at prices where many home systems will soon have them.

The range of power of this new generation of video cards can be seen in their very sharp resolutions and their ability to display almost any number of lines on the screen. The EGA and VGA cards' greatest feature lies in their ability to emulate other video cards.

These capabilities come with a price—more complicated hardware and programming are required. One result of this is that the features of an EGA card or a VGA card can no longer be realized with the traditional PC video controller (the Motorola 6845). Instead, most EGA and VGA cards contain a VLSI chip developed especially for use on an EGA card. At the heart of this component is a video controller that controls the video signal generation. Its basic task is similar to that of the 6845, but its registers differ from those of the 6845, both in number and interaction between registers. Comparing the 6845 and VSLI is like comparing BASIC and assembly language, where the increase of power is in proportion to the degree of language complexity.

We recommend that you avoid programming the hardware registers directly unless you absolutely must do so. Many tasks can be delegated to the BIOS without wasting much time. Not only will this keep your program code more compact and easier to read, it will greatly improve the compatibility of your code with other video cards. Among the tasks which the various functions of the BIOS video interrupt can perform are:

- Initialization of the video mode
- Selection of the display page
- Cursor positioning
- Defining the starting and ending line of the cursor
- Palette and border color selection
- Setting the size of the character matrix, and thereby the number of text lines which can be displayed on the screen
- Loading user-defined character sets
- Reading configuration data

Detailed information about traditional BIOS video functions and the new functions of the EGA/VGA BIOS can be found in Sections 7.4.

If you need speed and maximum control over the screen, you should still perform time-critical actions (e.g., manipulating video RAM) "by hand."

EGA/VGA and text mode

There is no difference between the EGA and MDA or CGA card in text mode. The video RAM and attribute byte are organized the same way for the EGA card as for the other two cards—even the location of the video RAM is the same. But since an EGA card can emulate either a CGA card or an MDA card, depending on the monitor to which it is connected, you should first determine what kind monitor is in use. From this the EGA can determine which of the two systems to emulate (routines presented in Section 10.7 show how this is done). The type of card being emulated determines where the video RAM can be found in memory, how the bits of the character attribute byte are interpreted, and how many screen pages are available.

Remember that the EGA or VGA card does not contain a 6845 CRTC, despite the fact that it can perfectly emulate its video predecessors. This means that the status and control registers of the MDA and CGA cards are unavailable. However, since the settings that are normally made with these registers can also be performed with the BIOS, we don't really need these registers. You should also remember that there are no restrictions to accessing the video RAM of an EGA card or a VGA card when it is in CGA emulation. It is unnecessary to synchronize screen access with the activity of the CRTC by reading the status register.

The parallels between the organization of the video RAM in the CGA and MDA cards also apply when the text mode is switched to 43 lines (which is impossible in CGA emulation). As with any other number of displayed lines, this does not change the basic structure of the video RAM at all. It is larger, but the formulas for calculating the offset position of a character and its attribute byte within the video RAM are still valid.

The VGA card is capable of 25, 43 and even 50 lines in text mode, depending on the monitor in use.

These parallels also apply to the graphics modes already available to the CGA card. The position of the video RAM and its structure are identical to the those of the CGA card.

EGA/VGA and graphic modes

The EGA card offers the following new graphics modes:

- 320x200 pixels, 16 colors (BIOS code: 0DH)
- 640x200 pixels, 16 colors (BIOS code: 0EH)
- 640x350 pixels, 2 colors (BIOS code: 0FH)

- 640x350 pixels, 16 colors (BIOS code: 10H)

The VGA card offers the following graphic modes:

- 640x480 pixels, 2 colors (BIOS code: 11H)
- 640x480 pixels, 16 colors (BIOS code: 12H)
- 320x200 pixels, 256 colors (BIOS code: 13H)

Some EGA cards have even more modes with higher resolution or more colors, but these modes are not part of the EGA standard and are supported by only a few programs.

It is somewhat difficult to talk about a "standard", because almost every manufacturer has their own modes. Let's look at the lowest common denominator—the modes which practically all EGA/VGA cards support. These are the modes supported by the original EGA card, the IBM EGA.

These video modes, in which the video RAM can occupy more than 100K, show a structure quite different from those used by the MDA, CGA and Hercules cards. The maximum of 256K of RAM is divided into four *bitplanes* which are arranged in a kind of a three-dimensional organization. From the processor's point of view these bitplanes reside between segment addresses A000H and B000H.

Each bitplane contains one bit for each individual pixel. If you place the bitplanes on top of each other, each pixel is represented by a total of four bits, which together make up the color value of the pixel. Bitplane zero contains bit zero of the color value of each pixel, bitplane one contains bit one, and so on. This limits the number of displayable colors to 16, since four bits (or bitplanes) can represent 2^4, or 16 different numbers.

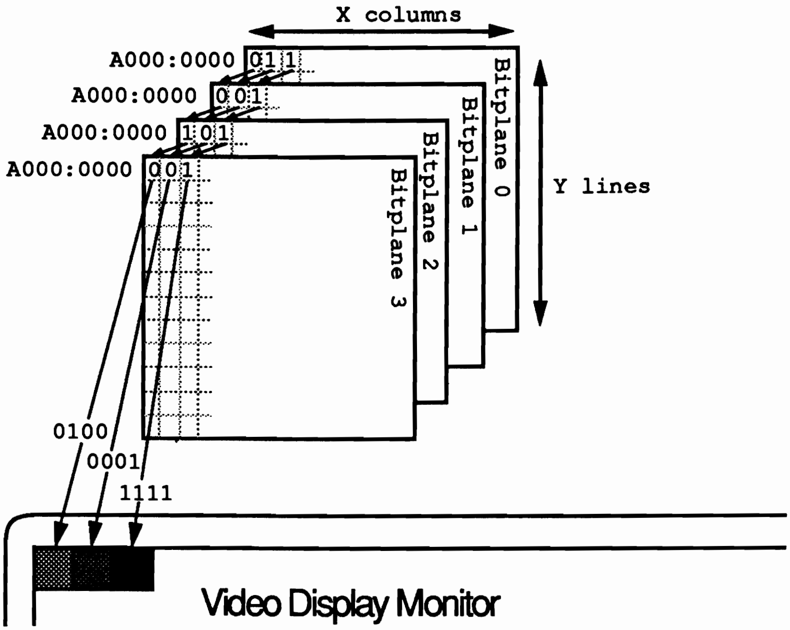
The color value obtained from combining individual bitplanes does not correspond directly to a color. It is actually used as an index into one of the 16 palette registers of the EGA card, each of which designates a particular color. Since the EGA card can display a total of 64 different colors, the palette registers allow you to select 16 of these colors to be displayed on the screen simultaneously. The individual palette registers can be loaded with the help of the extended EGA BIOS functions, as described in Section 7.4.

The structure of each bitplane corresponds to the organization of the pixels on the screen, and parallels that of video RAM in text mode. Since each pixel occupies one bit in the bitplane, eight consecutive pixels are combined into a byte. The pixels on each line are placed left to right in successive memory locations. The length of each line can be determined using the formula:

```
horizontal_resolution / 8
```

Since the individual screen lines follow each other in sequence starting from the top of the screen, the starting address of each line is obtained by multiplying the line number by this value. The byte within this line which contains the desired pixel is calculated by dividing the column number by eight (bits per byte). Adding this to the starting address of the line gives us the following formula, which calculates the offset address of the byte containing the coordinates (X, Y):

```
Y * (horizontal_resolution / 8) + X / 8
```



Bitplane arrangement on EGA card

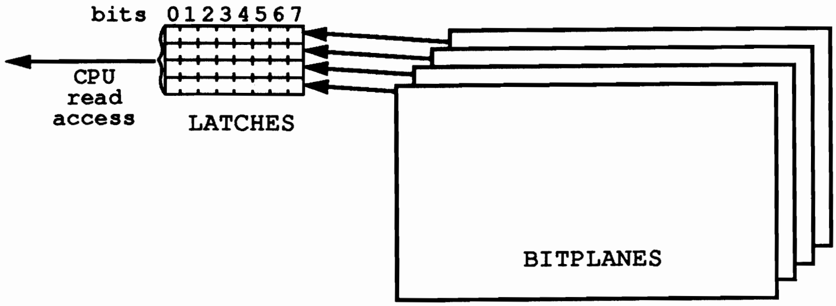
The bit number at which the pixel is located in this byte results from the remainder of the division of the column number by eight:

```
7 - (column_number MOD 8)
```

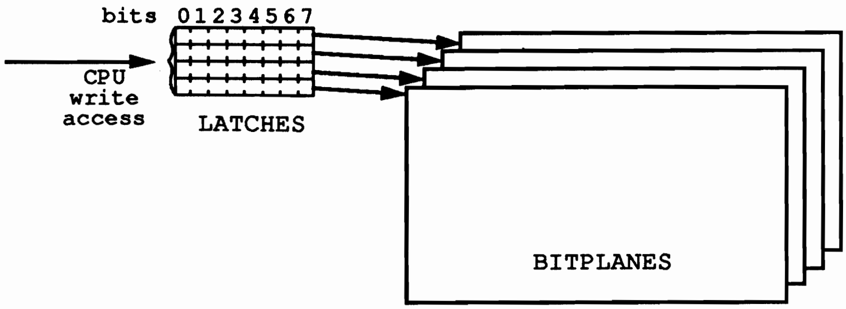
These two formulas can be used to localize a pixel within a bitplane and implement graphics primitives.

However, the bitplanes cannot be accessed individually because they all lie at the identical segment address. The EGA card has four latch registers, each of which contains a complete byte from one of the four bitplanes. When the CPU performs a read access from the EGA video RAM at segment address A000H, one byte is first read from each of the four bitplanes at the specified offset address and loaded into the four latch registers. This applies to instructions which access memory directly, such as MOV or LODS, as well as all instructions in which a byte from the video RAM appears as an operand. This can be the case with arithmetic instructions (ADD, SUB, OR, AND, etc.) and comparison instructions (CMP, CMPS).

The process is similar for writing bytes to the video RAM. In this situation the contents of the four latch registers are written back to the four bitplanes.



Video RAM access—loading the four latch registers



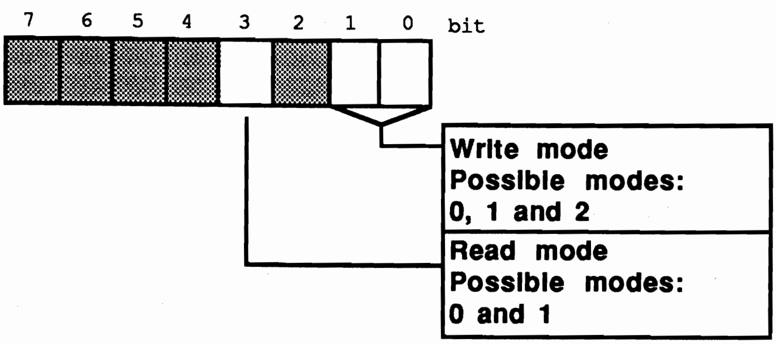
Video RAM access—writing the four latch registers

Since the latch registers are not directly accessible to the processor, we must alternate conversion between eight and 32 bits when reading and writing the video RAM. When reading, 32 bits from the latch registers must be compressed into one byte, while the eight bits from the CPU when writing must be divided among the 32 bits of the latch registers. The nine graphic controller registers in the EGA card perform this conversion.

EGA graphic controller registers and their default values		
Register	Meaning	Default
00H	Set / Reset	00H
01H	Enable Set / Reset	00H
02H	Color Compare	00H
03H	Function Select	00H
04H	Read Map Select	00H
05H	Mode	00H
06H	Miscellaneous	varies
07H	Color Don't Care	0FH
08H	Bit Mask	FFH

Access to these registers is similar to CRTC register access on the Hercules graphics card. Here too there is an address register at port address 3DEH, into which we must first load the number of the register in the graphics controller that we want to access. The value for this register can then be written to the data register located at address 3CFH, immediately after the address register. These ports do not have to be accessed separately: A 16-bit OUT instruction to the address register performs the access in one move. The AX register, which will be sent to this port, must contain the register number in the low-order byte (AL), and the value for this register in the high-order byte (AH). Although values can be loaded into the graphics controller registers in this manner, it is not possible to read data from the EGA card.

The contents of register number five, the mode register, are responsible for the behavior of the video RAM. This register controls the current read and write modes and thereby the manner in which the data from the latch registers is combined with the other registers in the graphics controller and the CPU data.

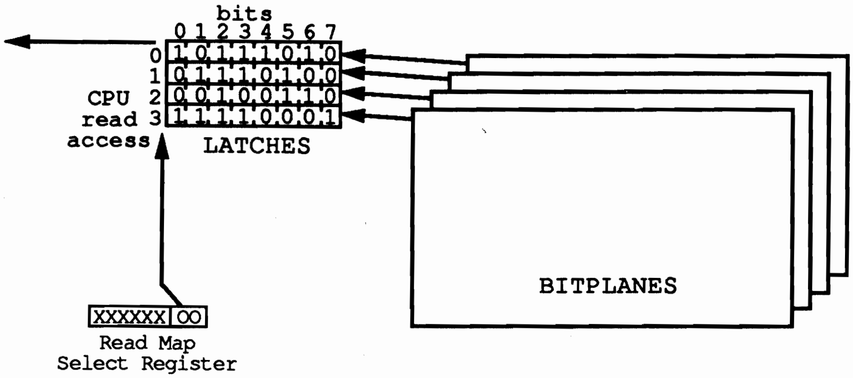


Mode register structure in EGA card graphics controller

There are a total of two different read modes and three write modes.

Read mode 0

Read mode 0 is the simpler of the two read modes. As usual, a read access in this mode first loads the specified byte from the four bitplanes into the four latch registers. Then the contents of the latch register specified by the lower two bits of the read map select register (register four) are transferred to the CPU.



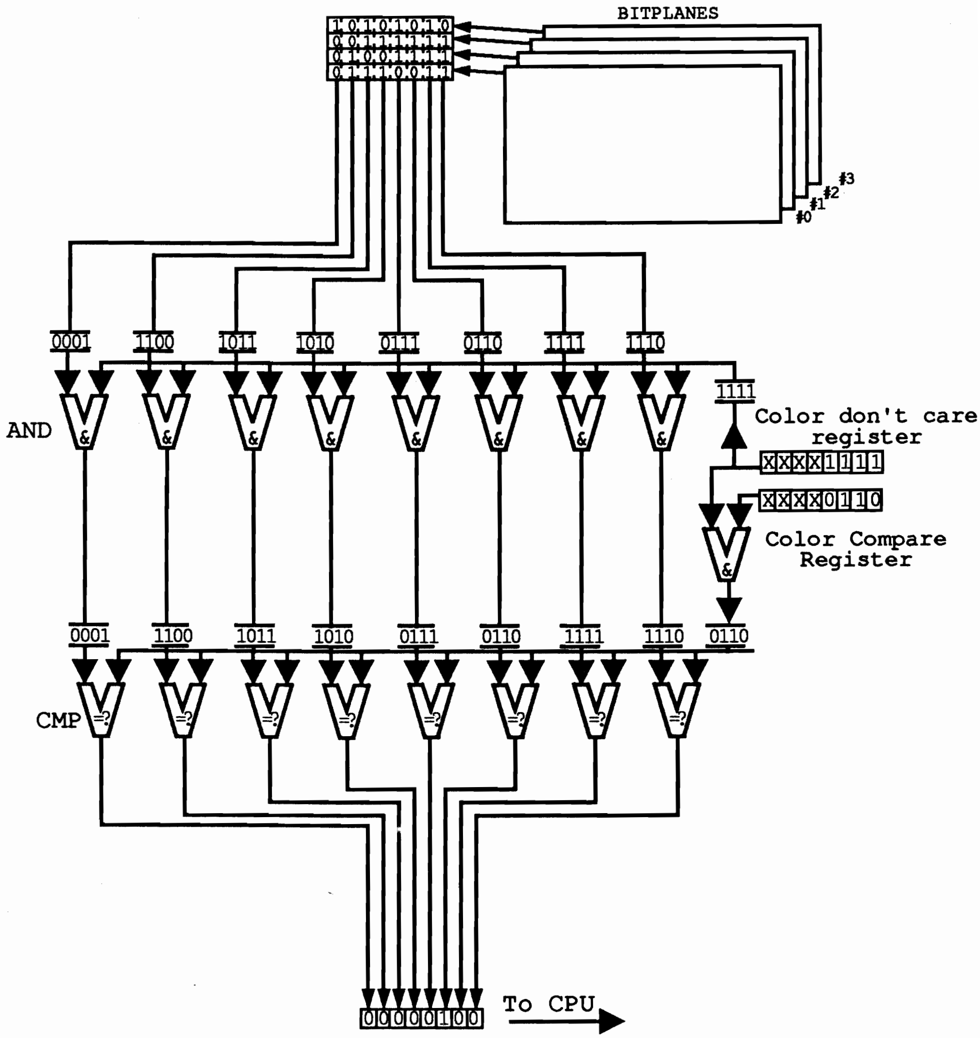
Video RAM read access in read mode 0

The following sequence of assembly language instructions first sets read mode 0, then writes the value 2 into the Read Map Select register, and finally reads a byte from offset address 0003H in the video RAM. As a result, the AL register contains the bit values for the pixels with coordinates (24, 0) to (31, 0) from bitplane 2.

```
mov dx,3CEh          ;port address of the graphics cont. addr. reg.
mov ax,0005h         ;write read mode 0 in the mode register
out dx,ax
mov ax,0204h         ;write the value 2 (plane number) in the
out dx,ax            ;read map select register
mov ax,0A000h        ;segment address of the video RAM
mov ds,ax            ;to DS
mov si,0003h         ;offset address into the video RAM
lodsb                ;read byte from plane 2
```

Read mode 1

Read mode 1 specifies which of the eight pixels in the specified byte of video RAM is set to a certain color. This is determined by the individual bits in the read byte which correspond to the one of the eight pixels from the specified byte in the video RAM. If a pixel has the specified color (appropriate bit map), then the corresponding bit will be 1, else 0. The bit pattern of the color to be compared must be loaded into the lower four bits of the Color Compare register. The lower four bits of the Color Don't Care register show which bitplanes will be taken into consideration in the comparison. The value 1 includes the given plane in the comparison, while the value 0 excludes it.



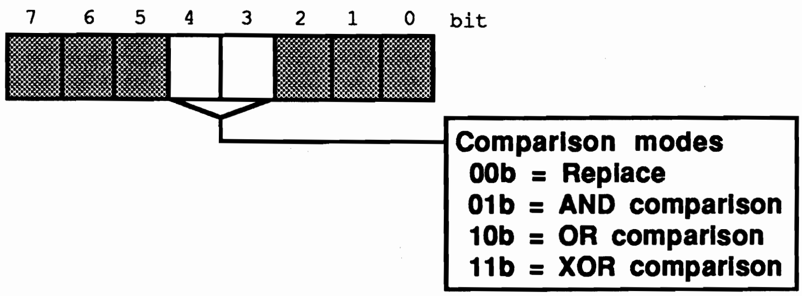
Video RAM read access in read mode 1

The following program sequence determines which of the pixels between coordinates (0, 0) and (7, 0) have color value five. First, read mode 1 is set by the Mode register. Then the color value to be tested (five) is loaded into the Color Compare register. We must also load the Color Don't Care register with the value 1111b so that all four bitplanes will be included in the comparison. However, this is the default value and we have not loaded any other value into this register, so we can skip this step. After programming the registers of the graphics controller, we load the segment and offset addresses of the pixels to be compared into the DS and SI registers. Then the read is executed from the video RAM.

```
mov dx,3CEh          ;port address of the graphics cont. addr. reg.
mov ax,0805h         ;write read mode 1 into the mode register
out dx,ax
mov ax,0502h         ;write color value 15 into the
out dx,ax            ;Color Compare register
mov ax,0A000h        ;segment address of the video RAM
mov ds,ax            ;to DS
xor si,si            ;load offset address 0
lodsb                ;read and compare pixels,
                     ;return result in AL
```

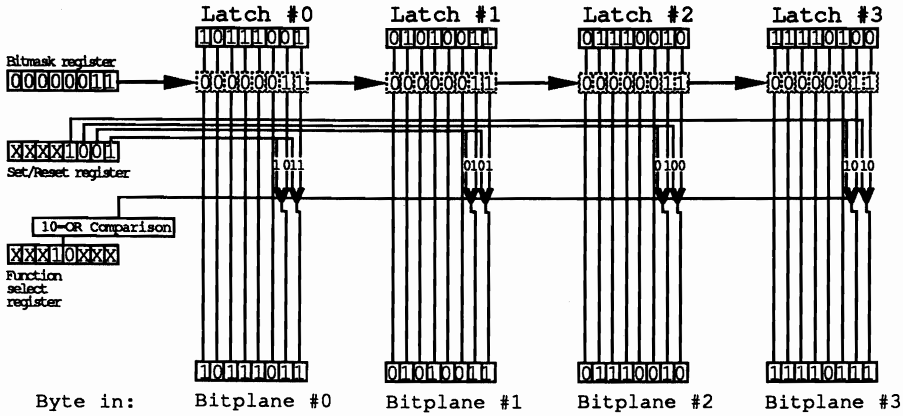
Write mode 0

Writing to the video RAM in write mode 0 results in a number of operations, all of which depend on the contents of several registers. The contents of the Bit Mask register determine whether the value of a bit in the four latch registers will be written unchanged to the found bitplanes or whether it will first be modified. The individual bits in the Bit Mask register correspond to the individual bits in the four latch registers. If a bit in the Bit Mask register is 0, the corresponding bits in the latch registers will be written to the bitplanes unchanged. If this bit is 1, a modification will take place, dependent on the contents of the Function Select register. As the following figure shows, the bits can be replaced or modified with the logical operations AND, OR, and XOR.



Function Select Register structure in EGA card graphics controller

The contents of the Enable Set/Reset register determines from where the other operand in these operations will come. If the lower four bits contain the value 1, the other operand will come from the lower four bits of the Set/Reset register. Each of these bits is then combined with the bits from the latch registers as described by the contents of the Function Select register. All of the bits to be modified from latch register 0 will then be operated on with bit 0 of the Set/Reset register. In the same manner, all of the bits to be modified from latch registers 1, 2, and 3 are combined with bits 1, 2, and 3 of the Set/Reset register, respectively. The byte which is actually written to the graphics controller becomes irrelevant at this point—the write access is reduced to a trigger, which cannot have any direct influence on the contents of the latch register (and therefore the bitplanes).



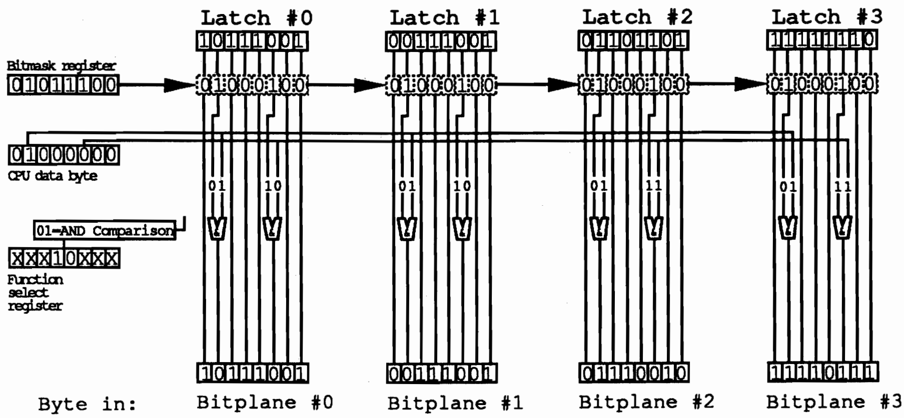
Write access to video RAM (write mode 0) when Enable Set/Reset register contains a value of 00001111(b)

The following assembly language fragment assigns the pixels at coordinates (5, 0) and (7, 0), found at offset address 0000H in the video RAM, the color 1011(b).

Since we don't want to change the color of the other pixels, the contents of the byte are first read into the latch register with a read access to the video RAM. It is not important which read mode is active because the byte transmitted to the CPU is irrelevant; all we are interested in is loading the latch register. Since only bits 0 (coordinates (7, 0)) and 2 (coordinates (5, 0)) will be changed, we load the value 00000101b (05h) into the bitmask register. In the Function Select register we write the value 0 because we want to replace bits 0 and 2 with a new bit combination. We write the color we want to give to the two bits (1011b = 0Bh) in the Set/Reset register. We must also write the value 1111(b) (0FH) to the Enable Set/Reset register of the graphics controller so that the color value will be taken from the Set/Reset register. We can then execute the write access to video RAM.

```
mov ax,0A000h        ;segment address of the video RAM
mov ds,ax            ;to DS
xor bx,bx            ;load offset address 0
mov al,[bx]          ;load byte 0 in the latch register
mov dx,3CEh          ;port address of the graphic cont. addr. reg.
mov ax,0005h         ;read mode 0, write more 0
out dx,ax            ;write in the mode register
mov al,03h           ;write 0 in the Function Select register
out dx,ax
mov ax,0508h         ;write bit mask in the bitmask register
out dx,ax
mov ax,0B00h         ;write new color value in the Set/Reset register
out dx,ax
mov ax,0F01h         ;write 1111b in the Enable Set/Reset register
out dx,ax
mov [bx],al          ;trigger latch register
```

Things are different when the Enable Set/Reset register contains the value zero. In this case all of the bits to be modified from the four latch registers are combined with the CPU byte latch by latch. Here again the type of operation performed depends on the contents of the Function Select register. For example, if the OR operation is selected and bits 1, 2, 4, and 6 are to be modified, than these bits of all four latch registers will be individually ORed with bits 1, 2, 4, and 6 in the CPU byte.



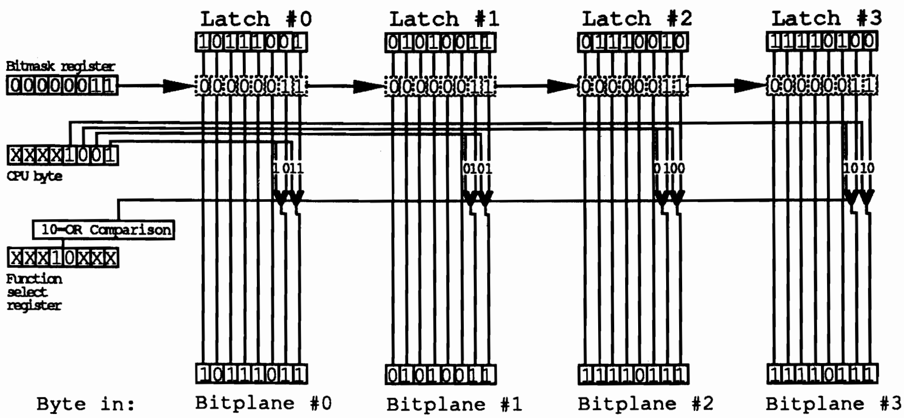
Write mode 1

Write mode 1 is quite simple compared to the complex operations of write mode 0. The contents of the registers and the CPU byte are irrelevant because the contents of the four latch registers are loaded unchanged into the specified offset address within the four bitplanes. This is useful for copying the color values of eight successive pixels to eight other pixels, for instance. The byte containing the eight pixels can be read under one of the read modes, placing it in the latch registers. Then a write access can be made to the byte in video RAM to which you want to copy the color values. The graphics controller will automatically copy the contents of the latch registers to the specified position within the four bitplanes.

To write these color values to other locations, you can use additional write accesses. No more read accesses are necessary, since the latch registers already contain the appropriate values and their contents are not changed by the write access.

Write mode 2

Write mode 2 resembles a combinátion of the various modes of write mode 0. As in write mode 0, the bitmask register determines which bits will be taken directly from the latch registers and which will be modified. The manner in which these bits are manipulated is again determined by the mode selected in the Function Select register. The lower four bits of the CPU byte will be combined with the latch registers, independent of the Enable Set/Reset register. Bit zero of the CPU byte is combined with all bits in latch register zero which are to be modified. The same applies for CPU bits 1, 2, and 3, which are combined with the bits of latch registers 1, 2, and 3, respectively.



Write access to video RAM in write mode 2

This mode is good for setting the colors of individual pixels, as we demonstrated in the example in write mode 0. In contrast to write mode 0, the assembly-language fragment is somewhat shorter because neither the Enable Set/Reset nor the Set/Reset register has to be programmed. Here is the same example using write mode 2:

```
mov ax,0A000h        ;segment address of the video RAM
mov ds,ax            ;in DS
xor bx,bx            ;load offset address 0
mov al,[bx]          ;load byte 0 in the latch registers
mov dx,3CEh          ;port address of the graphics cont. addr. reg.
mov ax,0205h         ;read mode 0, write mode 2
out dx,ax            ;write into the mode register
mov ax,0003h         ;write REPLACE mode (0) in the Function
out dx,ax            ;Select register
mov ax,0508h         ;write the bit mask to the bitmask register
out dx,ax
mov al,0Bh           ;new color value in AL
mov [bx],al          ;and from there to the video RAM and
                     ;into the latch regs and bitplanes
```

Demonstration program

The following program demonstrates the following basic graphics routines:

- Calculating the position of a pixel within the video RAM
- Setting the color of a pixel
- Reading the color of a pixel
- Filling the entire video RAM with a color

If you have followed this section closely, especially the material on the read and write modes, you won't have any problems following the logic of the various functions. Since it contains detailed documentation, we won't say anything more about it.

It should be noted that the program is intended for demonstration purposes only. You can develop it further if you want to make a graphics library out of these functions. For example, the function PIXPTR loads the segment address of the video RAM into the ES register for calculating the position of a pixel within the video RAM each time it is called. This can be eliminated by loading this address into the register once at the beginning of the program and leaving it there, as long as the other functions do not change this register.

The graphics controller register programming can also be improved. Here the various registers are reloaded with the ROM-BIOS default values after the function has completed. This can be eliminated as long as you do not use the BIOS functions for character output (in the graphics mode) or the functions for setting and testing points within the module or program. If you avoid these calls, then these registers can be reset to their default values once at the end of the program instead of at the end of each routine.

Assembler listing: VEGA.ASM

```
;**********************************************************************;
;*                             V E G A                                *;
;*--------------------------------------------------------------------*;
;*    Task            : Creates elementary functions for accessing the *;
;*                      graphic modes on an EGA/VGA card               *;
;*--------------------------------------------------------------------*;
;*    Author          : MICHAEL TISCHER                                *;
;*    Developed on    :  10/3/1988                                     *;
;*    Last update     :  6/19/1989                                     *;
;*--------------------------------------------------------------------*;
;*    Assembly        : MASM VEGA;                                     *;
;*                      LINK VEGA;                                     *;
;*--------------------------------------------------------------------*;
;*    Call            : VEGA                                           *;
;**********************************************************************;

;== Constants ==========================================================

VIO_SEG      = 0A000h               ;Segment address of video RAM
                                    ;in graphic mode
LINE_LEN     = 80                   ;Every graphi line in EGA/VGA graphic
                                    ;modes require 80 bytes
BITMASK_REG  = 8                    ;Bitmask register
MODE_REG     = 5                    ;Mode register
FUNCSEL_REG  = 3                    ;Function select register
MAPSEL_REG   = 4                    ;Map-Select register
ENABLE_REG   = 1                    ;Enable Set/Reset register
SETRES_REG   = 0                    ;Set/Reset register
GRAPH_CONT   = 3CEh                 ;Port addressd of graphic controller
OP_MODE      = 0                    ;Comparison operator mode:
                                    ;   00h = Replace
                                    ;   08h = AND comparison
                                    ;   10h = OR comparison
                                    ;   18h = EXCLUSIVE OR comparison

GR_640_350   = 10h                  ;BIOS code for 640x350-pixel
```

```
                                        ;16-color graphic mode
TX_80_25     = 03h                      ;BIOS code for 80*25-char.
                                        ;text mode

;== Stack ==============================================================

stack        segment para stack         ;Definition of stack segment

             dw 256 dup (?)             ;256-word stack

stack        ends                       ;End of stack segment

;== Data ===============================================================

data         segment para 'DATA'        ;Definition of data segment

;== Data for the demo program ==========================================

initm   db 13,10
          db "VEGA (c) 1988 by Michael Tischer"
     db 13,10,13,10
     db "This demonstration program operates only with an EGA/",13,10
          db "card and a hi-res monitor. If your PC doesn't have this",13,10
          db "configuration, please press the <s> key to abort the",13,10
          db "program.",13,10
     db "Press any other key to start the program.",13,10,"$"

data         ends                       ;End of data segment

;== Code ===============================================================

code         segment para 'CODE'        ;Definition of code segment

             assume cs:code, ds:data, es:data, ss:stack

;== Demo program =======================================================

demo         proc far

             mov  ax,data               ;Get segment addr. from data segment
             mov  ds,ax                 ;and load into DS
             mov  es,ax                 ;and ES

             ;-- Display opening message and wait for input ---------------

             mov  ah,9                  ;Function number for string display
             mov  dx,offset initm       ;Message address
             int  21h                   ;Call DOS interrupt

             xor  ah,ah                 ;Get function number for key
             int  16h                   ;Call BIOS keyboard interrupt
             cmp  al,"s"                ;Was <s> entered?
             je   ende                  ;YES --> End program
             cmp  al,"S"                ;Was <S> entered?
             jne  startdemo             ;NO --> Start demo

ende:        mov  ax,4C00h              ;Function no. for end program
             int  21h                   ;Call DOS interrupt 21H

             ;-- Initialize graphic mode ----------------------------------

startdemo label near

             mov  ax,GR_640_350         ;Initialize 64x350-pixel
             int  10h                   ;16-color graphic mode
```

```
         mov  ch,000100001b      ;Color: Blue
         mov  ax,350             ;Number of raster lines: 350
         call fillscr            ;Fill screen

         ;-- The program displays two squares on the screens (the   --
         ;-- second is really a copy of the first) until the user   --
         ;-- presses a key to end the program                       --

         xor  ch,ch              ;Set color to 0
dl:      mov  ax,100             ;Starting line of first square

         inc  ch                 ;Increment color
         and  ch,15              ;AND bits 4 and 7

d2:      mov  bx,245             ;Starting column of first square
d3:      call setpix             ;Set pixel
         push cx                 ;Save color
         call getpix             ;Get pixel color
         push ax                 ;Push coordinates onto stack
         push bx
         add  bx,100             ;Compute position of second
         add  ax,100             ;square
         call setpix             ;Set pixel of copy
         pop  bx                 ;Return coordinates of first square
         pop  ax
         pop  cx                 ;Get color
         inc  bx                 ;Increment column
         cmp  bx,295             ;Reached the last column?
         jne  d3                 ;NO --> Set next pixel

         inc  ax                 ;YES, Increment line
         cmp  ax,150             ;Reached the last line?
         jne  d2                 ;NO --> Work with next line

         mov  ah,1               ;Read keyboard
         int  16h                ;Call BIOS keyboard interrupt
         je   dl                 ;No key pressed --> Continue

         mov  ax,TX_80_25        ;80x25 text mode
         int  10h                ;Initialization
         jmp  short ende         ;End programm

demo     endp

;== Functions used in the demo program ================================

;-- PIXPTR: Computes the address of a pixel within video RAM for the   -
;--         new EGA/VGA graphic modes
;-- Input    : AX    = Graphic line
;--            BX    = Graphic column
;-- Output   : ES:BX = Pointer to the byte in video RAM containing pixel
;--            CL    = Number of right shifts for the byte
;--                  = Number of byte shifts in ES:BX needed to isolate
;--                    the pixel
;--            AH    = Bitmask for combining with all other pixels
;-- Registers: ES, AX, BX and CL are changed

pixptr   proc near

         push dx                 ;Push DX onto stack

         mov  cl,bl              ;Save low byte of graphic column
         mov  dx,LINE_LEN        ;Number of bytes per line to DX
         mul  dx                 ;AX = graphic line * LINE_LEN
         shr  bx,1               ;Shift graphic column three places to
         shr  bx,1               ;the right, divide by 8
```

```
         shr  bx,1
         add  bx,ax                ;Add line offset

         mov  ax,VIO_SEG           ;Load segment address of video RAM
         mov  es,ax                ;into ES

         and  cl,7                 ;And bits 4 - 7 of graphic column
         xor  cl,7                 ;Turn bits 0 - 3 then
                                   ;subtract 7 - CL
         mov  ah,1                 ;After shift, bit 0 should be
                                   ;left alone

         pop  dx                   ;Pop DX off of stack
         ret                       ;Back to caller

pixptr   endp

;-- SETPIX: Sets a graphic pixel in the new EGA/VGA graphic modes ------
;-- Input    : AX    = graphic line
;--            BX    = graphic column
;--            CH    = pixel color
;-- Output   : none
;-- Registers: ES, DX and CL are changed

setpix   proc near

         push ax                   ;Push coordinates onto
         push bx                   ;the stack

         call pixptr               ;Computer pointer to the pixel

         mov  dx,GRAPH_CONT        ;Load port addr. of graphic controller

         ;-- Set bit position in bitmask register --------------------

         shl  ah,cl                ;Mask for bit to be changed
         mov  al,BITMASK_REG       ;Move bitmask register from AL
         out  dx,ax                ;Write to register

         ;-- Set read mode 0 and write mode 2 -- ----------------------

         mov  ax,MODE_REG + (2 shl 8) ;Reg. no. and ,mode value
         out  dx,ax                ;Write in the register

         ;-- Define comparison mode between preceding latch -----------
         ;-- contents, and CPU byte                         -----------

         mov  ax,FUNCSEL_REG + (OP_MODE shl 8) ;Write register number
         out  dx,ax                ;and comparison operator

         ;-- Pixel control -------------------------------------------

         mov  al,es:[bx]           ;Load latches
         mov  es:[bx],ch           ;Move color into bitplanes

         ;-- Set altered registers to their default (BIOS) ------------
         ;-- status                                        ------------

         mov  ax,BITMASK_REG + (0FFh shl 8) ;Set old bitmask
         out  dx,ax                ;Write in the register
         mov  ax,MODE_REG          ;Write old value for for mode register
         out  dx,ax                ;into register
         mov  ah,FUNCSEL_REG       ;Write old value for function select
         out  dx,ax                ;register into register
```

```
         pop  bx                  ;Pop coordinates off of stack
         pop  ax                  ;
         ret                      ;Back to caller

setpix   endp

;-- GETPIX: Places a pixel's color in one of the new EGA/VGA -----------
;--        graphic modes
;-- Input    : AX    = graphic line
;--            BX    = graphic column
;-- Output   : CH    = graphic pixel color
;-- Registers: ES, DX , CX and DI are changed

getpix   proc near

         push ax                  ;Push coordinates onto
         push bx                  ;the stack

         call pixptr              ;Computer pointer to pixel
         mov  ch,ah               ;Move bitmask to CH
         shl  ch,cl               ;Shift bitmask by bit positions

         mov  di,bx               ;Move video RAM offset to DI
         xor  bl,bl               ;Color value will be computed in BL

         mov  dx,GRAPH_CONT       ;Load graphic controller port address
         mov  ax,MAPSEL_REG + (3 shl 8) ;Access bitplane #3

         ;-- Go through each of the four bitplanes -------------------

gpl:     out  dx,ax               ;Activate bitplane #AH only
         mov  bh,es:[di]          ;Get byte from the bitplane
         and  bh,ch               ;Omit uninteresting bits
         neg  bh                  ;Bit 7 = 1, when a pixel is set
         rol  bx,1                ;Shift bit 7 from BH to Bit 1 in BL

         dec  ah                  ;Decrement bitplane number
         jge  gpl                 ;Not  -1 yet? --> next bitplane

         ;-- The map select register must not be reset, since       --
         ;-- the EGA- and VGA-BIOS default to a value of 0          --

         mov  ch,bl               ;Get color from CH
         pop  bx                  ;Pop coordinates off
         pop  ax                  ;of stack
         ret                      ;Back to caller

getpix   endp

;-- FILLSCR: Sets all screen pixels to one color ------ ---------------
;-- Input    : AX    = number of graphic lines on the screen
;--            CH    = pixel color
;-- Output   : none
;-- Registers: ES, AX, CX, DI, DX and BL are changed

fillscr  proc near

         mov  dx,GRAPH_CONT       ;Load graphic controller port address
         mov  al,SETRES_REG       ;Numbmer of Set-/Reset registers
         mov  ah,ch               ;Move bit combination to AL
         out  dx,ax               ;Write to the register

         mov  ax,ENABLE_REG + (0Fh shl 8) ;Write 0FH in the
         out  dx,ax               ;Enable Set-/Reset register

         mov  bx,LINE_LEN / 2     ;Length of a graphic line / 2 into BX
         mul  bx                  ;Multiply by number of graphic lines
         mov  cx,ax               ;Move to CX as repeat counter
         xor  di,di               ;Address first byte in video RAM
         mov  ax,VIO_SEG          ;Segment address of video RAM
```

```
          mov  es,ax              ;Load into ES
          cld                     ;Increment on string instructions
          rep  stosw              ;Fill video RAM

          ;-- Return old contents of Enable Set-/Reset register    -----

          mov  dx,GRAPH_CONT      ;Load graphic controller port address
          mov  ax,ENABLE_REG      ;Write 00H in Enable Set-/
          out  dx,ax              ;Reset register

          ret                     ;Back to caller

fillscr   endp

;== End =================================================================

code      ends                    ;End of code segment
          end  demo               ;Start program execution with DEMO
```

10.6 Determining the Type of Video Card

Whenever you want to access video card hardware or use a BIOS function which is only available in special versions of the BIOS, you should first ensure that the card in question is actually installed in the system. If your program doesn't make such a test, then the result may not be what you wanted to appear on the screen.

It is especially important for an application program to recognize the type of video card installed, if your program is supposed to work the same on all types of cards while still directly accessing video hardware. The output routines need this information to make optimum use of the special properties of the given card.

Remember that the PC can have both a monochrome video card (MDA, HGC or EGA with a monochrome monitor) and a color video card (EGA, VGA, or CGA) installed, although only one of the two cards may be active at one time.

Combinations allowable for PC video cards					
	VGA	EGA	HGC	CGA	MDA
VGA			■		■
EGA			■	■	■
HGC	■	■		■	
CGA		■	■		■
MDA	■	■		■	

We need to find out what video cards are installed. There are no BIOS or DOS functions for doing this, nor are there any variables we can read. We have to write an assembly language routine which checks the existence of different video cards. We can refer to the documentation for the various cards, since most manufacturers include some procedure for determining if their card is in use. It is important to keep the test specific (i.e., it does not return a positive result if a certain type of video card is not installed). This presents problems for EGA and VGA cards, which can emulate CGA or MDA cards with the appropriate monitor, and are difficult to distinguish from true CGA or MDA cards.

All of the tests described here are found at the end of this section in the form of two assembly language programs intended for use with C and Pascal programs. The functions place the type of video card installed and the type of monitor connected to it into an array to which the function is passed a pointer. If two video cards are installed, their order in the array indicates which one is active.

The following cards can be detected by the assembly language routine:

- MDA cards
- CGA cards
- HGC cards

- EGA cards
- VGA cards

Since the assembly language routine checks selectively for the existence of a certain video card, there is a separate subroutine for each type of video card. It bears the name of the video card for which it tests. These routines have names like TEST_EGA, TEST_VGA, etc. The tests could be called sequentially, but certain tests can be excluded if we know they would return a negative result. This is case for the CGA test, for example, if an EGA or VGA card has already been detected and is connected to a high-resolution color monitor. A CGA card cannot be installed alongside such a card, so there is no point in testing for it.

There is a flag for each test which determines whether or not the test will be performed. Before the first test, the VGA test, all of the flags are set to 1 so that all of the tests will be performed in order. During the testing, certain flags can be set to 0 for reasons mentioned above, and the corresponding tests will not be made.

VGA test

The tests begin with the VGA test. It is very easy because there is a special function in the VGA BIOS, sub-function 00H of function 1AH, which returns precisely the information that the assembly language routine needs. The information is available only if a VGA card and hence a VGA BIOS is installed. This is the case if the value 1AH is found in the AL register after the call. If the test routine encounters a different value there, the VGA test will be terminated and the other tests will be performed. This indicates that a VGA card is not installed.

After this function is called, the BL register contains a special device code for the active video card and the BH register contains a code for the inactive card. The following codes can occur:

Code	Meaning
00H	No video card
01H	MDA card/monochrome monitor
02H	CGA card/color monitor
03H	Reserved
04H	EGA card/high-resolution monitor
05H	EGA card/monochrome monitor
06H	Reserved
07H	VGA card/analog monochrome monitor
08H	VGA card/analog color monitor

These codes are separated into values for the video card and the monitor connected to it, and loaded into the array whose address is passed to the assembly language routine. Since this routine already has information about both video cards, the following tests do not have to be performed. The routine executes the monochrome test, however, if the functions discover a monochrome card, since it cannot distinguish between an MDA and HGC card.

EGA test

After the VGA test comes the EGA test, which it performed only if the VGA test was unsuccessful, and thus the EGA flag was not cleared. It uses a function which is found only in the EGA BIOS: sub-function 10H of function 12H. If no EGA card is installed and this function is not available, the value 10H will still be found in the BL register after the function call. In this case the EGA test ends.

If an EGA card is installed, the CL register will contain the settings of the DIP switches on the EGA card after the call. These switches indicate what type of monitor is connected. They are converted to the monitor codes the assembly language routine uses and placed in the array along with the code for the EGA card. The CGA or monochrome test flag is cleared depending on the type of monitor connected. The EGA routine ends.

CGA test

If the CGA flag has not been cleared by the previous tests, the CGA test follows the EGA test. As with the monochrome test, there are no special BIOS functions which can be used and we have to check for the presence of the appropriate hardware. In both routines this is done by calling the routine TEST_6845, which tests to see if the 6845 video controller found on these cards is at the specified port address. On a CGA card this is port address 3D4H, which is passed to the routine TEST_6845.

The only way to test the existence of the CRTC at a given port address is to write some value (other than 0) to one of the CRTC registers and then read it back immediately. If the value read matches the value written, then the CRTC and thus the video card are present. But before writing a value into a CRTC register, we should stop to consider that these registers have a major impact on the construction of the video signals and careless access to them can not only thoroughly confuse the CRTC, it can even harm the monitor. Registers 0 to 9 are out of the question for this test, leaving us with registers 10 to 15, all of which have an effect on the screen contents. The best we can do is registers 10 and 11, which control the starting and ending lines of the cursor.

The assembly language routine first reads the contents of register 10 before it loads any value into this register. After a short pause so that the CRTC can react to the output, the contents of this register are read back. Before the value read is compared to the original value, the old value is first written back into the register so that the test disturbs the screen as little as possible. If the comparison is positive, then a CRTC is present and so is the video card (CGA in this case). The CGA routine responds by loading the code for a color monitor into the array, since this is the only type of monitor which can be used with a CGA card.

Monochrome test

The last test is the monochrome test, which also checks for the existence of a CRTC, this time at port address 3B4H. If it finds a CRTC there, then a monochrome card is installed and we have to figure out if it is an MDA or HGC hard. The status registers of the two cards, at port address 3BAH, are used to determine this. While bit 7 of this register has no significance on the MDA card and its value is thus undefined, it contains a 1 on an HGC card whenever the electron beam is returning across the screen. Since this is not permanent and occurs only at intervals of about two milliseconds, the contents of this bit constantly alternates between 0 and 1.

Hercules

The test routine first reads the contents of this register and masks out bits 0 to 6. The resulting value is used in a maximum of 32768 loop passes, where the value is read again and compared with the original value. If the value changes, meaning that the state of bit 7 changes, then an HGC card is probably installed. If this bit does not change over the course of 32768 loop passes, then an MDA card is in use.

Here again we place the appropriate code for the video card in the array. The monitor code is also set to monochrome, since this is the only monitor which can be connected to an MDA or HGC card.

Primary and secondary video systems

The tests are now over. Now we have to figure out which card is active (primary) and which is inactive (secondary). If the outcome of the VGA test was positive, we can skip this because the VGA BIOS routine determines the active card automatically.

In other cases we can determine the active video card from the current video mode, which can be read with the help of function 0FH of the BIOS video interrupt. If the value seven is returned, then the 80x25 text mode of the monochrome card is active. All of the other modes indicate that a CGA, EGA, or VGA card is active. This information is used to exchange the order of the two entries in the array if it does not match the actual situation.

The assembly language routine returns control to the calling program.

Here we include C and Pascal programs which call the function GetVIOS from the assembly language module, and demonstrate how GetVIOS works.

C listing: VIOSC.C

```
/***********************************************************************/
/*                              V I O S C                              */
/*---------------------------------------------------------------------*/
/*    Task          : Determines the type of video card and monitor    */
/*                    installed in the system.                         */
/*---------------------------------------------------------------------*/
/*    Author        : MICHAEL TISCHER                                   */
/*    Developed on  : 10/02/1988                                        */
/*    Last update   : 06/20/1988                                        */
/*---------------------------------------------------------------------*/
/*    (MICROSOFT C)                                                     */
/*    Creation      : CL /AS /c VIOSC.C                                 */
/*                    LINK VIOSC VIOSCA                                 */
/*    Call          : VIOSC                                             */
/*---------------------------------------------------------------------*/
/*    (BORLAND TURBO C)                                                 */
/*    Creation      : Create project file made of the following:       */
/*                    VIOSC                                             */
/*                    VIOSCA.OBJ                                        */
/*    Info          : Some cards may return errors or "unknown"        */
/***********************************************************************/

/*== Declarations of external functions ===============================*/

extern void get_vios( struct vios * );

/*== Type defs =======================================================*/

typedef unsigned char BYTE;                           /* Create a byte */

/*== Structures ======================================================*/

struct vios {            /* Describes video card and attached monitor */
             BYTE vcard,
                  monitor;
           };

/*== Constants =======================================================*/

/*-- Constants for the video card ----------------------------------*/

#define NO_VIOS    0                                  /* No video card */
#define VGA        1                                      /* VGA card */
#define EGA        2                                      /* EGA card */
#define MDA        3                     /* Monochrome Display Adapter */
#define HGC        4                       /* Hercules Graphics Card */
#define CGA        5                       /* Color Graphics Adapter */

/*-- Constants for monitor type -------------------------------------*/

#define NO_MON     0                                    /* No monitor */
#define MONO       1                            /* Monochrome monitor */
#define COLOR      2                                 /* Color monitor */
#define EGA_HIRES  3                    /* High-res/multisync monitor */
#define ANLG_MONO  4                     /* Analog monochrome monitor */
#define ANLG_COLOR 5                          /* Analog color monitor */

/***********************************************************************/
/**                            MAIN PROGRAM                          **/
/***********************************************************************/

void main()

{
 static char *vcnames[] = {          /* Pointer to the video card name */
                           "VGA",
                           "EGA",
```

```
                        "MDA",
                        "HGC",
                        "CGA"
                       };

  static char *monnames[] = {    /* Pointer to the monitor type's name */
                         "monochrome monitor",
                         "color monitor",
                         "high-res/multisync monitor",
                         "analog monochrome monitor",
                         "analog color monitor"
                        };

  struct vios vsys[2];                         /* Vector for GET_VIOS */

  get_vios( vsys );                          /* Determine video system */
  printf("\nVIOSC (c) 1988 by Michael Tischer\n\n");
  printf("Primary Video System:   %s card/ %s\n",
         vcnames[vsys[0].vcard-1], monnames[vsys[0].monitor-1]);
  if ( vsys[1].vcard != NO_VIOS )  /* Is there secondary video system? */
    printf("Secondary Video System: %s card/ %s\n",
           vcnames[vsys[1].vcard-1], monnames[vsys[1].monitor-1]);
}
```

Assembler listing: VIOSCA.ASM

```
;**********************************************************************;
;*                             V I O S C A                            *;
;*--------------------------------------------------------------------*;
;*    Task            : Creates a function for determining video      *;
;*                      adapter and monitor type, when linked with    *;
;*                      a C program.                                  *;
;*--------------------------------------------------------------------*;
;*    Author          : MICHAEL TISCHER                               *;
;*    Developed on    : 10/02/1988                                    *;
;*    Last update     : 06/20/1989                                    *;
;*--------------------------------------------------------------------*;
;*    Assembly        : MASM VIOSCA;                                  *;
;*                      ... link to a C program                       *;
;**********************************************************************;

;== Constants for VIOS structure =======================================

                                      ;Video card constants
NO_VIOS     = 0                       ;No video card
VGA         = 1                       ;VGA card
EGA         = 2                       ;EGA card
MDA         = 3                       ;Monochrome Display Adapter
HGC         = 4                       ;Hercules Graphics Card
CGA         = 5                       ;Color Graphics Adapter

                                      ;Monitor constants
NO_MON      = 0                       ;No monitor
MONO        = 1                       ;Monochrome monitor
COLOR       = 2                       ;Color monitor
EGA_HIRES   = 3                       ;High-resolution or multisync monitor
ANLG_MONO   = 4                       ;Analog monochrome monitor
ANLG_COLOR  = 5                       ;Analog color monitor

;== Segment declarations for the C program ============================

IGROUP group _text                    ;Addition to program segment
DGROUP group const, bss,  _data       ;Addition to data segment
       assume CS:IGROUP, DS:DGROUP, ES:DGROUP, SS:DGROUP

CONST  segment word public 'CONST';This segment includes all read-only
CONST  ends                           ;constants

_BSS   segment word public 'BSS'  ;This segment includes all
```

```
_BSS   ends                      ;un-initialized static variables

_DATA  segment word public 'DATA' ;Data segment

vios_tab   equ this byte

           ;-- Conversion table for return values of function 1AH, ---
           ;-- sub-function 00H of the VGA-BIOS                    ---

           db NO_VIOS, NO_MON     ;No video card
           db MDA    , MONO       ;MDA card and monochrome monitor
           db CGA    , COLOR      ;CGA card and color monitor
           db ?      , ?          ;Code 3 unused
           db EGA    , EGA_HIRES  ;EGA card and hi-res monitor
           db EGA    , MONO       ;EGA card and monochrome monitor
           db ?      , ?          ;Code 6 unused
           db VGA    , ANLG_MONO  ;VGA card and analog mono monitor
           db VGA    , ANLG_COLOR ;VGA card and analog color monitor

ega_dips   equ this byte

           ;-- Conversion table for EGA card DIP switch settings -------

           db COLOR, EGA_HIRES, MONO
           db COLOR, EGA_HIRES, MONO

_DATA  ends

;== Program ============================================================

_TEXT  segment byte public 'CODE' ;Program segment

public     _get_vios

;-----------------------------------------------------------------------
;-- GET_VIOS: Determines types of installed video cards ----------------
;-- Call from C : void get_vios( struct vios *vp );
;-- Declaration : struct vios { BYTE vcard, monitor; };
;-- Return value: none
;-- Info        : This example uses function in SMALL memory model

_get_vios  proc near

sframe     struc                  ;Stack access structure
cga_possi  db ?                   ;Local variable
ega_possi  db ?                   ;Local variable
mono_possi db ?                   ;Local variable
bptr       dw ?                   ;Take BP
ret_adr    dw ?                   ;Return address to caller
vp         dw ?                   ;Pointer to first VIOS structure
sframe     ends                   ;End of structure

frame      equ [ bp - cga_possi ] ;Address elements of the structure

           push bp                ;Push BP onto stack
           sub  sp,3              ;Allocate space for local variables
           mov  bp,sp             ;Transfer SP to BP
           push di                ;Push DI onto stack

           mov  frame.cga_possi,1 ;Could be CGA
           mov  frame.ega_possi,1 ;Could be EGA
           mov  frame.mono_possi,1;Could be MDA or HGC

           mov  di,frame.vp       ;Get offset address of structure
           mov  word ptr [di],NO_VIOS    ;Still no video
           mov  word ptr [di+2],NO_VIOS  ;system found

           call test_vga          ;Test for VGA card
           cmp  frame.ega_possi,0 ;EGA card still possible?
           je   gv1               ;NO --> Test for CGA
```

```
                call test_ega           ;Test for EGA card
gv1:            cmp  frame.cga_possi,0 ;CGA card still possible
                je   gv2               ;NO --> Test for MDA/HGC

                call test_cga           ;Test for CGA card
gv2:            cmp  frame.mono_possi,0;MDA or HGC card still possibleh?
                je   gv3               ;NO --> End tests

                call test_mono          ;Test for MDA/HGC cards

                ;-- Determine active video card ---------------------------

gv3:            cmp  byte ptr [di],VGA ;VGA card active?
                je   gvi_end           ;YES, active card already determined
                cmp  byte ptr [di+2],VGA ;VGA card as secondary system?
                je   gvi_end           ;YES, active card already determined

                mov  ah,0Fh            ;Determine active video mode using the
                int  10h               ;BIOS video interrupt

                and  al,7              ;Only modes 0-7 are of interest
                cmp  al,7              ;Monochrome card active?
                jne  gv4               ;NO, in CGA  or EGA mode

                ;-- MDA, HGC, or EGA card (mono) is active -----------------

                cmp  byte ptr [di+1],MONO ;Mono monitor in first structure?
                je   gvi_end           ;YES, Sequence o.k.
                jmp  short switch      ;NO, Change sequence

                ;-- CGA or EGA card currently active ----------------------

gv4:            cmp  byte ptr [di+1],MONO ;Mono monitor in first structure?
                jne  gvi_end           ;NO, Sequence o.k.

switch:         mov  ax,[di]           ;Get contents of first structure
                xchg ax,[di+2]         ;Exchange with second structure
                mov  [di],ax

gvi_end:        pop  di                ;Get DI from stack
                add  sp,3              ;Get local variables from stack
                pop  bp                ;Get BP from stack
                ret                    ;Return to C program

_get_vios       endp

;-----------------------------------------------------------------------
;-- TEST_VGA: Determines whether a VGA card is installed

test_vga        proc near

                mov  ax,1a00h          ;Function 1AH, sub-function 00H
                int  10h               ;calls VGA-BIOS
                cmp  al,1ah            ;Is this function supported?
                jne  tvga_end          ;NO --> End routine

                ;-- If function is supported, BH contains the active video --
                ;-- system code; BH contains the inactive video sys. code  --

                mov  cx,bx             ;Move result to CX
                xor  bh,bh             ;Set BH to 0
                or   ch,ch             ;Just one video system?
                je   tvga_1            ;YES --> Convey first system's code

                ;-- Convert code of second system -------------------------

                mov  bl,ch             ;Move second system code to BL
                add  bl,bl             ;Add offset to table
                mov  ax,offset DGROUP:vios_tab[bx] ;Get code from table and
```

```
            mov   [di+2],ax          ;place in caller's structure
            mov   bl,cl              ;Move first system's codes to BL

            ;-- Convert code of first system ---------------------------

tvga_1:     add   bl,bl              ;Add offset to table
            mov   ax,offset DGROUP:vios_tab[bx] ;Get code from table and
            mov   [di],ax            ;place in caller's structure

            mov   frame.cga_possi,0 ;CGA test failed
            mov   frame.ega_possi,0 ;EGA test failed
            mov   frame.mono_possi,0 ;MONO still needs testing

            mov   bx,di              ;Address of active structure
            cmp   byte ptr [bx],MDA ;Monochrome system available?
            je    do_tmono           ;YES --> Execute MDA/HGC test

            add   bx,2               ;Address of inactive structure
            cmp   byte ptr [bx],MDA ;Monochrome system available?
            jne   tvga_end           ;NO --> End routine

do_tmono:   mov   word ptr [bx],0    ;Pretend that this system
                                     ;is still unavailable
            mov   frame.mono_possi,1;Execute monochrome test

tvga_end:   ret                      ;Back to caller

test_vga    endp

;-----------------------------------------------------------------------
;-- TEST_EGA: Determines whether an EGA card is installed

test_ega    proc near

            mov   ah,12h             ;Function 12H
            mov   bl,10h             ;Sub-function 10H
            int   10h                ;Call EGA-BIOS
            cmp   bl,10h             ;Is the function supported?
            je    tega_end           ;NO --> End routine

            ;-- When this function is supported, CL contains the EGA ----
            ;-- card's DIP switch settings                           ----

            mov   al,cl              ;Move DIP switch settings to AL
            shr   al,1               ;Shift one position to the right
            mov   bx,offset DGROUP:ega_dips ;Offset address of table
            xlat                     ;Move element AL from table to AL
            mov   ah,al              ;Move monitor type to AH
            mov   al,EGA             ;It's an EGA card
            call  found_it           ;Move data to vector

            cmp   ah,MONO            ;Connected to monochrome monitor?
            je    is_mono            ;YES --> not MDA or HGC

            mov   frame.cga_possi,0 ;Cannot be a CGA card
            jmp   short tega_end     ;End routine

is_mono:    mov   frame.mono_possi,0;If EGA card is connected to a mono
                                     ;monitor, it can be installed as
                                     ;either an HGC or MDA

tega_end:   ret                      ;Back to callerr

test_ega    endp

;-----------------------------------------------------------------------
;-- TEST_CGA: Determines whether a CGA card is installed

test_cga    proc near
```

```
            mov  dx,3D4h            ;CGA tests port addr. of CRTC addr.
            call test_6845          ;reg., to see if 6845 is installed
            jc   tega_end           ;NO --> End test

            mov  al,CGA             ;YES --> CGA is installed
            mov  ah,COLOR           ;CGA has color monitor attached
            jmp  found_it           ;Transfer data to vector

test_cga    endp

;-----------------------------------------------------------------------
;-- TEST_MONO: Checks for the existence of an MDA or HGC card

test_mono   proc near

            mov  dx,3B4h            ;Check port address of CRTC addr. reg.
            call test_6845          ;with MONO to see if there's a 6845
                                    ;installed
            jc   tega_end           ;NO --> End test

            ;-- If there is a monochrome video card installed, the ------
            ;-- following determines whether it's an MDA or an HGC ------

            mov  dl,0BAh            ;Read MONO status port using 3BAH
            in   al,dx              ;
            and  al,80h             ;Check bit 7 only and
            mov  ah,al              ;move to AH

            ;-- If contents of bit 7 change during one of the following -
            ;-- readings, the card is handled as an HGC                  -

            mov  cx,8000h           ;Maximum of 32768 loop executionse
test_hgc:   in   al,dx              ;Read status port
            and  al,80h             ;Check bit 7 only
            cmp  al,ah              ;Contents changed?
            jne  is_hgc             ;Bit 7 = 1 --> HGC
            loop test_hgc           ;Continue loop

            mov  al,MDA             ;Bit 7 <> 1 --> MDA
            jmp  set_mono           ;Set parameters

is_hgc:     mov  al,HGC             ;Bit 7 = 1 --> ist HGC
set_mono:   mov  ah,MONO            ;MDA/HGC on mono monitor
            jmp  found_it           ;Set parameters

test_mono   endp

;-----------------------------------------------------------------------
;-- TEST_6845: Sets carry flag if no 6845 exists in port address of DX

test_6845   proc near

            mov  al,0Ah             ;Register 10
            out  dx,al              ;Register number of CRTC address reg.
            inc  dx                 ;DX now in CRTC data register

            in   al,dx              ;Get contents of register 10
            mov  ah,al              ;and move to AH

            mov  al,4Fh             ;Any value
            out  dx,al              ;Write to register 10

            mov  cx,100             ;Short delay loop--gives 6845 time
wait:       loop wait               ;to react

            in   al,dx              ;Read contents of register 10
            xchg al,ah              ;Exchange AH and AL
            out  dx,al              ;Send old valuen

            cmp  ah,4Fh             ;Written value read?
```

```
          je   t6845_end         ;YES --> End test

          stc                    ;NO --> Set carry flag

t6845_end: ret                   ;Back from caller

test_6845  endp

;----------------------------------------------------------------------
;-- FOUND_IT: Transfers video card type to AL and monitor type to -----
;--           AH in the video vector                              -----

found_it   proc near

           mov bx,di              ;Address of active structure
           cmp word ptr [bx],0    ;Video system already onboard?
           je  set_data           ;NO --> Data in active structure

           add bx,2               ;YES, Address of inactive structure

set_data:  mov [bx],ax            ;Place data in structure
           ret                    ;Back to caller

found_it   endp

;----------------------------------------------------------------------

_text      ends                   ;End of code segment
           end                    ;End of program
```

Pascal listing: VIOSP.PAS

```
{**********************************************************************}
{*                              V I O S P                             *}
{*--------------------------------------------------------------------*}
{*    Task            : Returns the type of video card installed.     *}
{*--------------------------------------------------------------------*}
{*    Author          : MICHAEL TISCHER                               *}
{*    Developed on    :  10/02/1988                                   *}
{*    Last update     :  06/19/1989                                   *}
{*--------------------------------------------------------------------*}
{*    Info            : Some of the values given here may not coincide *}
{*                      with some video cards (e.g., some CGA cards   *}
{*                      may return "Unknown card").                   *}
{**********************************************************************}

program VIOSP;

{$L c:\masm\viospa}                          { Link assembler module }
                                     { Change path to suit your DOS needs }
const NO_VIOS     = 0;                                 { No video card }
      VGA         = 1;                                    { VGA card }
      EGA         = 2;                                    { EGA card }
      MDA         = 3;                    { Monochrome Display Adapter }
      HGC         = 4;                       { Hercules Graphics Card }
      CGA         = 5;                      { Color Graphics Adapter }

      NO_MON      = 0;                                  { No monitor }
      MONO        = 1;                          { Monochrome monitor }
      COLOR       = 2;                               { Color monitor }
      EGA_HIRES   = 3;                     { High-resolution monitor }
      ANLG_MONO   = 4;                   { Monochrome analog monitor }
      ANLG_COLOR  = 5;                        { Color analog monitor }

type Vios = record           { Describes video card and attached monitor }
              VCard,
              Monitor : byte;
             end;
```

```
    ViosPtr = ^Vios;                    { Pointer to a VIOS structure }

procedure GetVios( vp : ViosPtr ) ; external ;

var VidSys : array[1..2] of Vios;  { Array containing video structures }

{***********************************************************************}
{* PrintSys: Gives information about a video system                   *}
{* Input    : - VCARD: Code number of the video card                  *}
{*            - MON  : Code number of the attached monitor            *}
{* Output   : none                                                    *}
{***********************************************************************}

procedure PrintSys( VCard, Mon : byte );

begin
  write(' ');
  case VCard of
    NO_VIOS : write('Unknown');                    { For "other" code }
    VGA : write('VGA');
    EGA : write('EGA');
    MDA : write('MDA');
    CGA : write('CGA');
    HGC : write('HGC');
  end;
  write(' card/ ');
  case Mon of
    NO_MON : write('unknown monitor');         { For "other" monitors }
    MONO        : writeln('monochrome monitor');
    COLOR       : writeln('color monitor');
    EGA_HIRES   : writeln('high-resolution monitor');
    ANLG_MONO   : writeln('monochrome analog monitor');
    ANLG_COLOR  : writeln('color analog monitor');
  end;
end;

{***********************************************************************}
{**                          MAIN  PROGRAM                            **}
{***********************************************************************}

begin
  GetVios( @VidSys );                     { Check installed video card }
  writeln('VIOSP  -  (c) 1988 by MICHAEL TISCHER');
  write('Primary video system: ');
  PrintSys( VidSys[1].VCard, VidSys[1].Monitor );
  writeln(#13#10);
  if VidSys[2].VCard <> NO_VIOS then  { Second video system installed? }
    begin                                                       { YES }
      write('Secondary video system:');
      PrintSys( VidSys[2].VCard, VidSys[2].Monitor );
      writeln(#13#10);
    end;
end.
```

Assembler listing: VIOSPA.ASM

```
;**********************************************************************;
;*                             V I O S P A                            *;
;*--------------------------------------------------------------------*;
;*    Task            : Creates a function for determining the type   *;
;*                      of video card installed on a system. This     *;
;*                      routine must be assembled into an OBJ file,   *;
;*                      then linked to a Turbo Pascal (4.0) program.  *;
;*--------------------------------------------------------------------*;
;*    Author          : MICHAEL TISCHER                               *;
;*    Developed on    : 10/02/1988                                    *;
;*    Last update     : 06/19/1989                                    *;
;*--------------------------------------------------------------------*;
;*    assembly        : MASM VIOSPA;                                  *;
```

```
;*                      ... Link to a Turbo Pascal program           *;
;*                          using the {$L VIOSPA} compiler directive   *;
;*********************************************************************;

;== Constants for the VIOS structure ================================

                                   ;Video card constants
NO_VIOS     = 0                    ;No video card/unrecognized card
VGA         = 1                    ;VGA card
EGA         = 2                    ;EGA card
MDA         = 3                    ;Monochrome Display Adapter
HGC         = 4                    ;Hercules Graphics Card
CGA         = 5                    ;Color Graphics Adapter

                                   ;Monitor constants
NO_MON      = 0                    ;No monitor/unrecognized code
MONO        = 1                    ;Monochrome monitor
COLOR       = 2                    ;Color Monitor
EGA_HIRES   = 3                    ;High-resolution/multisync monitor
ANLG_MONO   = 4                    ;Monochrome analog monitor
ANLG_COLOR  = 5                    ;Analog color monitor

;== Data segment =====================================================

DATA   segment word public         ;Turbo data segment

DATA   ends

;== Code segment =====================================================

CODE        segment byte public    ;Turbo code segment

            assume cs:CODE, ds:DATA

public      getvios

;-- Initialized global variables must be placed in the code segment ----

vios_tab    equ this word

            ;-- Conversion table for supplying return values of VGA  ----
            ;-- BIOS function 1A(h), sub-function 00(h)              ----

            db NO_VIOS, NO_MON     ;No video card
            db MDA    , MONO       ;MDA card/monochrome monitor
            db CGA    , COLOR      ;CGA card/color monitor
            db ?      , ?          ;Code 3 unused
            db EGA    , EGA_HIRES  ;EGA card/hi-res monitor
            db EGA    , MONO       ;EGA card/monochrome monitor
            db ?      , ?          ;Code 6 unused
            db VGA    , ANLG_MONO  ;VGA card/analog mono monitor
            db VGA    , ANLG_COLOR ;VGA card/analog color monitor

ega_dips    equ this byte

            ;-- Conversion table for EGA card DIP switches -----

            db COLOR, EGA_HIRES, MONO
            db COLOR, EGA_HIRES, MONO

;---------------------------------------------------------------------
;-- GETVIOS: Determines type(s) of installed video card(s) ------------
;-- Pascal call : GetVios ( vp : ViosPtr ); external;
;-- Declaration : Type Vios = record VCard, Monitor: byte;
;-- Return Value: None

getvios  proc near

sframe      struc                  ;Stack access structure
cga_possi   db ?                   ;local variables
```

```
ega_possi  db ?                    ;local variables
mono_possi db ?                    ;local variables
bptr       dw ?                    ;BPTR
ret_adr    dw ?                    ;Return address of calling program
vp         dd ?                    ;Pointer to first VIOS structure
sframe     ends                    ;End of structure

frame      equ [ bp - cga_possi ] ;Address elements of structure

           push bp                 ;Push BP onto stack
           sub  sp,3               ;Allocate memory for local variables
           mov  bp,sp              ;Transfer SP to BP

           mov  frame.cga_possi,1 ;Is it a CGA?
           mov  frame.ega_possi,1 ;Is it an EGA?
           mov  frame.mono_possi,1;Is it an MDA or HGC?

           mov  di,word ptr frame.vp    ;Get offset addr. of structure
           mov  word ptr [di],NO_VIOS   ;No video system or unknown
           mov  word ptr [di+2],NO_VIOS ;system found

           call test_vga           ;Test for VGA card
           cmp  frame.ega_possi,0 ;Or is it an EGA card?
           je   gv1                ;NO -->Go to CGA test

           call test_ega           ;Test for EGA card
gv1:       cmp  frame.cga_possi,0 ;Or is it a CGA card?
           je   gv2                ;NO --> Go to MDA/HGC test

           call test_cga           ;Test for CGA card
gv2:       cmp  frame.mono_possi,0;Or is it an MDA or HGC card?
           je   gv3                ;NO --> End tests

           call test_mono          ;Test for MDA/HGC card

           ;-- Determine video configuration ---------------------------

gv3:       cmp  byte ptr [di],VGA ;VGA card?
           je   gvi_end            ;YES --> Active card already indicated
           cmp  byte ptr [di+2],VGA;VGA card part of secondary system?
           je   gvi_end            ;YES --> Active card already indicated

           mov  ah,0Fh             ;Determine video mode using BIOS video
           int  10h                ;interrupt

           and  al,7               ;Only modes 0-7 are of interest
           cmp  al,7               ;Mono card active?
           jne  gv4                ;NO --> CGA or EGA mode

           ;-- MDA, HGC or EGA card (mono) currently active ------------

           cmp  byte ptr [di+1],MONO ;Mono monitor in first structure?
           je   gvi_end            ;YES, Sequence o.k.
           jmp  short switch       ;NO, Switch sequence

           ;-- CGA or EGA card currently active ------------------------

gv4:       cmp  byte ptr [di+1],MONO ;Mono monitor in first structure?
           jne  gvi_end            ;NO -->Sequence o.k.

switch:    mov  ax,[di]            ;Get contents of first structure
           xchg ax,[di+2]          ;Switch with second structure
           mov  [di],ax

gvi_end:   add  sp,3               ;Add local variables from stack
           pop  bp                 ;Pop BP off of stack
           ret  4                  ;Clear variables off of stack;
                                   ;Return to Turbo
getvios    endp
```

```
;-------------------------------------------------------------------
;-- TEST_VGA: Determines whether a VGA card is installed

test_vga   proc near

           mov  ax,1a00h          ;Function 1A(h), sub-function 00(h)
           int  10h               ;Call VGA-BIOS
           cmp  al,1ah            ;Function supported?
           jne  tvga_end          ;NO --> End routine

           ;-- If function is supported, BL contains the code of the ---
           ;-- active video system, while BH contains the code of    ---
           ;-- the inactive video system                             ---

           mov  cx,bx             ;Move result in CX
           xor  bh,bh             ;Set BH to 0
           or   ch,ch             ;Only one video system?
           je   tvga_1            ;YES --> Display first system's code

           ;-- Convert code of second system --------------------------

           mov  bl,ch             ;Move second system's code to BL
           add  bl,bl             ;Add offset to table
           mov  ax,vios_tab[bx]   ;Get code from table and move into
           mov  [di+2],ax         ;caller's structure
           mov  bl,cl             ;Move first system's code into BL

           ;-- Convert code of second system --------------------------

tvga_1:    add  bl,bl             ;Add offset to table
           mov  ax,vios_tab[bx]   ;Get code from table
           mov  [di],ax           ;and move into caller's structure

           mov  frame.cga_possi,0 ;CGA test fail?
           mov  frame.ega_possi,0 ;CGA test fail?
           mov  frame.mono_possi,0 ;Test for mono

           mov  bx,di             ;Address of active structure
           cmp  byte ptr [bx],MDA ;Monochrome system online?
           je   do_tmono          ;YES --> Execute MDA/HGC test

           add  bx,2              ;Address of inactive structure
           cmp  byte ptr [bx],MDA ;Monochrome system online?
           jne  tvga_end          ;NO --> End routine

do_tmono:  mov  word ptr [bx],0   ;Emulate if this system
                                  ;isn't available
           mov  frame.mono_possi,1;Execute monochrome test

tvga_end:  ret                    ;Return to caller

test_vga   endp

;-------------------------------------------------------------------
;-- TEST_EGA: Determine whether an EGA card is installed

test_ega   proc near

           mov  ah,12h            ;Function 12(h)
           mov  bl,10h            ;Sub-function 10(h)
           int  10h               ;Call EGA-BIOS
           cmp  bl,10h            ;Is this function supported?
           je   tega_end          ;NO --> End routine

           ;-- If the function IS supported, CL contains the       ----
           ;-- EGA card DIP switch settings                        ----

           mov  bl,cl             ;Move DIP switches to BL
           shr  bl,1              ;Shift one position to the right
           xor  bh,bh             ;Index high byte to 0
```

```
            mov  ah,ega_dips[bx]   ;Get element from table
            mov  al,EGA            ;Is it an EGA card?
            call found_it          ;Transfer data to the vector

            cmp  ah,MONO           ;Mono monitor connected?
            je   is_mono           ;YES --> Not MDA or HGC

            mov  frame.cga_possi,0 ;No CGA card possible
            jmp  short tega_end    ;End routine

is_mono:    mov  frame.mono_possi,0;EGA can either emulate MDA or HGC,
                                   ;if mono monitor is attached

tega_end:   ret                    ;Back to caller

test_ega    endp

;----------------------------------------------------------------------
;-- TEST_CGA: Determines whether a CGA card is installed

test_cga    proc near

            mov  dx,3D4h           ;Port addr. of CGA's CRTC addr. reg.
            call test_6845         ;Test for installed 6845 CRTC
            jc   tega_end          ;NO --> End test

            mov  al,CGA            ;YES, CGA installed
            mov  ah,COLOR          ;CGA uses color monitor
            jmp  found_it          ;Transfer data to vector

test_cga    endp

;----------------------------------------------------------------------
;-- TEST_MONO: Checks for MDA or HGC card

test_mono   proc near

            mov  dx,3B4h           ;Port addr. of MONO's CRTC addr. reg.
            call test_6845         ;Test for installed 6845 CRTC
            jc   tega_end          ;NO --> End test

            ;-- Monochrome video card installed                   ------
            ;--
            mov  dl,0BAh           ;MONO status port at 3BA(h)
            in   al,dx             ;Read status port
            and  al,80h            ;Separate bit 7 and
            mov  ah,al             ;move to AH

            ;-- If the contents of bit 7 in the status port change  ----
            ;-- during the following readings, it is handled as an  ----
            ;-- HGC                                                 ----

            mov  cx,8000h          ;maximum 32768 loop executions
test_hgc:   in   al,dx             ;Read status port
            and  al,80h            ;Isolate bit 7
            cmp  al,ah             ;Contents changed?
            jne  is_hgc            ;Bit 7 = 1 --> HGC
            loop test_hgc          ;Continue

            mov  al,MDA            ;Bit 7 <> 1 --> MDA
            jmp  set_mono          ;Set parameters

is_hgc:     mov  al,HGC            ;Bit 7 = 1 --> HGC
set_mono:   mov  ah,MONO           ;MDA and HGC set as mono screen
            jmp  found_it          ;Set parameters

test_mono   endp

;----------------------------------------------------------------------
;-- TEST_6845: Returns set carry flag if 6845 doesn't lie in the
```

```
;--            port address in DX

test_6845  proc near

           mov  al,0Ah            ;Register 10
           out  dx,al             ;Register number in CRTC address reg.
           inc  dx                ;DX now in CRTC data register

           in   al,dx             ;Get contents of register 10
           mov  ah,al             ;and move to AH

           mov  al,4Fh            ;Any value
           out  dx,al             ;Write to register 10

           mov  cx,100            ;Short wait loop to which
wait:      loop wait              ;6845 can react

           in   al,dx             ;Read contents of register 10
           xchg al,ah             ;Exchange Ah and AL
           out  dx,al             ;Send value

           cmp  ah,4Fh            ;Written value been read?
           je   t6845_end         ;YES --> End test

           stc                    ;NO --> Set carry flag

t6845_end: ret                    ;Back to caller

test_6845  endp

;-----------------------------------------------------------------------
;-- FOUND_IT: Transfers type of video card to AL and type of       -----
;--           monitor in AH in the video vector                     -----

found_it   proc near

           mov bx,di              ;Address of active structure
           cmp word ptr [bx],0    ;Video system already  onboard?
           je  set_data           ;NO --> Data in  active structure

           add bx,2               ;YES --> Address of inactive structure

set_data:  mov [bx],ax            ;Place data in structure
           ret                    ;Back to caller

found_it   endp

;-----------------------------------------------------------------------

code       ends                   ;End of code segment
           end                    ;End of program
```

10.7 Accessing Video RAM from High Level Languages

The beginning of this chapter mentioned the option of video RAM access from high level languages. This would allow the developer to write screen output routines for high level languages that would execute faster than output commands available to the languages, BIOS functions, or DOS functions. This option would be particularly attractive if it meant that we could write these routines without assembly language programming.

The demonstration programs below implement direct video RAM access routines which display a string on the screen. Althrough there are some major differences between the three programs as a result of the differences between the respective languages (BASIC, Pascal and C), all three programs contain the same elements.

Initialization

Each program includes an initialization routine which determines the segment address of the video RAM. The routine has a variable which contains the address of the CRTC address register. There is a direct relationship between the video RAM and this address register: just as this register is always at port address 3B4H, the video RAM on a monochrome card is always found at segment address B000H. This combination also applies to color cards, where the address register is at port address 3D4H and the video RAM is at segment address B800H. If we know the port address of the CRTC address register, we can determine the segment address of the video RAM. Once we have determined this address, we can place it in a global variable and execute the initialization routine.

Output

All three programs have an output routine which uses the segment address we determined above. Each time the routine displays something, it determines the starting address of the video page currently displayed on the screen. This ensures that the output appears on the visible screen, and not on an undisplayed video page. We can find this from the CRT_START BIOS variable. This variable is located at address 0040:004E, and specifies the offset address of the displayed video page relative to the video page found at offset address 0000H.

After this address is determined, we can access the video RAM. The method used in the program depends on the given programming language. Let's look at each program in more detail.

The C implementation

From a programming point of view, this is the cleanest of the three implementations because the video RAM can be treated as a normal variable in C. We first define the structure VELB, which describes the ASCII/attribute pair as it appears in the video RAM. We create a new data type, VP, to act as a pointer to this structure. It is important that this pointer be of type FAR because these structures are in the video RAM and therefore outside the C data segment. Smaller memory models in C require the declaration of this pointer as a FAR pointer.

The global variable VPTR is initialized to be a pointer to the first ASCII/attribute pair in page 0 of the video RAM. This occurs in the INIT_DPRINT routine. It is used within the DPRINT function (the function used for display) as the basis for addressing the characters within the video RAM.

The DPRINT function loads the LPTR pointer with the address of the screen output position passed to the routine. LPTR is first loaded with the contents of the global variable VPTR, and then with the offset address of the active video page, as found in the CRT_START BIOS variable. LPTR must be cast as a BYTE pointer because the contents of the BIOS variable refers to bytes, and not to VELB structures. If the cast operator were missing, the C compiler would generate code which would first multiply the contents of the BIOS variable by the length of the VELB structure before adding it, resulting in the wrong value.

We can now add the display position to this pointer. The output position is passed to DPRINT as row and column coordinates. The video RAM is treated as an array of 2000 components, each of which is a VELB structure. Since we have computed the base address of the array in LPTR, all we need is to index into it. We multiply the row coordinate by 80 (columns per line) and then add the column coordinate. Finally we have a pointer to the output position in video RAM, which we can treat like any other C pointer.

Each time through, the loop increments the pointer to the next VELB structure. We write the ASCII code of the character and the color passed to DPRINT to the specified address. This repeats until the program reaches the end of the string.

C listing: DVIC.C

```
/**********************************************************************/
/*                               D V I C                              */
/*--------------------------------------------------------------------*/
/*    Task            : Demonstrates direct access to video RAM.      */
/*--------------------------------------------------------------------*/
/*    Author          : MICHAEL TISCHER                               */
/*    Developed on    : 10/01/1988                                    */
/*    Last update     : 06/21/1989                                    */
/*--------------------------------------------------------------------*/
/*    (MICROSOFT C)                                                   */
/*    Creation        : CL /AS DVIC.C                                 */
/*    Call            : DVIC                                          */
/*--------------------------------------------------------------------*/
/*    (BORLAND TURBO C)                                               */
/*    Creation        : RUN menu command (no project file needed)     */
/**********************************************************************/

/*== Include files ===================================================*/

#include <dos.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <bios.h>
```

```
/*== Type definitions ==========================================*/

typedef unsigned char BYTE;                         /* Create a byte */
typedef struct velb far * VP;          /* VP = FAR pointer in video RAM */
typedef BYTE BOOL;                     /* similar to BOOLEAN in Pascal */

/*== Structures ================================================*/

struct velb {            /* Describes a 2-byte position on the screen */
             BYTE character,                         /* ASCII code */
                  attribute;                /* Character attribute */
            };

/*== Macros ====================================================*/

/*-- MK_FP creates a FAR pointer to an object from a segment   -------*/
/*-- address and offset address                                -------*/

#ifndef MK_FP                                 /* MK_FP not defined yet? */
#define MK_FP(seg, ofs) ((void far *) ((unsigned long) (seg)<<16|(ofs)))
#endif

#define COLOR(VG, HG) ((VG << 3) + HG)

/*== Constants =================================================*/

#define TRUE  1                       /* Constants for use with BOOL */
#define FALSE 0

/*-- The following constants return pointers to variables from the ---*/
/*-- BIOS variable segment at segment address 0x40                ---*/

#define CRT_START ((unsigned far *) MK_FP(0x40, 0x4E))
#define ADDR_6845 ((unsigned far *) MK_FP(0x40, 0x63))

#define NORMAL        0x07          /* Character attribute definition */
#define BRIGHT        0x0f          /*  Based on monochrome video card*/
#define INVERSE       0x70
#define UNDERSCORED   0x01
#define BLINKING      0x80

#define BLACK         0x00         /* Color attributes for color card */
#define BLUE          0x01
#define GREEN         0x02
#define COBALTBLUE    0x03
#define RED           0x04
#define VIOLET        0x05
#define BROWN         0x06
#define LIGHTGRAY     0x07
#define DARKGRAY      0x01
#define LIGHTBLUE     0x09
#define LIGHTGREEN    0x0A
#define LIGHTCOBALT   0x0B
#define LIGHTRED      0x0C
#define LIGHTVIOLET   0x0D
#define YELLOW        0x0E
#define WHITE         0x0F

/*== Global variables ==========================================*/

VP vptr;                    /* Pointer to first character in video RAM */

/********************************************************************
*  Function          : D P R I N T                                  *
**----------------------------------------------------------------**
*  Task              : Writes a string directly to video RAM        *
*                                                                   *
*  Input parameters : - COLUMN   = Output column                    *
*                     - LINES    = Output row                       *
*                     - COLOR    = Character attribute              *
```

```
*                         - STRING   = Pointer to string               *
*  Return value       : None                                           *
***********************************************************************/

void dprint(BYTE column, BYTE lines, BYTE color, char * string)

{
 register VP lptr;                    /* Floating pointer in video RAM */
 register BYTE i;                    /* Points to number of characters */

 /*-- Set pointer to output position in video RAM -------------------*/
 lptr = (VP) ((BYTE far *) vptr + *CRT_START) + lines * 80 + column;
 for (i=0 ; *string ; ++lptr, ++i)                /* Execute string */
  {
   lptr->character = *(string++);           /* Character in video RAM */
   lptr->attribute = color;                /* Set character attribute */
  }
}

/***********************************************************************
*  Function           : I N I T _ D P R I N T                          *
**--------------------------------------------------------------------**
*  Task               : Determines video RAM segment address for DPRINT *
*  Input parameters   : None                                           *
*  Return value       : None                                           *
*  Info               : Allocates segment address of video RAM in VPTR *
*                       global variable                                *
***********************************************************************/

void init_dprint()

{
 vptr = (VP) MK_FP( (*ADDR_6845 == 0x3B4) ? 0xB000 : 0xB800, 0 );
}

/***********************************************************************
*  Function           : C L S                                          *
**--------------------------------------------------------------------**
*  Task               : Clears the screen with the help of DPRINT      *
*                                                                      *
*  Input parameters   : - COLOR    = Character attribute               *
*  Return value       : None                                           *
***********************************************************************/

void cls( BYTE color )

{
 static char blankline[81] =
  { ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
    ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
    ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
    ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
    ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
    ' ',' ',' ',' ',' ','\0'
  };

 register BYTE i;                                    /* Loop counter */

 for (i=0; i<24; ++i)                           /* Execute each line */
  dprint(0, i, color, blankline);              /* Display blank line */
}

/***********************************************************************
*  Function           : N O K E Y                                      *
**--------------------------------------------------------------------**
*  Task               : Tests for a keypress                           *
*  Input parameters   : None                                           *
*  Return value       : TRUE if a key is pressed, otherwise FALSE      *
***********************************************************************/
```

```
BOOL nokey()

{
#ifdef __TURBOC__                        /* Compiling this with TURBO C? */
 return( bioskey( 1 ) == 0 );            /* YES, read keyboard from BIOS */
#else                                             /* Using Microsoft C */
 return( _bios_keybrd( _KEYBRD_READY ) == 0 );       /* Read from BIOS */
#endif
}

/**********************************************************************/
/**                           MAIN  PROGRAM                          **/
/**********************************************************************/

void main()
{
 BYTE firstcol,                   /* Color of first square on the screen */
      color,                                  /* Color of current square */
      column,                                 /* Current output position */
      lines;

 init_dprint();               /* Determine segment address of video RAM */
 cls( COLOR(BLACK, GREEN) );                             /* Clear screen */
 dprint(22, 0, WHITE, "DVIC  - (c) 1988 by Michael Tischer");
 firstcol = BLACK  ;                                 /* Start with black */
 while( nokey() )                /* Repeat until the user presses a key */
  {
   if (++firstcol > WHITE)                         /* Reached last color? */
    firstcol = BLUE;                            /* YES, continue with blue */
   color = firstcol;                       /* Set first color on the screen */

   /*-- Fill screen with squares -------------------------------------*/

   for ( column=0; column < 80; column += 4)
    for (lines=1; lines < 24; lines += 2)
      {
      dprint( column, lines,   color, "████");/* Block characters can */
      dprint( column, lines+1, color, "████");/* be created by press- */
      color = ++color & 15;                  /* ing <Alt><2><1><9>    */
     }
  }
}
```

The Pascal implementation

By using the keyword ABSOLUTE or by linking in a small assembly language routine it would also be possible to treat the video RAM as a normal variable in Turbo Pascal. But there's an easier way.

Turbo Pascal offers the arrays MEMW and MEM for accessing memory which is outside of the data segment of the Turbo Pascal program. The array MEM consists of bytes and the array MEMW of words. The two arrays don't actually exist and are just mapped to the address space, but that doesn't affect their usefulness.

We can write values into the array as well as read from it. This is done with the following statement:

```
MEMW[ segment address : offset address ] := expression
```

or

```
variable := MEMW[ segment address : offset address ]
```

The MEM array might be easier to use for this particular application since we will be alternating between ASCII characters and a constant attribute. However, the output procedure DPrint uses the MEMW array instead, because 16-bit accesses are performed faster than two successive 8-bit accesses on 16-bit machines.

When accessing the MEMW array, DPrint takes the segment address of the video RAM from the variable VSeg, which is initialized at the start of the program in the procedure InitDPrint. As described before, this is done by examining the BIOS variable which contains the port address of the CRTC address register. This and the other BIOS variables are declared using the ABSOLUTE keyword, allowing them to be used in the program like any other global variables.

The offset within the MEMW array is computed from the starting address of the screen page. The coordinates are passed to DPrint, in which the row coordinate is multiplied by 160 and the column coordinate by two. When running through the string to be printed, the memory offset is incremented by two on each pass, moving it one ASCII/attribute pair to the right.

Pascal listing: DVIP.P

```
{**********************************************************************}
{*                              D V I P                               *}
{*--------------------------------------------------------------------*}
{*    Task              : Demonstrates direct access to video RAM from *}
{*                        Turbo Pascal                                *}
{*--------------------------------------------------------------------*}
{*    Author            : MICHAEL TISCHER                             *}
{*    Developed on      : 10/02/1987                                  *}
{*    Last update       : 06/20/1989                                  *}
{**********************************************************************}

program DVIP;

Uses Crt, Dos;                              { Use CRT and DOS units }

const NORMAL        = $07;             { Define character attributes in }
      LIGHT         = $0f;             { conjunction with monochrome    }
      INVERSE       = $70;             { video card                     }
      UNDERSCORED   = $01;
      BLINKING      = $80;

      BLACK         = $00;             { Color attributes for color card }
      BLUE          = $01;
      GREEN         = $02;
      COBALTBLUE    = $03;
      RED           = $04;
      VIOLET        = $05;
      BROWN         = $06;
      LIGHTGRAY     = $07;
      DARKGRAY      = $01;
      LIGHTBLUE     = $09;
      LIGHTGREEN    = $0A;
      LIGHTCOBALT   = $0B;
      LIGHTRED      = $0C;
      LIGHTVIOLET   = $0D;
      YELLOW        = $0E;
      WHITE         = $0F;

type TextTyp = string[80];

var VSeg : word;                          { Segment address of video RAM }
```

```
{**********************************************************************}
{* InitDPrint: Determines segment address of video RAM for DPrint     *}
{* Input    : none                                                    *}
{* Output   : none                                                    *}
{**********************************************************************}

procedure InitDPrint;

var CRTC_PORT : word absolute $0040:0063;  { Variable in BIOS var.seg. }

begin
  if CRTC_PORT = $3B4 then                   { Monochrome card connected? }
    VSeg := $B000                           { YES, video RAM at B000:0000 }
  else                                       { NO, must be a color card }
    VSeg := $B800;                            { Video RAM at B800:0000 }
end;

{**********************************************************************}
{* DPrint: Writes a string direct into video RAM                      *}
{* Input    : - COLUMN: Output column                                 *}
{*            - LINES : Output line                                   *}
{*            - COLOR : Color (attribute) for individual characters   *}
{*            - STROUT: String to be displayed                        *}
{* Output   : none                                                    *}
{**********************************************************************}

procedure DPrint( Column, Lines, Color : byte; StrOut : TextTyp);

var PAGE_OFS : word absolute $0040:$004E;  { Variable in BIOS var.seg. }
    Offset   : word;               { Pointer to current output position }
    i, j     : byte;                                    { Loop counter }
    Attribute : word;                           { Attribute for output }

begin
  Offset := Lines * 160 + Column * 2 + PAGE_OFS;
  Attribute := Color shl 8;    { High byte for word access to video RAM }
  i := length( StrOut );                     { Determine string length }
  for j:=1 to i do                                    { Execute string }
    begin             { Put character & attribute directly into video RAM }
      memw[VSeg:Offset] := Attribute or ord( StrOut[j] );
      Offset := Offset + 2;  { Set offset to next ASCII/attribute pair }
    end;
end;


{**********************************************************************}
{* Demo: Demonstrates application of DPrint                           *}
{* Input    : none                                                    *}
{* Output   : none                                                    *}
{**********************************************************************}

procedure demo;

var Column,                                  { Current output position }
    Lines,
    Color    : integer;

begin
  TextBackGround( BLACK );                     { Turn background black }
  ClrScr;                                               { Clear screen }
  DPrint( 22, 0, WHITE, 'DVIP  - (c) 1988 by Michael Tischer');
  Randomize;                          { Enable random number generator }
  while not KeyPressed do            { Repeat until user presses a key }
    begin
      Column := Random( 76 );                { Select column, row and  }
      Lines := Random( 22 ) + 1;             { color at random         }
      Color := Random( 14 ) + 1;
      DPrint( Column, Lines,   Color, '[[[[');{ Block character can be }
```

```
      DPrint( Column, Lines+1, Color, '[[[[');{ created by pressing    }
    end;                                      {<Alt><2><1><9>          }
  ClrScr;                                                { Clear screen }
end;

{************************************************************************}
{**                          MAIN  PROGRAM                             **}
{************************************************************************}

begin
  InitDPrint;                         { Initialize output using DPrint }
  Demo;                                          { Demonstrate DPrint }
end.
```

The BASIC implementation

This version doesn't really fulfill its goal, since it is slower than the already slow PRINT command. But we have included it for the sake of completeness, and because it is a good example of how you can access the entire address space of the 8088 from within BASIC.

The commands DEF SEG, PEEK, and POKE are the heart of memory access in BASIC. DEF SEG sets the segment address of the "current" 64K segment. PEEK and POKE can then be used to read and write bytes from or to this segment. This technique is used in the initialization routine at line number 50000, which first defines the BIOS variable segment as the current segment. From there two PEEK commands read the port address of the CRTC address register and the variable VR is loaded with the segment address of the video RAM.

This address is used in the output routine at line number 51000 in combination with the DEF SEG command, which defines the video RAM as the current segment. But first we calculate the offset address in the video RAM by reading the start address of the current screen page from the BIOS variable area and then adding the offset address of the output position within the video RAM. As in the Pascal version, this is calculated by adding the product of the row coordinate (variable CLINE%) by 160 and the column coordinate (COLUMN%) by 2.

BASIC listing: DVIB.B

```
100 '**********************************************************************'
110 '*                              D V I B                               *'
120 '*--------------------------------------------------------------------*'
130 '*  Task           : Demonstrates direct access to video RAM          *'
150 '*  Author         : MICHAEL TISCHER                                  *'
160 '*  Developed on   : 10/01/1988                                       *'
170 '*  Last update    : 06/21/1989                                       *'
180 '**********************************************************************'
190 '
200 CLS : KEY OFF
210 GOSUB 50000                    'Determine segment address of video RAM
220 COLUMN%=22 : CLINE%=0 : COL% = 15
230 T$ = "DIVB - (c) 1988 by MICHAEL TISCHER" : GOSUB 51000
240 FCOL% = 0 : T$ = "[[[["             'Define string and starting color
250 A$ = INKEY$ : IF A$<>"" THEN 400    'Repeat until user presses a key
260 FCOL% = FCOL% + 1                           'Increment starting color
270 IF FCOL% > 15 THEN FCOL% = 1               'When FCOL%=16 make FCOL%=1
280 COL% = FCOL%                              'Set color for first square
290 FOR COLUMN%=0 TO 76 STEP 4                    'Execute for each column
300   FOR Z%=1 TO 24 STEP 2                         'Execute for each line
```

```
310     CLINE% = Z% : GOSUB 51000           'Display first line of square
320     CLINE% = Z%+1 : GOSUB 51000                  'Display second line
330     COL% = COL% + 1 AND 15                             'Set next color
340   NEXT
350 NEXT
360 GOTO 250
370 '
400 CLS                                                      'Clear screen
410 END
460 '
50000 '***************************************************************'
50010 '* Determine segment address of video RAM                      *'
50020 '*-------------------------------------------------------------*'
50030 '* Input  : none                                               *'
50040 '* Output : VR is the segment address of video RAM             *'
50050 '***************************************************************'
50060 '
50070 DEF SEG = &H40               'Segment address of BIOS variable range
50080 VR = PEEK(&H63) + PEEK(&H64) * 256                    'Get CRTC port
50090 IF VR = &H3B4 THEN VR = &HB000 ELSE VR = &HB800
50100 RETURN                                               'Back to caller
50120 '
51000 '***************************************************************'
51010 '* Write string direct into video RAM                          *'
51020 '*-------------------------------------------------------------*'
51030 '* Input  : - COLUMN% = the output column                      *'
51040 '*          - CLINE%  = the output line                        *'
51050 '*          - COL%    = string color                           *'
51060 '*          - T$      = the string to be displayed             *'
51070 '* Output : none                                               *'
51080 '***************************************************************'
51090 '
51100 DEF SEG = &H40               'Segment address of BIOS variable range
51110 OF% = PEEK(&H4E) + PEEK(&H4F) * 256       'Starting address of page
51120 OF% = OF% + COLUMN% * 2 + CLINE% * 160  'Offset of first character
51130 DEF SEG = VR                      'Set segment address of video RAM
51140 FOR I%=1 TO LEN(T$)                                 'Execute string
51150   POKE OF%, ASC(MID$(T$,I%,1))               'ASCII code in video RAM
51160   POKE OF%+1, COL%                               'Color in video RAM
51170   OF% = OF% + 2                     'Set offset to next character
51180 NEXT
51190 RETURN                                               'Back to caller
51200 '
```