Preface vii

The Intended Audience vii
Organization of the Manual vii
Suggestions for the Reader viii

PART I — Operating Principles

Chapter 1. Overview 1–1

Hardware Components 1–1
Resolution Modes 1–3
Operational Modes 1–3

Chapter 2. Monitor Configurations 2–1

Monochrome Monitor Only 2–2
Color Monitor Only 2–3
Dual Monitors 2–4

Chapter 3. Graphics Option Logic 3–1

General 3–1
Data Logic 3–2
Address Logic 3–2
Display Logic 3–6
GDC Command Logic 3–9
Contents

Chapter 4. Graphics Option Components  4–1
I/O Ports  4–1
Indirect Register  4–2
Write Buffer  4–2
Write Mask Registers  4–4
Pattern Generator  4–5
Foreground/Background Register  4–6
ALU/PS Register  4–8
Color Map  4–9
Mode Register  4–15
Scroll Map  4–16

PART II — Programming Guidelines

Chapter 5. Initialization and Control  5–1
Test for Option Present  5–1
Test for Motherboard Version  5–2
Initialize the Graphics Option  5–6
Controlling Graphics Output  5–24
Modifying and Loading the Color Map  5–25

Chapter 6. Bitmap Write Setup (General)  6–1
Loading the ALU/PS Register  6–1
Loading the Foreground/Background Register  6–2

Chapter 7. Area Write Operations  7–1
Display Data from Memory  7–1
Set a Rectangular Area to a Color  7–4

Chapter 8. Vector Write Operations  8–1
Setting Up the Pattern Generator  8–1
Display a Pixel  8–4
Display a Vector  8–5
Display a Circle  8–9

Chapter 9. Text Write Operations  9–1
Write a Byte-Aligned Character  9–1
Define and Position the Cursor  9–32
Write a Text String  9–38
## Chapter 10. Read Operations 10-1

- The Read Process 10-1
- Read the Entire Bitmap 10-1
- Pixel Write After a Read Operation 10-5

## Chapter 11. Scroll Operations 11-1

- Vertical Scrolling 11-1
- Horizontal Scrolling 11-4

## Chapter 12. Programming Notes 12-1

- Shadow Areas 12-1
- Bitmap Refresh 12-1
- Software Reset 12-2
- Setting Up Clock Interrupts 12-2
- Operational Requirements 12-3
- Set-Up Mode 12-3
- Timing Considerations 12-4

## PART III — Reference Material

### Chapter 13. Option Registers, Buffers, and Maps 13-1

- I/O Ports 13-1
- Indirect Register 13-3
- Write Buffer 13-4
- Write Mask Registers 13-5
- Pattern Register 13-6
- Pattern Multiplier 13-7
- Foreground/Background Register 13-8
- ALU/PS Register 13-9
- Color Map 13-10
- Mode Register 13-11
- Scroll Map 13-12

### Chapter 14. GDC Registers and Buffers 14-1

- Status Register 14-1
- FIFO Buffer 14-2

### Chapter 15. GDC Commands 15-1

- Introduction 15-1
- Video Control Commands 15-2
- Display Control Commands 15-8
- Drawing Control Commands 15-13
- Data Read Commands 15-18
PART IV — Appendixes

Appendix A. Option Specification Summary  A-1

Physical Specifications  A-1
Environmental Specifications  A-1
Power Requirements  A-2
Standards and Regulations  A-2
Part and Kit Numbers  A-3

Appendix B. Rainbow Graphics Option — Block Diagram  B-1

Appendix C. Getting Help  C-1

Index  I-1

Figures

Figure 1. Monochrome Monitor Only System  2-2
Figure 2. Color Monitor Only System  2-3
Figure 3. Dual Monitor System  2-4
Figure 4. Rows and Columns in Display Memory  3-3
Figure 5. Relationship of Display Memory to Address Logic  3-4
Figure 6. GDC Screen Control Parameters  3-8
Figure 7. Write Buffer as Accessed by the CPU and the GDC  4-3
Figure 8. Write Mask ReGiSter  4-4
Figure 9. Pattern Generator  4-5
Figure 10. Foreground/Background ReGiSter  4-7
Figure 11. Bitmap/Color Map Interaction (medium resolution)  4-10
Figure 12. Bitmap/Color Map Interaction (high resolution)  4-11
Figure 13. Sample Color Map with Loading Sequence  4-12
Figure 14. Scroll Map Operation  4-16
Figure 15. Rainbow Graphics Option — Block Diagram  B-3

Tables

Table 1. Colors and Monochrome Intensities — Displayed/Available  1-1
Table 2. Intensity Values vs Video Drive Voltages  4-14
Table 3. Clock Interrupt Parameters  12-2
Preface

The Intended Audience

The *Rainbow Color/Graphics Option Programmer's Reference Guide* is written for the experienced systems programmer who will be programming applications that display graphics on Rainbow video monitors. It is further assumed that the system programmer has had both graphics and 8088 programming experience.

The information contained in this document is not unique to any operating system; however, it is specific to the 8088 hardware and 8088-based software.

Organization of the manual

The *Rainbow Color/Graphics Option Programmer's Reference Guide* is subdivided into four parts containing fifteen chapters and three appendixes as follows:

- PART I — OPERATING PRINCIPLES contains the following four chapters:
  - Chapter 1 provides an overview of the Graphics Option including information on the hardware, logical interface to the CPU, general functionality, color and monochrome ranges, and model dependencies.
  - Chapter 2 describes the monitor configurations supported by the Graphics Option.
Preface

- Chapter 3 discusses the logic of data generation, bitmap addressing, and the GDC’s handling of the screen display.
- Chapter 4 describes the software components of the Graphics Option such as the control registers, maps, and buffer areas accessible under program control.

• PART II — PROGRAMMING GUIDELINES contains the following eight chapters:
  - Chapter 5 discusses programming the Graphics Option for initialization and control operations.
  - Chapter 6 discusses programming the Graphics Option for setting up bitmap write operations.
  - Chapter 7 discusses programming the Graphics Option for area write operations.
  - Chapter 8 discusses programming the Graphics Option for vector write operations.
  - Chapter 9 discusses programming the Graphics Option for text write operations.
  - Chapter 10 discusses programming the Graphics Option for read operations.
  - Chapter 11 discusses programming the Graphics Option for scroll operations.
  - Chapter 12 contains programming notes and timing considerations.

• PART III — REFERENCE MATERIAL contains the following three chapters:
  - Chapter 13 provides descriptions and contents of the Graphics Option’s registers, buffers, masks, and maps.
  - Chapter 14 provides descriptions and contents of the GDC’s status register and FIFO buffer.
  - Chapter 15 provides a description of each supported GDC command arranged in alphabetic sequence within functional grouping.

• PART IV — APPENDIXES contain the following three appendixes:
  - Appendix A contains the Graphics Option’s Specification Summary.
  - Appendix B is a fold-out sheet containing a block diagram of the Graphics Option.
  - Appendix C lists DIGITAL’s International Help Line phone numbers.

Suggestions for the Reader

For more information about the Graphics Display Controller refer to the following:

• uPD7220 GDC Design Manual—NEC Electronics U.S.A. Inc.
• uPD7220 GDC Design Specification—NEC Electronics U.S.A. Inc.

**Terminology**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU/PS</td>
<td>Arithmetic Logical Unit and Plane Select (register)</td>
</tr>
<tr>
<td>Bitmap</td>
<td>Video display memory</td>
</tr>
<tr>
<td>GDC</td>
<td>Graphics Display Controller</td>
</tr>
<tr>
<td>Motherboard</td>
<td>A term used to refer to the main circuit board where the processors and main</td>
</tr>
<tr>
<td></td>
<td>memory are located — hardware options, such as the Graphics Option, plug</td>
</tr>
<tr>
<td></td>
<td>into and communicate with the motherboard</td>
</tr>
<tr>
<td>Nibble</td>
<td>A term commonly used to refer to a half byte (4 bits)</td>
</tr>
<tr>
<td>Pixel</td>
<td>Picture element when referring to video display output</td>
</tr>
<tr>
<td>Resolution</td>
<td>A measure of the sharpness of a graphics image — usually given as the</td>
</tr>
<tr>
<td></td>
<td>number of addressable picture elements for some unit of length (pixels per</td>
</tr>
<tr>
<td></td>
<td>inch)</td>
</tr>
<tr>
<td>RGB</td>
<td>Red, green, blue — the acronym for the primary additive colors used in color</td>
</tr>
<tr>
<td></td>
<td>monitor displays</td>
</tr>
<tr>
<td>RGO</td>
<td>Rainbow Graphics Option</td>
</tr>
<tr>
<td>RMW</td>
<td>Read/Modify/Write, the action taken when accessing the bitmap during a write</td>
</tr>
<tr>
<td></td>
<td>or read cycle</td>
</tr>
<tr>
<td>VSS</td>
<td>Video Subsystem</td>
</tr>
</tbody>
</table>
Part I
Operating Principles
PART I

Chapter 1. Overview 1-1
   Hardware Components 1-1
      Video Memory (Bitmap) 1-2
   Additional Hardware 1-2
   Resolution Modes 1-3
      Medium Resolution Mode 1-3
      High Resolution Mode 1-3
   Operational Modes 1-3

Chapter 2. Monitor Configurations 2-1
   Monochrome Monitor Only 2-2
   Color Monitor Only 2-3
   Dual Monitors 2-4

Chapter 3. Graphics Option Logic 3-1
   General 3-1
   Data Logic 3-2
   Address Logic 3-2
   Display Logic 3-6
      Bitmap Logic 3-6
      Screen Logic 3-7
   GDC Command Logic 3-9
Chapter 4. Graphics Option Components  4–1

I/O Ports  4–1
Indirect Register  4–2
Write Buffer  4–2
Write Mask Registers  4–4
Pattern Generator  4–5
Foreground/Background Register  4–6
ALU/PS Register  4–8
Color Map  4–9
  Loading the Color Map  4–12
  Video Drive Voltages  4–13
Mode Register  4–15
Scroll Map  4–16
  Loading the Scroll Map  4–17
Hardware Components

The Graphics Option is a user-installable module that adds graphics and color display capabilities to the Rainbow system. The graphics module is based on a NEC uPD7220 Graphics Display Controller (GDC) and an $8 \times 64K$ dynamic RAM video memory that is also referred to as the bitmap.

The Graphics Option is supported, with minor differences, on Rainbow systems with either the model A or model B motherboard. The differences involve the number of colors and monochrome intensities that can be simultaneously displayed and the number of colors and monochrome intensities that are available to be displayed (see Table 1). Chapter 5 includes a programming example of how you can determine which model of the motherboard is present in your system.

Table 1. Colors and Monochrome Intensities — Displayed/Available

<table>
<thead>
<tr>
<th>CONFIG.</th>
<th>MED. RESOLUTION</th>
<th>HIGH RESOLUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MODEL</td>
<td>COLOR</td>
</tr>
<tr>
<td>MONOCHROME MONITOR ONLY</td>
<td>100-A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>100-B</td>
<td>N/A</td>
</tr>
<tr>
<td>COLOR MONITOR ONLY</td>
<td>100-A</td>
<td>16/1024</td>
</tr>
<tr>
<td></td>
<td>100-B</td>
<td>16/4096</td>
</tr>
<tr>
<td>DUAL MONITORS</td>
<td>100-A</td>
<td>16/4096</td>
</tr>
<tr>
<td></td>
<td>100-B</td>
<td>16/4096</td>
</tr>
</tbody>
</table>
The GDC, in addition to performing the housekeeping chores for the video display, can also:

- Draw lines at any angle
- Draw arcs of specified radii and length
- Fill rectangular areas
- Transfer character bit-patterns from font tables in main memory to the bitmap

**Video Memory (Bitmap)**

The CPUs on the motherboard have no direct access to the bitmap memory. All writes are performed by the external graphics option hardware to bitmap addresses generated by the GDC.

The bitmap is composed of eight 64K dynamic RAMs. This gives the bitmap a total of $8 \times 64K$ of display memory. In high resolution mode, this memory is configured as two planes, each $8 \times 32K$. In medium resolution mode, this memory is configured as four planes, each $8 \times 16K$. However, as far as the GDC is concerned, there is only one plane. All plane interaction is transparent to the GDC.

Although the bitmap is made up of $8 \times 64K$ bits, the GDC sees only 16K of word addresses in high resolution mode ($2 \text{ planes} \times 16 \text{ bits} \times 16K \text{ words}$). Similarly, the GDC sees only 8K of word addresses in medium resolution mode ($4 \text{ planes} \times 16 \text{ bits} \times 8K \text{ words}$). Bitmap address zero is displayed at the upper left corner of the monitor screen.

**Additional Hardware**

The option module also contains additional hardware that enhances the performance and versatility of the GDC. This additional hardware includes:

- A $16 \times 8$-bit Write Buffer used to store byte-aligned or word-aligned characters for high performance text writing or for fast block data moves from main memory to the bitmap
- An 8-bit Pattern Register and a 4-bit Pattern Multiplier for improved vector writing performance
- Address offset hardware ($256 \times 8$-bit Scroll Map) for full and split-screen vertical scrolling
- ALU/PS register to handle bitplane selection and the write functions of Replace, Complement, and Overlay
- A $16 \times 16$-bit Color Map to provide easy manipulation of pixel color and monochrome intensities
- Readback hardware for reading a selected bitmap memory plane into main memory
Resolution Modes

The Graphics Option operates in either of two resolution modes:

- Medium Resolution Mode
- High Resolution Mode

Medium Resolution Mode

Medium resolution mode displays 384 pixels horizontally by 240 pixels vertically by four bitmap memory planes deep. This resolution mode allows up to 16 colors to be simultaneously displayed on a color monitor. Up to sixteen monochrome shades can be displayed simultaneously on a monochrome monitor.

High Resolution Mode

High resolution mode displays 800 pixels horizontally by 240 pixels vertically by two bitmap memory planes deep. This mode allows up to four colors to be simultaneously displayed on a color monitor. Up to four monochrome shades can be simultaneously displayed on a monochrome monitor.

Operational Modes

The Graphics Option supports the following modes of operations:

- WORD MODE to write 16-bit words to selected planes of the bitmap memory for character and image generation
- VECTOR MODE to write pixel data to bitmap addresses provided by the GDC
- SCROLL MODE for full- and split-screen vertical scrolling and full-screen horizontal scrolling
- READBACK MODE to read 16-bit words from a selected plane of bitmap memory for special applications, hardcopy generation or diagnostic purposes
In the Rainbow system with the Graphics Option installed, there are three possible monitor configurations: Monochrome only, Color only, and Dual (color and monochrome). In all three configurations, the selection of the option’s monochrome output or the motherboard VT102 video output is controlled by bit two of the system maintenance port (port 0Ah). A 0 in bit 2 selects the motherboard VT102 video output while a 1 in bit 2 selects the option’s monochrome output.
Monochrome Monitor Only

As shown in Figure 1, the monochrome monitor can display either graphics option data or motherboard data depending on the setting of bit 2 of port 0Ah. Writing an 87h to port 0Ah selects the Graphics Option data. Writing an 83h to port 0Ah selects the motherboard VT102 data. The red, green and blue data areas in the Color Map should be loaded with all F’s to reduce any unnecessary radio frequency emissions.

---

Figure 1. Monochrome Monitor Only System
Color Monitor Only

When the system is configured with only a color monitor, as in Figure 2, the green gun does double duty. It either displays the green component of the graphics output or it displays the monochrome output of the motherboard VT102 video subsystem. Because the green gun takes monochrome intensities, all green intensities must be programmed into the monochrome data area of the Color Map. The green data area of the Color Map should be loaded with all F’s to reduce any unnecessary radio frequency emissions.

When motherboard VT102 data is being sent to the green gun, the red and blue output must be turned off at the Graphics Option itself. If not, the red and blue guns will continue to receive data from the option and this output will overlay the motherboard VT102 data and will also be out of synchronization. Bit 7 of the Mode Register is the graphics option output enable bit. If this bit is a 1 red and blue outputs are enabled. If this bit is a 0 red and blue outputs are disabled.

As in the monochrome only configuration, bit 2 of port 0Ah controls the selection of either the graphics option data or the motherboard VT102 data. Writing an 87h to port 0Ah enables the option data. Writing an 83h to port 0Ah selects the motherboard VT102 data.
Dual Monitors

In the configuration shown in Figure 3, both a color monitor and a monochrome monitor are available to the system. Motherboard VT102 video data can be displayed on the monochrome system while color graphics are being displayed on the color monitor. If the need should arise to display graphics on the monochrome monitor, the monochrome intensity output can be directed to the monochrome monitor by writing an 87h to port 0Ah. Writing an 83h to port 0Ah will restore motherboard VT102 video output to the monochrome monitor.

When displaying graphics on the monochrome monitor, the only difference other than the lack of color is the range of intensities that can be simultaneously displayed on systems with model A motherboards.

Systems with model A motherboards can display only four monochrome intensities at any one time. Even though sixteen entries can be selected when operating in medium resolution mode, only the two low-order bits of the monochrome output are active. This limits the display to only four unique intensities at most. On systems with the model B motherboard, all sixteen monochrome intensities can be displayed.
General

The Graphics Display Controller (GDC) can operate either on one bit at a time or on an entire 16-bit word at a time. It is, however, limited to one address space and therefore can only write into one plane at a time. The Graphics Option is designed in such a manner that while the GDC is doing single pixel operations on just one video plane, the external hardware can be doing 16-bit word operations on up to four planes of video memory.

Write operations are multi-dimensional. They have width, depth, length and time.

- Width refers to the number of pixels involved in the write operation.
- Depth refers to the number of planes involved in the write operation.
- Length refers to the number of read/modify/write cycles the GDC is programmed to perform.
- Time refers to when the write operation occurs in relation to the normal housekeeping operations the GDC has to perform in order to keep the monitor image stable and coherent.
Data Logic

The Graphics Option can write in two modes: word mode (16 bits at a time) and vector mode (one pixel at a time).

In word mode, the data patterns to be written into the bitmap are based on bit patterns loaded into the Write Buffer, Write Mask, and the Foreground/Background Register, along with the type of write operation programmed into the ALU/PS Register.

In vector mode, the data patterns to be written to the bitmap are based on bit patterns loaded into the Pattern Register, the Pattern Multiplier, the Foreground/Background Register, and the type of write operation programmed into the ALU/PS Register.

In either case, the data will be stored in the bitmap at a location determined by the addressing logic.

Address Logic

The addressing logic of the Graphics Option is responsible for coming up with the plane, the line within the plane, the word within the line, and even the pixel within the word under some conditions.

The display memory on the Graphics Option is one-dimensional. The GDC scans this linear memory to generate the two dimensional display on the CRT. The video display is organized similarly to the fourth quadrant of the Cartesian plane with the origin in the upper left corner. Row addresses (y coordinates of pixels) start at zero and increase downwards while column addresses (x coordinates of pixels) start at zero and increase to the right (see Figure 4). Pixel data is stored in display memory by column within row.
The GDC accesses the display memory as a number of 16-bit words where each bit represents a pixel. The number of words defined as well as the number of words displayed on each line is dependent on the resolution. The relationship between words and display lines is shown in Figure 5.
Graphics Option Logic

Figure 5. Relationship of Display Memory to Address Logic

<table>
<thead>
<tr>
<th>LINE 0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>Q-1</th>
<th>P-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE 1</td>
<td>P</td>
<td>P+1</td>
<td>P+2</td>
<td>P+Q-1</td>
<td>2P-1</td>
</tr>
<tr>
<td>LINE 2</td>
<td>2P</td>
<td>2P+1</td>
<td></td>
<td>2P+Q-1</td>
<td>3P-1</td>
</tr>
<tr>
<td></td>
<td>3P</td>
<td></td>
<td></td>
<td>3P+Q-1</td>
<td>4P-1</td>
</tr>
<tr>
<td></td>
<td>4P</td>
<td></td>
<td></td>
<td>4P+Q-1</td>
<td>5P-1</td>
</tr>
<tr>
<td></td>
<td>(M-1)P</td>
<td></td>
<td></td>
<td></td>
<td>MP-1</td>
</tr>
<tr>
<td>LINE N-1</td>
<td>(N-1)P</td>
<td></td>
<td></td>
<td></td>
<td>NP-1</td>
</tr>
</tbody>
</table>

WHERE:

\[ P = \text{WORDS/LINE DEFINED} \]
- 32 in MEDIUM RESOLUTION.
- 64 in HIGH RESOLUTION.

\[ Q = \text{WORDS/LINE DISPLAYED} \]
- 24 in MEDIUM RESOLUTION
- 50 in HIGH RESOLUTION

\[ N = \text{NO. OF LINES DEFINED} \]
- 256

\[ M = \text{NO. OF LINES DISPLAYED} \]
- 240
In order to address specific pixels, the GDC requires the word address and the pixel location within that word. The conversion of pixel coordinates to addresses in display memory is accomplished by the following formulas:

Given the pixel coordinates \((x,y)\):

Word Address of pixel = \((\text{words/line defined} \times y) + \text{integer}(x/16)\)

Pixel Address within word = remainder\((x/16)\) \(* 16\)

Because the Graphics Option is a multi-plane device, a way is provided to selectively enable and disable the reading and writing of the individual planes. This function is performed by the ALU/PS and Mode registers. More than one plane at a time can be enabled for a write operation; however, only one plane can be enabled for a read operation at any one time.

The entire address generated by the GDC does not go directly to the bitmap. The low-order six bits address a word within a line in the bitmap and do go directly to the bitmap. The high-order eight bits address the line within the plane and these bits are used as address inputs to a Scroll Map. The Scroll Map acts as a translator such that the bitmap location can be selectively shifted in units of 64 words. In high resolution mode, 64 words equate to one scan line; in medium resolution mode, they equate to two scan lines. This allows the displayed vertical location of an image to be moved in 64-word increments without actually rewriting it to the bitmap. Programs using this feature can provide full and split screen vertical scrolling. The Scroll Map is used in all bitmap access operations: writing, reading, and refreshing.

If an application requires addressing individual pixels within a word, the two 8-bit Write Mask Registers can be used to provide a 16-bit mask that will write-enable selected pixels. Alternately, a single pixel vector write operation can be used.

There is a difference between the number of words/line defined and the number of words/line displayed. In medium resolution, each scan line is 32 words long but only 24 words are displayed \((24 \times 16 \text{ bits/word} = 384 \text{ pixels})\). The eight words not displayed are unusable. Defining the length of the scan line as 24 words would be a more efficient use of memory but it would take longer to refresh the memory. Because display memory is organized as a 256 by 256 array, it takes 256 bytes of scan to refresh the entire 64K byte memory. Defining the scan line length as 32 words long enables the entire memory to be refreshed in four line scan periods. Defining the scan line length as 24 words long would require five line scans plus 16 bytes.

Similarly, in high resolution, each scan line is 64 words long but only 50 words are displayed. With a 64 word scan line length, it takes two line scan periods to refresh the entire 64K byte memory. If the scan line length were 50 words, it would take two lines plus 56 bytes to refresh the memory.

Another advantage to defining scan line length as 32 or 64 words is that cursor locating can be accomplished by a series of shift instructions which are considerably faster than multiplying.
Display Logic

The display logic of the Graphics Option will be discussed as it applies to both the bitmap and the screen.

Bitmap Logic

Data in the bitmap does not go directly to the monitor. Instead, the bitmap data is used as an address into a Color Map. The output of this Color Map, which has been preloaded with color and monochrome intensity values, is the data that is sent to the monitor.

In medium resolution mode there are four planes to the bitmap; each plane providing an address bit to the Color Map. Four bits can address sixteen unique locations at most. This gives a maximum of 16 addressable Color Map entries. Each Color Map entry is 16 bits wide. Four of the bits are used to drive the color monitor’s red gun, four go to the green gun, four go to the blue gun, and four drive the output to the monochrome monitor. In systems with the Model 100-A motherboard, only the two low-order bits of the monochrome output are used. Therefore, although there are 16 possible monochrome selections in the Color Map, the number of unique intensities that can be sent to the monochrome monitor is four.

In high resolution mode there are two planes to the bitmap; each plane providing an address bit to the Color Map. Two bits can address four entries in the Color Map at most. Again, each Color Map entry is sixteen bits wide with 12 bits of information used for color and four used for monochrome shades. In systems with the Model 100-A motherboard, only the two low-order bits of the monochrome output are used. This limits the number of unique monochrome intensities to four.

Although the Color Map is 16 bits wide, the color intensity values are loaded one byte at a time. First, the 16 pairs of values representing the red and green intensities are loaded into bits 0 through 7 of the map. Then, the 16 pairs of values representing the blue and monochrome intensities are loaded into bits 8 through 15 of the map.
Screen Logic

The image displayed on the screen is generated by an electron beam performing a series of horizontal line scans from left to right. At the end of each horizontal scan line, a horizontal retrace takes place at which time the electron beam reverses its horizontal direction. During this horizontal retrace, the electron beam is also being moved down to the beginning of the next scan line. When the last line has completed its horizontal retrace, a vertical retrace takes place at which time the electron beam's vertical movement is reversed and the beam is positioned at the beginning of the first scan line.

The GDC writes to the bitmap only during the screen's horizontal and vertical retrace periods. During active screen time, the GDC is taking information out of the bitmap and presenting it to the video screen hardware. For example, if the GDC is drawing a vector to the bitmap, it will stop writing during active screen time and resume writing the vector at the next horizontal or vertical retrace.

In addition to the active screen time and the horizontal and vertical retrace times, there are several other screen control parameters that precede and follow the active horizontal scans and active lines. These are the Vertical Front and Back Porches and the Horizontal Front and Back Porches. The relationship between the screen control parameters is shown in Figure 6. Taking all the parameters into account, the proportion of active screen time to bitmap writing time is approximately four to one.
Figure 6. GDC Screen Control Parameters
GDC Command Logic

Commands are passed to the GDC command processor from the Rainbow system by writing command bytes to port 57h and parameter bytes to port 56h. Data written to these two ports is stored in the GDC’s FIFO buffer, a 16 x 9-bit area that is used to both read from and write to the GDC. The FIFO buffer operates in half-duplex mode — passing data in both directions, one direction at a time. The direction of data flow at any one time is controlled by GDC commands.

When commands are stored in the FIFO buffer, a flag bit is associated with each data byte depending on whether the data byte was written to the command address (57h) or the parameter address (56h). A flag bit of 1 denotes a command byte; a flag bit of 0 denotes a parameter byte. The command processor tests this flag bit as it interprets the contents of the FIFO buffer.

The receipt of a command byte by the command processor signifies the end of the previous command and any associated parameters. If the command is one that requires a response from the GDC such as RDAT, the FIFO buffer is automatically placed into read mode and the buffer direction is reversed. The specified data from the bitmap is loaded into the FIFO buffer and can be accessed by the system using read operations to port 57h. Any commands or parameters in the FIFO buffer that follow the read command are lost when the FIFO buffer’s direction is reversed.

When the FIFO buffer is in read mode, any command byte written to port 57h will immediately terminate the read operation and reverse the buffer direction to write mode. Any data that has not been read by the Rainbow system from the FIFO buffer will be lost.
The CPUs on the Rainbow system’s motherboard use a number of 8-bit I/O ports to exchange information with the various subsystems and options. The I/O ports assigned to the Graphics Option are ports 50h through 57h. They are used to generate and display graphic images, inquire status, and read the contents of video memory (bitmap). The function of each of the Graphics Option’s I/O ports is as follows:

<table>
<thead>
<tr>
<th>Port</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>50h</td>
<td>Graphics option software reset. Any write to this port also resynchronizes the read/modify/write memory cycles of the Graphics Option to those of the GDC.</td>
</tr>
<tr>
<td>51h</td>
<td>Data written to this port is loaded into the area selected by the previous write to port 53h.</td>
</tr>
<tr>
<td>52h</td>
<td>Data written to this port is loaded into the Write Buffer.</td>
</tr>
<tr>
<td>53h</td>
<td>Data written to this port provides address selection for indirect addressing (see Indirect Register).</td>
</tr>
<tr>
<td>54h</td>
<td>Data written to this port is loaded into the low-order byte of the Write Mask.</td>
</tr>
<tr>
<td>55h</td>
<td>Data written to this port is loaded into the high-order byte of the Write Mask.</td>
</tr>
<tr>
<td>56h</td>
<td>Data written to this port is loaded into the GDC’s FIFO Buffer and flagged as a parameter. Data read from this port reflects the GDC status.</td>
</tr>
<tr>
<td>57h</td>
<td>Data written to this port is loaded into the GDC’s FIFO Buffer and flagged as a command. Data read from this port reflects information extracted from the bitmap.</td>
</tr>
</tbody>
</table>
Indirect Register

The Graphics Option uses indirect addressing to enable it to address more registers and storage areas on the option module than there are address lines (ports) to accommodate them. Indirect addressing involves writing to two ports. A write to port 53h loads the Indirect Register with a bit array in which each bit selects one of eight areas.

The Indirect Register bits and the corresponding areas are as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Area Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Write Buffer (*)</td>
</tr>
<tr>
<td>1</td>
<td>Pattern Multiplier</td>
</tr>
<tr>
<td>2</td>
<td>Pattern Register</td>
</tr>
<tr>
<td>3</td>
<td>Foreground/Background Register</td>
</tr>
<tr>
<td>4</td>
<td>ALU/PS Register</td>
</tr>
<tr>
<td>5</td>
<td>Color Map (*)</td>
</tr>
<tr>
<td>6</td>
<td>Mode Register</td>
</tr>
<tr>
<td>7</td>
<td>Scroll Map (*)</td>
</tr>
</tbody>
</table>

(*) Also clears the associated index counter

After selecting an area by writing to port 53h, you access and load data into most selected areas by writing to port 51h. For the Write Buffer however, you need both a write of anything to port 51h to access the buffer and clear the counter and then a write to port 52h to load the data.

Write Buffer

A 16 × 8-bit Write Buffer provides the data for the bitmap when the Graphics Option is in Word Mode. You can use the buffer to transfer blocks of data from the system’s memory to the bitmap. The data can be full screen images of the bitmap or bit-pattern representations of font characters that have been stored in motherboard memory. The buffer has an associated index counter that is cleared when the Write Buffer is selected.

Although the CPU accesses the Write Buffer as sixteen 8-bit bytes, the GDC accesses the buffer as eight 16-bit words. (See Figure 7.) A 16-bit Write Mask gives the GDC control over individual bits of a word.
The output of the Write Buffer is the inverse of its input. If a word is written into the buffer as FFB6h, it will be read out of the buffer as 0049h. To have the same data written out to the bitmap as was received from the CPU requires an added inversion step. You can exclusive or (XOR) the CPU data with FFh to pre-invert the data before going through the Write Buffer. Alternately, you can write zeros into the Foreground Register and ones into the Background Register to re-invert the data after it leaves the Write Buffer and before it is written to the bitmap. Use one method or the other, not both.

In order to load data into the Write Buffer, you first write an FEh to port 53h and any value to port 51h. This not only selects the Write Buffer but also sets the Write Buffer Index Counter to zero. The data is then loaded into the buffer by writing it to port 52h in high-byte low-byte order. If more than 16 bytes are written to the buffer the first 16 bytes will be overwritten.

If you load the buffer with less than 16 bytes (or other than a multiple of 16 bytes for some reason or other) the GDC will find an index value other than zero in the counter. Starting at a location other than zero alters the data intended for the bitmap. Therefore, before the GDC is given the command to write to the bitmap, you must again clear the Write Buffer Index Counter so that the GDC will start accessing the data at word zero.
Write Mask Registers

When the Graphics Option is in Word Mode, bitmap operations are carried out in units of 16-bit words. A 16-bit Write Mask controls the writing of individual bits within a word. A zero in a bit position of the mask allows writing to the corresponding position of the word. A one in a bit position of the mask disables writing to the corresponding position of the word.

While the GDC accesses the mask as a 16-bit word, the CPU accesses the mask as two of the Graphic Option's I/O ports. The high-order Write Mask Register is loaded with a write to port 55h and corresponds to bits 15 through 8 of the Write Mask. The low-order Write Mask Register is loaded with a write to port 54h and corresponds to bits 7 through 0 of the Write Mask. (See Figure 8.)

![Figure 8. Write Mask Registers](image)
Pattern Generator

When the Graphics Option is in Vector Mode, the Pattern Generator provides the data to be written to the bitmap. The Pattern Generator is composed of a Pattern Register and a Pattern Multiplier.

The Pattern Register is an 8-bit recirculating shift register that is first selected by writing FBh to port 53h and then loaded by writing an 8-bit data pattern to port 51h.

The Pattern Multiplier is a 4-bit register that is first selected by writing FDh to port 53h and then loaded by writing a value of 0-Fh to port 51h.

**NOTE**
You must load the Pattern Multiplier before loading the Pattern Register.

Figure 9 shows the logic of the Pattern Generator. Data destined for the bitmap originates from the low-order bit of the Pattern Register. That same bit continues to be the output until the Pattern Register is shifted. When the most significant bit of the Pattern Register has completed its output cycle, the next bit to shift out will be the least significant bit again.
The shift frequency is the write frequency from the option clock divided by 16 minus the value in the Pattern Multiplier. For example, if the value in the Pattern Multiplier is 12, the shift frequency divisor would be 16 minus 12 or four. The shift frequency would be one fourth of the write frequency and therefore each bit in the Pattern Register would be replicated in the output stream four times. A multiplier of 15 would take 16 – 15 or one write cycle for each Pattern Register bit shifted out. A multiplier of five would take 16 – 5 or 11 write cycles for each bit in the Pattern Register.

**NOTE**

Do not change the contents of the Pattern Multiplier or the Pattern Register before the GDC has completed all pending vector mode write operations. If you do, the vector pattern that is in the process of being displayed will take on the new characteristics of the Pattern Generator.

**Foreground/Background Register**

The Foreground/Background Register is an eight-bit write-only register. The high-order nibble is the Foreground Register; the low-order nibble is the Background Register. Each of the four bitmap planes has a Foreground/Background bit-pair associated with it (see Figure 10). The bit settings in the Foreground/Background Register, along with the mode specified in the ALU/PS Register, determine the data that is eventually received by the bitmap. For example; if the mode is REPLACE, an incoming data bit of 0 is replaced by the corresponding bit in the Background Register. If the incoming data bit is a 1, the bit would be replaced by the corresponding bit in the Foreground Register.

Each bitmap plane has its own individual Foreground/Background bit pair. Therefore, it is possible for two enabled planes to use the same incoming data pattern and end up with different bitmap patterns.
NOTE
Do not change the contents of the Foreground/Background Register before the GDC has completed all pending write operations. If you do, the information that is in the process of being displayed will take on the new values of the Foreground/Background Register.

Figure 10. Foreground/Background Register
ALU/PS Register

The ALU/PS Register has two functions.

Bits 0 through 3 of the ALU/PS Register are used to inhibit writes to one or more of the bitmap planes. If you could not inhibit writes to the bitmap planes, each write operation would affect all available planes. When a plane select bit is set to 1, writes to that plane will be inhibited. When a plane select bit is set to 0, writes to that plane will be allowed.

**NOTE**
During a readback mode operation, all plane select bits should be set to ones to prevent accidental changes to the bitmap data.

Bits 4 and 5 of the ALU/PS Register define an arithmetic logic unit function. The three logic functions supported by the option are REPLACE, COMPLEMENT, and OVERLAY. These functions operate on the incoming data from the Write Buffer or the Pattern Generator as modified by the Foreground/Background Register as well as the current data in the bitmap and generate the new data to be placed into the bitmap.

When the logic unit is operating in REPLACE mode, the current data in the bitmap is replaced by the Foreground/Background data selected as follows:

- An incoming data bit 0 selects the Background data.
- An incoming data bit 1 selects the Foreground data.

When the logic unit is operating in COMPLEMENT mode, the current data in the bitmap is modified as follows:

- An incoming data bit 0 results in no change.
- An incoming data bit 1 results in the current data being exclusively or'ed (XOR) with the appropriate Foreground bit. If the Foreground bit is 0, the current data is unchanged. If the Foreground bit is 1, the current data is complemented by binary inversion. In effect, the Foreground Register acts as a plane select register for the complement operation.
When the logic unit is operating in OVERLAY mode, the current data in the bitmap is modified as follows:

- An incoming data bit 0 results in no change.
- An incoming data bit 1 results in the current data being replaced by the appropriate Foreground bit.

**NOTE**
Do not change the contents of the ALU/PS Register before the GDC has completed all pending write operations. If you do, the information that is in the process of being displayed will take on the new characteristics of the ALU/PS Register.

**Color Map**

The Color Map is a 16 × 16-bit RAM area where each of the 16 entries is composed of four 4-bit values representing color intensities. These values represent, from high order to low order, the monochrome, blue, red, and green outputs to the video monitor. Intensity values are specified in inverse logic. At one extreme, a value of zero represents maximum intensity (100% output) for a particular color or monochrome shade. At the other extreme, a value of 0Fh represents minimum intensity (zero output).

Bitmap data is not directly displayed on the monitor, each bitmap plane contributes one bit to an index into the Color Map. The output of the Color Map is the data that is passed to the monitor. Four bitmap planes (medium resolution) provide four bits to form an index allowing up to 16 intensities of color or monochrome to be simultaneously displayed on the monitor. Two bitmap planes (high resolution) provide two bits to form an index allowing only four intensities of color or monochrome to be simultaneously displayed on the monitor.
In Figure 11, a medium resolution configuration, the bitmap data for the display point \(x, y\) is 0110b. This value, when applied as an index into the Color Map, selects the seventh entry out of a possible sixteen. Each Color Map entry is sixteen bits wide. Four of the bits are used to drive the color monitor’s red gun, four go to the green gun, four go to the blue gun, and four drive the output to the monochrome monitor. The twelve bits going to the color monitor support a color palette of 4096 colors; the four bits to the monochrome monitor support 16 shades. (In systems with the Model 100-A motherboard, only the two low-order bits of the monochrome output are active. This limits the monochrome output to four unique intensities.)

\[\text{BITMAP} \to \text{COLOR MAP}\]

\(0110b\)

4\(^(*)\) BITS OF MONOCROME LEVEL TO MONO. MONITOR
4 BITS OF BLUE LEVEL TO COLOR MONITOR
4 BITS OF RED LEVEL TO COLOR MONITOR
4 BITS OF GREEN LEVEL TO COLOR MONITOR

\(\text{(*) 2 LOW-ORDER BITS ON MODEL 100-A SYSTEMS}\)

Figure 11. Bitmap/Color Map Interaction (medium resolution)
In Figure 12, a high resolution configuration, the bitmap data for point \((x,y)\) is \(10b\). This value, when applied as an index into the Color Map, selects the third entry out of a possible four. Again, each Color Map entry is sixteen bits wide; 12 bits of information are used for color and four are used for monochrome. (In systems with the Model 100-A motherboard, only the two low-order bits of the monochrome output are active. This limits the monochrome output to four unique intensities.)

\((*)\) 2 LOW-ORDER BITS ON MODEL 100-A SYSTEMS

Figure 12. Bitmap/Color Map Interaction (high resolution)
Loading the Color Map

The Graphics Option accesses the Color Map as sixteen 16-bit words. However, the CPU accesses the Color Map as 32 eight-bit bytes. The 32 bytes of intensity values are loaded into the Color Map one entire column of 16 bytes at a time. The red and green values are always loaded first, then the monochrome and blue values. (See Figure 13.)

<table>
<thead>
<tr>
<th>ADDRESS VALUE</th>
<th>MONO. DATA</th>
<th>BLUE DATA</th>
<th>RED DATA</th>
<th>GREEN DATA</th>
<th>COLOR DISPLAYED</th>
<th>MONOCHROME DISPLAYED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>BLACK</td>
<td>BLACK</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>15</td>
<td>0</td>
<td>15</td>
<td>RED</td>
<td>•</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>GREEN</td>
<td>•</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>0</td>
<td>15</td>
<td>15</td>
<td>BLUE</td>
<td>•</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>MAGENTA</td>
<td>•</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>CYAN</td>
<td>•</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>YELLOW</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>WHITE</td>
<td>WHITE</td>
</tr>
</tbody>
</table>

Figure 13. Sample Color Map With Loading Sequence
Writing the value DFh to port 53h selects the Color Map and also clears the Color Map Index Counter to zero. To load data into the Color Map requires writing to port 51h. Each write to port 51h will cause whatever is on the motherboard data bus to be loaded into the current Color Map location. After each write, the Color Map Index Counter is incremented by one. If 33 writes are made to the Color Map, the first Color Map location will be overwritten.

**NOTE**
Do not change the contents of the Color Map before the GDC has completed all pending write operations. If you do, the information that is in the process of being displayed will take on the new Color Map characteristics.

**Video Drive Voltages**

The output of the Color Map, as shown in Figures 11 and 12, consists of four 4-bit values that represent the red, green, blue, and monochrome intensities to be displayed on some applicable monitor. These four intensity values are the input to four digital-to-analog converters. (Refer to the block diagram in Appendix B.) The output of these converters are the video drive voltages that are applied to pins 9 through 12 of the J3 Video Output Jack.

The output of the digital-to-analog converters for the red, green, and blue intensities is not dependent on the model of the system motherboard. The digital-to-analog converter for the monochrome intensities, however, produces different output depending on whether the motherboard is a model A or a model B. On systems with a model A motherboard, only the two low-order bits of the intensity value are active. This provides a limited range of only four output voltages for the monochrome signal. On a color monitor only configuration, where the green output is derived from the monochrome portion of the Color Map, the same limited range applies. On systems with a model B motherboard, all four bits of the intensity value are active. This provides the full range of 16 output voltages for the red, green, blue, and monochrome signals. The conversion of Color Map intensity values to video drive voltages for each of these ranges are shown in Table 2.

The perceived intensity of a display is not linearly related to the video drive voltages. A given difference in drive voltage at the high end of the range is not as noticeable as the same difference occurring at the low end of the range.
### Table 2. Intensity Values vs Video Drive Voltages

<table>
<thead>
<tr>
<th>HEX</th>
<th>BINARY</th>
<th>LIMITED RANGE</th>
<th>FULL RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>1.09</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0.79</td>
<td>0.85</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0.71</td>
<td>0.79</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0.09</td>
<td>0.73</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>1.09</td>
<td>0.67</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>0.79</td>
<td>0.61</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>0.71</td>
<td>0.55</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0.09</td>
<td>0.49</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1.09</td>
<td>0.43</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>0.79</td>
<td>0.38</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>0.71</td>
<td>0.31</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>0.09</td>
<td>0.26</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>1.09</td>
<td>0.21</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>0.79</td>
<td>0.12</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>0.71</td>
<td>0.07</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>0.09</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**LIMITED RANGE:** MODEL A — ALL MONOCHROME OUTPUT
- GREEN OUTPUT ON COLOR MONITOR ONLY SYSTEM

**FULL RANGE:** MODEL A — RED/BLUE OUTPUT ON COLOR MONITOR ONLY SYSTEM
- RED/GREEN/BLUE OUTPUT ON DUAL MONITOR SYSTEM
MODEL B — RED/BLUE/GREEN/MONOCHROME OUTPUT ON ALL SYSTEMS
Mode Register

The Mode Register is an 8-bit multi-purpose register that is loaded by first selecting it with a write of BFh to port 53h and then writing a data byte to port 51h. The bits in the Mode Register have the following functions:

- Bit 0 determines the resolution mode:
  - 0 = medium resolution mode (384 pixels across)
  - 1 = high resolution mode (800 pixels across)

- Bit 1 determines the write mode:
  - 0 = word mode, 16 bits/RMW cycle, data from Write Buffer
  - 1 = vector mode, 1 bit/RMW cycle, data from Pattern Generator

- Bits 3 and 2 select a bitmap plane for readback mode operation:
  - 00 = plane 0
  - 01 = plane 1
  - 10 = plane 2
  - 11 = plane 3

- Bit 4 determines the option's mode of operation:
  - 0 = read mode, bits 3 and 2 determine readback plane
  - 1 = write mode, writes to the bitmap allowed but not mandatory

- Bit 5 controls writing to the Scroll Map:
  - 0 = writing is enabled (after selection by the Indirect Register)
  - 1 = writing is disabled

- Bit 6 controls the interrupts to the CPU generated by the Graphics Option every time the GDC issues a vertical sync pulse:
  - 0 = interrupts are disabled, any pending interrupts are cleared
  - 1 = interrupts are enabled

- Bit 7 controls the video data output from the option:
  - 0 = output is disabled, other option operations still take place
  - 1 = output is enabled

NOTE
Do not change the contents of the Mode Register before the GDC has completed all pending write operations. If you do, the functions controlled by the Mode Register will take on the new characteristics and the results may be indeterminate.
Scroll Map

The Scroll Map is a $256 \times 8$-bit recirculating ring buffer that is used to offset scan line addresses in the bitmap in order to provide full and split-screen vertical scrolling. The entire address as generated by the GDC does not go directly to the bitmap. Only the low-order six bits of the GDC address go directly to the bitmap. They represent one of the 64 word addresses that are the equivalent of one scan line in high resolution mode or two scan lines in medium resolution mode. The eight high-order bits of the GDC address represent a line address and are used as an index into the 256-byte Scroll Map. The eight bits at the selected location then become the new eight high-order bits of the address that the bitmap sees. (See Figure 14.) By manipulating the contents of the Scroll Map, you can perform quick dynamic relocations of the bitmap data in 64-word blocks.

![Figure 14. Scroll Map Operation](image-url)
Loading the Scroll Map

Start loading the offset addresses into the Scroll Map at the beginning of a vertical retrace. First set bit 5 of the Mode Register to zero to enable the Scroll Map for writing. Write a 7Fh to port 53h to select the Scroll Map and clear the Scroll Map Index Counter to zero. Then do a series of writes to port 51h with the offset values to be stored in the Scroll Map. Loading always begins at location zero of the Scroll Map. With each write, the Scroll Map Index Counter is automatically incremented until the write operations terminate. If there are more than 256 writes, the index counter loops back to Scroll Map location zero. This also means that if line 255 requires a change, lines 0-254 will have to be rewritten first.

All 256 scroll map entries should be defined even if all 256 addresses are not displayed. This is to avoid mapping undesirable data onto the screen. After the last write operation, bit 5 of the Mode Register should be set to one to disable further writing to the Scroll Map.

The time spent to load the Scroll Map should be kept as short as possible. During loading, the GDC’s address lines no longer have a path to the bitmap and therefore memory refresh is not taking place. Delaying memory refresh can result in lost data.

While it is possible to read out of the Scroll Map, time constraints preclude doing both a read and a rewrite during the same vertical retrace period. If necessary, a shadow image of the Scroll Map can be kept in some area in memory. The shadow image can be updated at any time and then transferred into the Scroll Map during a vertical retrace.
Part II
Programming Guidelines
PART II

Chapter 5. Initialization and Control  5-1
  Test for Option Present  5-1
  Example of Option Test  5-1
  Test for Motherboard Version  5-2
    Example of Version Test for CP/M System  5-2
    Example of Version Test for MS-DOS System  5-3
    Example of Version Test for Concurrent CP/M System  5-5
  Initialize the Graphics Option  5-6
  Reset the GDC  5-6
  Initialize the GDC  5-7
  Initialize the Graphics Option  5-8
  Example of Initializing the Graphics Option  5-9
  Controlling Graphics Output  5-24
    Example of Enabling a Single Monitor  5-24
    Example of Disabling a Single Monitor  5-25
  Modifying and Loading the Color Map  5-25
    Example of Modifying and Loading Color Data in a Shadow Map  5-26

Chapter 6. Bitmap Write Setup (General)  6-1
  Loading the ALU/PS Register  6-1
    Example of Loading the ALU/PS Register  6-1
  Loading the Foreground/Background Register  6-2
    Example of Loading the Foreground/Background Register  6-2
Chapter 7. Area Write Operations  7-1
Display Data from Memory     7-1
  Example of Displaying Data from Memory     7-1
Set a Rectangular Area to a Color     7-4
  Example of Setting a Rectangular Area to a Color     7-4

Chapter 8. Vector Write Operations  8-1
Setting Up the Pattern Generator     8-1
  Example of Loading the Pattern Register     8-1
  Example of Loading the Pattern Multiplier     8-3
Display a Pixel     8-4
  Example of Displaying a Single Pixel     8-4
Display a Vector     8-5
  Example of Displaying a Vector     8-6
Display a Circle     8-9
  Example of Drawing a Circle     8-9

Chapter 9. Text Write Operations  9-1
Write a Byte-Aligned Character     9-1
  Example of Writing a Byte-Aligned Character     9-1
Define and Position the Cursor     9-32
  Example of Defining and Positioning the Cursor     9-32
Write a Text String     9-38
  Example of Writing a Text String     9-38

Chapter 10. Read Operations  10-1
The Read Process     10-1
Read the Entire Bitmap     10-1
  Example of Reading the Entire Bitmap     10-2
Pixel Write After a Read Operation     10-5

Chapter 11. Scroll Operations  11-1
Vertical Scrolling     11-1
  Example of Vertical Scrolling One Scan Line     11-2
Horizontal Scrolling     11-4
  Example of Horizontal Scrolling One Word     11-4

Chapter 12. Programming Notes  12-1
Shadow Areas     12-1
Bitmap Refresh     12-1
Software Reset     12-2
Setting Up Clock Interrupts     12-2
Operational Requirements     12-3
Set-Up Mode     12-3
Timing Considerations     12-4
The examples in this chapter cover the initialization of the Graphics Display Controller (GDC) and the Graphics Option, the control of the graphics output, and the management of the option's color palette.

Test for Option Present

Before starting any application, you should ensure that the Graphics Option has been installed on the Rainbow system. Attempting to use the Graphics Option when it is not installed can result in a system reset that can in turn result in the loss of application data. The following code will test for the option's presence.

Example of Option Test

```
;******************************************************************************
; procedure option_present_test
;
; purpose: test if Graphics Option is present.
; entry: none.
; exit: dl = 1 option present.
; dl = 0 option not present.
; register usage: ax,dx

;******************************************************************************
```
### Initialization and Control

```assembly
    cseg     segment byte public 'codesg'
    public  option_present_test
    assume cs:cseg,ds:nothing,es:nothing,ss:nothing
    option_present_test proc near
    mov     dl,1      ;set dl for option present
    in      al,8      ;input from port 8
    test    al,04h   ;test bit 2 to see if option present
    jz      opt1      ;if option is present, exit
    xor     dl,dl     ;else, set dl for option not present
    opt1: ret
    option_present_test endp
    cseg    ends
    end
```

### Test for Motherboard Version

When you initially load or subsequently modify the Color Map, it is necessary to know what version of the motherboard is installed in the Rainbow system. The code to determine this is operating system dependent. The examples in the following sections are written for CP/M, MS-DOS, and Concurrent CP/M.

### Example of Version Test for CP/M System

```assembly
;;; ******************************************************
;;; ; procedure test_board_version
;;; purpose: Test motherboard version
;;; restriction: This routine will work under cp/m only.
;;; entry: none.
;;; exit: flag := 0 = 'A' motherboard
;;;        1 = 'B' motherboard
;;; register usage: ax,bx,cx,dx,di,si,es
;;; ******************************************************
```

5-2
Initialization and Control

Example of Version Test for MS-DOS System

;******************************************************************************
; ; procedure test_board_version
; ; purpose: test motherboard version
; ; restriction: this routine will work under MS-DOS only
; ; entry: none
; ; exit: flag = 0 = 'A' motherboard
; ; 1 = 'B' motherboard
; ; register usage: ax,bx,cx,dx,di,si
;******************************************************************************
Initialization and Control

: cseg    segment byte    public 'codesg'
public test_board_version
assume cs:cseg,ds:dseg,es:dseg,ss:nothing
:
test_board_version    proc    near
push    bp               ;save bp
mov     di,0            ;clear buffer to be sure
mov     cx,14           ;14 bytes to clear
xor     al,al           ;clear clearing byte
:
tb1: mov     byte ptr buffer[di],al ;do the clear
inc     di
loop    tb1             ;loop till done
mov     ax,ds           ;point bp:dx at buffer for
mov     bp,ax           ;int 18h call
mov     dx,offset buffer
mov     di,1ah          ;set opcode for call to get hw #
int     18h             ;int 40 remapped to 18h under MS-DOS
mov     si,0
mov     cx,8            ;set count for possible return ASCII
:
tb2: cmp     byte ptr buffer[si],0
jne     tb3             ;got something back, have rainbow 'B'
inc     si
loop    tb2             ;no ASCII, set rainbow 'A' type
jmp     tb4
:
tb3: mov     flag,0      ;got ASCII, set rainbow B type
jmp     tb4
:
tb4: pop     bp          ;recover bp
ret

endp
test_board_version
:
cseg    ends


dseg    segment byte    public 'datasg'
public flag
flag db 0
buffer db 14 dup (?)
dseg    ends
Example of Version Test for Concurrent CP/M System

;******************************************************************************
; procedure test_board_version
;******************************************************************************
;
purpose: test motherboard version

restriction: this routine for Concurrent CP/M only

entry: none

exit: flag := 0 = 'A' motherboard

1 = 'B' motherboard

register usage: ax,bx,cx,dx,si

mov control_b+2,ds
mov dl,offset biosd
mov bx,3
mov [di+bx],ds
mov dx,offset biosd ;setup for function 50 call
mov cl,32h
int 0e0h ;function 50
mov flag,0 ;set flag for rainbow 'A'
mov bx,6 ;offset to array_14
mov si,offset array_14
mov al,'0'
cmp [si+bx],al ;'O', could be a rainbow 'A'
jne found_b ;no, must be rainbow 'B'
inc bx ;next number...
mov al,'1' ;can be either 1...
cmp [si+bx],al
je test_board_exit
mov al,'2' ;or 2 ...
cmp [si+bx],al
je test_board_exit
mov al,'3' ;or 3 if its a rainbow 'A'
cmp [si+bx],al
je test_board_exit
Initialization and Control

```assembly
found_b:
    mov flag,1         ; it's a rainbow 'B'

test_board_exit:
    ret

dseg
biosd  db  80h
       dw  offset control_b
       dw  0
control_b dw  4
       dw  0
       dw  offset array_14
array_14 rs  14
flag db  0
end
```

**Initialize the Graphics Option**

Initializing the Graphics Option can be separated into the following three major steps:

- Reset the GDC to the desired display environment.
- Initialize the rest of the GDC's operating parameters.
- Initialize the Graphic Option's registers, buffers, and maps.

**Reset the GDC**

To reset the GDC, give the RESET command with the appropriate parameters followed by commands and parameters to set the initial environment. The RESET command is given by writing a zero byte to port 57h. The reset command parameters are written to port 56h.

The GDC Reset Command parameters are the following:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12h</td>
<td>The GDC is in graphics mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Video display is noninterlaced</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No refresh cycles by the GDC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drawing permitted only during retrace</td>
</tr>
<tr>
<td>2</td>
<td>16h</td>
<td>For medium resolution</td>
</tr>
<tr>
<td></td>
<td>30h</td>
<td>For high resolution</td>
</tr>
</tbody>
</table>

The number of active words per line, less two. There are 24 (18h) active words per line in medium resolution mode and 50 (32h) words per line in high resolution mode.
Initialization and Control

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>61h</td>
<td>For medium resolution</td>
</tr>
<tr>
<td></td>
<td>64h</td>
<td>For high resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The low-order five bits are the horizontal sync width in words less one (medium res. HS=2, high res. HS=5). The high-order three bits are the low-order three bits of the vertical sync width in lines (VS=3).</td>
</tr>
<tr>
<td>4</td>
<td>04h</td>
<td>For medium resolution</td>
</tr>
<tr>
<td></td>
<td>08h</td>
<td>For high resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The low-order two bits are the high-order two bits of the vertical sync width in lines. The high-order six bits are the horizontal front porch width in words less one (medium res. HFP=2, high res. HFP=3).</td>
</tr>
<tr>
<td>5</td>
<td>02h</td>
<td>For medium resolution</td>
</tr>
<tr>
<td></td>
<td>03h</td>
<td>For high resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Horizontal back porch width in words less one (medium res. HBP=3, high res. HBP=4).</td>
</tr>
<tr>
<td>6</td>
<td>03h</td>
<td>Vertical front porch width in lines (VFP=3).</td>
</tr>
<tr>
<td>7</td>
<td>F0h</td>
<td>Number of active lines per video field (single field, 240 line display).</td>
</tr>
<tr>
<td>8</td>
<td>40h</td>
<td>The low-order two bits are the high-order two bits of the number of active lines per video field. The high-order six bits are the vertical back porch width in lines (VBP=16).</td>
</tr>
</tbody>
</table>

Initialize the GDC

Now that the GDC has been reset and the video display has been defined, you can issue the rest of the initialization commands and associated parameters by writing to ports 57h and 56h respectively.

Start the GDC by issuing the START command (6Bh).

ZOOM must be defined; however, since there is no hardware support for the Zoom feature, program a zoom magnification factor of one by issuing the ZOOM command (46h) with a parameter byte of 00.

Issue the WDAT command (22h) to define the type of Read/Modify/Write operations as word transfers - low byte, then high byte. No parameters are needed at this time because the GDC is not being asked to do a write operation; it is only being told how to relate to the memory.
Initialization and Control

Issue the PITCH command (47h) with a parameter byte of 20h for medium resolution or 40h for high resolution to tell the GDC that each scan line begins 32 words after the previous one for medium resolution and 64 words after the previous one for high resolution. Note, however, that only 24 or 50 words are displayed on each screen line. The undisplayed words left unscanned are unusable.

The GDC can simultaneously display up to four windows. The PRAM command defines the window display starting address in words and its length in lines. The Graphics Option uses only one display window with a starting address of 0000 and a length of 256 lines. To set this up, issue the PRAM command (70h) with four parameter bytes of 00,00,F0,0F.

Another function of the GDC’s parameter RAM is to hold soft character fonts and line patterns to be drawn into the bitmap. The Graphics Option, rather than using the PRAM for this purpose, uses the external Character RAM and Pattern Generator. For the external hardware to work properly, the PRAM command bytes 9 and 10 must be loaded with all ones. Issue the PRAM command (78h) with two parameter bytes of FF,FF.

Issue the CCHAR command (4Bh) with three parameter bytes of 00,00,00, to define the cursor characteristics as being a non-displayed point, one line high.

Issue the VSYNC command (6Fh) to make the GDC operate in master sync mode.

Issue the SYNC command (0Fh) to start the video refresh action.

The GDC is now initialized.

Initialize the Graphics Option

First you must synchronize the Graphics Option with the GDC’s write cycles. Reset the Mode Register by writing anything to port 50h and then load the Mode Register.

Next, load the Scroll Map. Wait for the start of a vertical retrace, enable Scroll Map addressing, select the Scroll Map, and load it with data.

Initialize the Color Map with default data kept in a shadow area. The Color Map is a write-only area and using a shadow area makes the changing of the color palette more convenient.

Set the Pattern Generator to all ones in the Pattern Register and all ones in the Pattern Multiplier.

Set the Foreground/Background Register to all ones in the foreground and all zeros in the background.

Set the ALU/PS Register to enable all four planes and put the option in REPLACE mode.

Finally, clear the screen by setting the entire bitmap to zeros.
Initialization and Control

Example of Initializing the Graphics Option

The following example is a routine that will initialize the Graphics Option including the GDC. This initialization procedure leaves the bitmap cleared to zeros and enabled for writing but with graphics output turned off. Use the procedure in the next section to turn the graphics output on. Updating of the bitmap is independent of whether the graphics output is on or off.

;********************************************************************
; procedure init_option
; purpose: initialize the graphics option
; entry: dx = 1 medium resolution
; dx = 2 high resolution
; exit: all shadow bytes initialized
; register usage: none, all registers are saved
;********************************************************************
cseg segment byte public 'codesg'
extrn alus:near,pattern_register:near,pattern_mult:near,fgbg:near
public init_option
assume cs:cseg,ds:dseg,es:dseg,ss:nthing
init_option proc near
push ax ;save the registers
push bx
push cx
push dx
push di
push si
clid ;make sure that stos incs.

;First we have to find out what the interrupt vector is for the
;graphics option. If this is a Model 100-A, interrupt vector
;22h is the graphics interrupt. If this is a Model 100-B, the
;interrupt vector is relocated up to A2. If EE00:0F44h and
;04<0, we have the relocated vectors of a Model 100-B and need
;to OR the msb of our vector.

mov ax,ds
mov word ptr cs:segment_save,ax
push es ;save valid es
mov bx,0ee00h ;test if vectors are relocated
mov es,bx
mov ax,08h ;100-A int. vector base addr
test es:byte ptr 0f44h,4 ;relocated vectors?
jz g0 ;jump if yes
mov ax,288h ;100-B int. vector base addr

5-9
Initialization and Control

G0:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>word ptr g_int_vec, ax</td>
</tr>
<tr>
<td>pop</td>
<td>es</td>
</tr>
<tr>
<td>cmp</td>
<td>dx, 1</td>
</tr>
<tr>
<td>jz</td>
<td>mid_res</td>
</tr>
<tr>
<td>jmp</td>
<td>hi_res</td>
</tr>
</tbody>
</table>

; medium resolution?
; jump if yes
; else is high resolution

Mid_res:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>al, 00</td>
</tr>
<tr>
<td>out</td>
<td>57h, al</td>
</tr>
<tr>
<td>mov</td>
<td>gbmod, 030h</td>
</tr>
<tr>
<td>call</td>
<td>mode</td>
</tr>
<tr>
<td>mov</td>
<td>al, 12h</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 16h</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 61h</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 04</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 02,</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 03</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 0f0h</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 40h</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 047h</td>
</tr>
<tr>
<td>out</td>
<td>057h, al</td>
</tr>
<tr>
<td>mov</td>
<td>al, 32</td>
</tr>
<tr>
<td>out</td>
<td>056h, al</td>
</tr>
<tr>
<td>mov</td>
<td>word ptr nmritl, 3fff</td>
</tr>
<tr>
<td>mov</td>
<td>word ptr xmax, 383</td>
</tr>
<tr>
<td>mov</td>
<td>byte ptr num_planes, 4</td>
</tr>
<tr>
<td>mov</td>
<td>byte ptr shifts_per_line, 5</td>
</tr>
<tr>
<td>mov</td>
<td>byte ptr words_per_line, 32</td>
</tr>
<tr>
<td>jmp</td>
<td>common_init</td>
</tr>
</tbody>
</table>
hi_res: mov al,00 ; high resolution reset command
    out 57h,al
    mov gbmod,031h ; mode = high res, text, no readback
    call mode ; disable graphics output
    mov al,12h ; p1. refresh, draw enabled during retrace
    out 056h,al
    mov al,30h ; p2. 50 words/line - 2
    out 056h,al
    mov al,64h ; p3. hsync w-1=4(low 5 bits), vsync
    out 056h,al ; w=3(upper three bits)
    mov al,08 ; p4. hor fp w-1=2(upper 2 bits),
    out 056h,al ; vsync high byte = 0
    mov al,03 ; p5. hbp-1. 3 words hbp
    out 056h,al
    mov al,03 ; p6. vertical front porch, 3 lines
    out 056h,al
    mov al,0f0h ; p7. active lines displayed
    out 056h,al
    mov al,40h ; p8. 6 bits vbp/2 bits lines per field
    out 056h,al ; high byte. vbp=16 lines
    mov al,047h ; p9. pitch command, high res, straight up
    out 057h,al
    mov al,64 ; p10. high res pitch is 64 words/line
    out 056h,al
    mov word ptr nmritl,7fffh
    mov word ptr xmax,799 ; 800 pixels across
    mov byte ptr num_planes,2 ; 2 planes in high res
    mov byte ptr shifts_per_line,6 ; shifts for 64 wds/line
    mov byte ptr words_per_line,64 ; number of words/line

common_init:
    mov al,00 ; setup start window display for memory
    mov startl,al ; location 00
    mov starth,al
    mov al,06bh ; start command
    out 057h,al ; start the video signals going
    mov al,046h ; zoom command
    out 057h,al
    mov al,0 ; magnification assumed to be 0
    out 056h,al
    mov al,22h ; setup R/M/W memory cycles for
    out 57h,al ; figure drawing

Initialization and Control
Initialization and Control

; Initialize PRAM command. Start window at the address in startl, starth. Set the window length for 256 lines. Fill PRAM parameters 8 and 9 with all ones so GDC can do graphics draw commands without altering the data we want drawn.

mov al,070h ; issue the pram command, setup
out 057h,al ; GDC display
mov al,startl ; p1. display window starting address
out 056h,al ; low byte
mov al,starth ; p2. display window starting address
out 056h,al ; high byte
mov al,0ffh ; p3. make window 256 lines
out 056h,al
mov al,0fh ; p4. high nibble display line on
out 056h,al ; right, the rest = 0
mov al,078h ; issue pram command pointing to p8
out 057h,al
mov al,0ffh ; fill pram with ones pattern
out 056h,al
out 056h,al
mov al,04bh ; issue the cchar command
out 057h,al
xor al,al ; initialize cchar parameter bytes
mov cchp1,al ; graphics cursor is one line, not displayed, non-blinking
mov cchp2,al
out 056h,al
mov cchp3,al
out 056h,al
mov al,06fh ; vsync command
out 057h,al
out 050h,al ; reset the graphics board
mov al,0bfh
out 53h,al
mov al,byte ptr gbmod ; enable, then disable interrupts
or al,40h ; to flush the interrupt hardware
out 51h,al ; latches
mov cx,4920 ; wait for a vert sync to happen
Initialization and Control

; disable the interrupts
mov al,0bfh
out 53h,al
mov al,byte ptr gbmod
out 51h,al

; load colormap
call assert_colormap

; initialize scroll map
call inscr1

; set pattern multiplier to 16-b1
mov b1,1
mov b1,0ffh

; set pattern data of all bits set
call pattern_register
mov b1,0f0h

; enable all foreground registers
mov fgbg
mov b1,0

; enable planes 0-3, REPLACE logic
mov alups
mov di,offset p1
mov al,0ffh
mov cx,16
rep stosb
mov al,0
mov gbmskl,al
mov gbmskh,al
mov al,0ffh
mov gdcml,al
mov gdcmh,al
mov word ptr curl0,0
mov ax,word ptr gbmskl
out 54h,al
mov al,ah
out 55h,al

; set cursor to top screen left
call setram

; then set ram to p1 thru p16 data
mov word ptr ymax,239
mov al,0dh
out 57h,al

; enable the display
pop si

; recover the registers
pop di
pop dx
pop cx
pop bx
pop ax
ret

init_option endp
Initialization and Control

;******************************
;*  graphics subroutines      *
;******************************

;xsubs proc near
public setram,assert_colormap,gdc_not_busy,imode,color_int,scrol_int
public cxy2cp,mode

;******************************
; subroutine assert_colormap  
;******************************

colormap is located at clmpda which is defined in 
procedure "change_colormap"

entry: clmpda = colormap to be loaded
exit: none
register usage: ax,bx

assert_colormap:
    cld
call gdc_not_busy       ;make sure nothing's happening

;The graphics interrupt vector "giv" is going to be either 22h or 
A2h depending on whether this is a Model 100-A or a Model 100-B 
with relocated vectors. Read the old vector, save it, then 
overwrite it with the new vector.

push    es
xor     ax,ax
mov     es,ax
mov     bx,word ptr g_int_vec          ;fetch address of "giv"
cli
mov     ax,es:[bx]                     ;temp. disable interrupts
mov     word ptr old_int_off,ax
mov     ax,es:[bx+2]                   ;read the old offset
mov     word ptr old_int_seg,ax
mov     word ptr es:[bx],offset color_int ;load new offset
mov     ax,cs
mov     es:[bx+2],ax                   ;load new int segment
sti    ;re-enable interrupts
pop     es
mov     byte ptr int_done,0           ;clear interrupt flag
or     byte ptr gbmod,40h             ;enable graphics interrupt

5-14
Initialization and Control

ac1:   test byte ptr int_done,0ffh ;has interrupt routine run?
   jz ac1
   push es ;restore interrupt vectors
   xor ax,ax
   mov es,ax
   mov bx,word ptr g_int_vec ;fetch graphics vector offset
   cli
   mov ax,word ptr old_int_off ;restore old interrupt vector
   mov es:[bx],ax
   mov ax,word ptr old_int_seg
   mov es:[bx+2],ax
   sti
   pop es ;make lodsb inc si
   ret

color_int:
   push es
   push ds
   push si
   push cx
   push ax
   mov ax,word ptr cs:segment_save ;can’t depend on es or ds
   mov ds,ax ;reload segment registers
   mov es,ax
   cld
   and byte ptr gbmod,0b0h ;disable graphics interrupts'
   call mode
   mov si,offset clmpda ;fetch color source
   mov al,0d0h ;get the color map’s attention
   out 053h,al
   mov cx,32 ;32 color map entries
   lodsb ;fetch current color map data
   out 051h,al ;load color map
   loop c11 ;loop until all color map data loaded
   mov byte ptr int_done,0ffh ;set "interrupt done" flag
   pop ax
   pop cx
   pop si
   pop ds
   pop es
   iret
Initialization and Control

; ****************************************************
; subroutine cxy2cp
; CXY2CP takes the xinit and yinit numbers, converts them to
; an absolute memory location and puts that location into
; cur10,1,2. yinit is multiplied by the number of words per
; line. The lower 4 bits of xinit are shifted to the left
; four places and put into cur12. xinit is shifted right four
; places to get rid of pixel information and then added to
; yinit times words per line. This result becomes cur10,
; cur11.
;
; entry: xinit = x pixel location
; yinit = y pixel location
; exit: cur10,1,2
; register usage: ax,bx,cx,dx
; *****************************************************

cxy2cp: mov cl,byte ptr shifts_per_line
mov ax,yinit ;compute yinit times words/line
shl ax,cl ;ax has yinit times words/line
mov bx,xinit ;calculate the pixel address
mov dx,bx ;save a copy of xinit
mov cl,4 ;shift xinit 4 places to the left
shl bl,cl ;bl has pixel within word address
mov cur12,bl ;pixel within word address
mov cl,4 ;shift xinit 4 places to right
shr dx,cl ;to get xinit words
add ax,dx
mov word ptr cur10,ax ;word address
ret

; *****************************************************
; subroutine gdc_not_busy
; gdc_not_busy will put a harmless command into the GDC and
; wait for the command to be read out of the command FIFO.
; This means that the GDC is not busy doing a write or read
; operation.
;
; entry: none
; exit: none
; register usage: ax
; *****************************************************
; gdc_not_busy:
push cx ;use cx as a time-out loop counter
in al,056h ;first check if the FIFO is full
test al,2
jz gnb2 ;jump if not
mov cx,8000h ;wait for FIFO not full or reasonable
gnb0: in al,056h ;time, whichever happens first
test al,2 ;has a slot opened up yet?
jz gnb2 ;jump if yes
loop gnb0 ;if loop count exceeded, go on anyway
gnb2: mov al,0dh ;issue a screen-on command to GDC
out 057h,al
in al,056h ;did that last command fill it?
test al,2
jz gnb4 ;jump if not
mov cx,8000h
gnb3: in al,056h ;read status register
test al,2 ;test FIFO full bit
jnz gnb4 ;jump if FIFO not full
loop gnb3 ;loop until FIFO not full or give up
gnb4: mov ax,40dh ;issue another screen-on,
out 057h,al ;wait for FIFO empty
mov cx,8000h
gnb5: in al,056h ;read the GDC status
test ah,al ;FIFO empty bit set?
jnz gnb6 ;jump if not.
loop gnb5
gnb6: pop cx
ret
****************************
; subroutine imode
;
; issue Mode command with the parameters from register gbmod
;
; entry: gbmod
; exit: none
; register usage: ax

imode: call gdc_not_busy
mov al,0bfh ;address the mode register through
out 53h,al ;the indirect register
mov al,gbmod
out 51h,al ;load the mode register
ret
Initialization and Control

mode: mov al,0bfh ;address the mode register through
      out 53h,al ;the indirect register
mov al,gbmod
out 51h,al ;load the mode register
ret

******************************************************************************

; subroutine inscr1
;
; initialize the scroll map
;
; entry: none
; exit: none
; register usage: ax,bx,cx,dx,di,si
******************************************************************************

inse1: cld
mov cx,256 ;initialize all 256 locations of the
xor al,al ;shadow area to desired values
mov di,offset scrltb
inse0: stosb
inc al
loop inse0

;The graphics interrupt vector is going to be either 22h or A2h
;depending on whether this is a Model 100-A or a Model 100-B with
;relocated vectors. Read the old vector, save it, and overwrite it
;with the new vector. Before we call the interrupt, we need to
;make sure that the GDC is not writing something out to the bitmap.

ascrol: call gdc_not_busy ;check if GDC is busy
       push es
       xor ax,ax
       mov es,ax
       mov bx,word ptr g_int_vec
       cli ;temporarily disable interrupts
       mov ax,es:[bx] ;read the old offset
       mov word ptr old_int_off,ax
       mov ax,es:[bx+2] ;read the old segment
       mov word ptr old_int_seg,ax
       mov word ptr es:[bx],offset scrol_int ;load new offset
       mov ax,cs
       mov es:[bx+2],ax ;load new interrupt segment
       sti ;re-enable interrupts
       pop es
       mov byte ptr int_done,0 ;clear interrupt flag
       or byte ptr gbmod,40h ;enable graphics interrupt
       call mode
Initialization and Control

```assembly
asm:  test  byte ptr int_done,0ffh ;has interrupt routine run?
      jz  asm
      push es                      ;restore the interrupt vectors
      xor  ax,ax
      mov  es,ax
      mov  bx,word ptr g_int_vec ;fetch graphics vector offset
      cli
      mov  ax,word ptr old_int_off ;restore old interrupt vector
      mov  es:[bx],ax
      mov  ax,word ptr old_int_seg
      mov  es:[bx+2],ax
      sti
      pop  es
      ret

;Scrollmap loading during interrupt routine.
;Fetch the current mode byte and enable scroll map addressing.

scroll_int:
      push  es
      push  ds
      push  si
      push  dx
      push  cx
      push  ax
      mov  ax,word ptr cs:segment_save ;can't depend on ds
      mov  ds,ax                       ;reload it
      mov  es,ax
      and  byte ptr gbmod,0bfh         ;disable graphics interrupts
      mov  al,gbmod                    ;prepare to access scroll map
      mov  gtemp1,al                   ;first save current gbmode
      and  gbmod,0dfh                   ;enable writing to scroll map
      call  mode                      ;do it
      mov  al,07fh                     ;select scroll map and reset scroll
      out  53h,al                      ;map address counter
      mov  dl,51h                      ;output port destination
      xor  dh,dh
      mov  si,offset scrollb          ;first line's high byte address=0
      mov  cx,16                      ;256 lines to write to
      test  byte ptr gbmod,1          ;high resolution?
      jnz  ins1
      jmp  if yes
      shr  cx,1                       ;only 128 lines if medium resolution
```

5-19
Initialization and Control

```assembly
ins1: lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
lodsw ; fetch two scrollmap locations
out dx,al ; assert the even byte
mov al,ah
out dx,al ; assert the odd byte
loop inst
mov al,gtemp1 ; restore previous mode register
mov gbmod,al
call mode
mov byte ptr int_done,0ffh ; set interrupt-done flag
pop ax
pop cx
pop dx
pop si
pop ds
pop es
iret ; return from interrupt
```
Initialization and Control

;********************************************************************
; subroutine setram
; set video ram to a value stored in the p table
; entry: 16 byte p1 table
; exit: none
; register usage: ax,bx,cx,dx,di,si
;********************************************************************

setram: mov byte ptr twdir,2 ;set write direction to the right
call gdc_not_busy ;make sure that the GDC isn't busy
mov al,0feh ;select the write buffer
out 053h,al
out 051h,al ;reset the write buffer counter
mov si,offset p1 ;initialize si to start of data
mov cx,10h ;load 16 chars into write buffer
setr1: lodsb ;fetch byte to go to write buffer
out 52h,al
loop setr1
mov al,0feh ;select the write buffer
out 053h,al
out 051h,al ;reset the write buffer counter
mov al,049h ;issue GDC cursor location command
out 57h,al
mov al,byte ptr cur10 ;fetch word location low byte
out 56h,al ;load parameter
mov al,byte ptr cur11 ;fetch word location high byte
out 56h,al ;load parameter
mov al,4ah ;set the GDC mask to all F's
out 57h,al
mov al,0ffh
out 56h,al
out 56h,al
mov al,04ch ;issue figs command
out 57h,al
mov al,byte ptr twdir ;direction to write.
out 56h,al
mov al,nmritl ;number of GDC writes, low byte
out 56h,al
mov al,nmrith ;number of GDC writes, high byte
out 56h,al
mov al,22h ;wdat command
out 57h,al
mov al,0ffh ;p1 and p2 are dummy parameters
out 56h,al ;the GDC requires them for internal
out 56h,al ;purposes - no effect on the outside
ret
segment_save dw 0 ;ds save area for interrupts
gsubs endp
cseg ends
dseg segment byte public 'datasg'
extrn cimpd:byte
public xmax,ymax,alu,d,d1,d2,d3
dpublic cur10,cur11,cur12,dir,fg,gbmskl,gbmskh,gbmod,gdcml,gdcmh
dpublic nmredi,nmredh,nmril,nmrith,p1,prdata,prmult,scrltb,startl
extrn gtemp3,gtemp4,starth,gtemp,gtemp1,gtemp2,twdir,xinit,xfinal
extrn yinit,yfinal,ascrol,num_planes,shifts_per_line
extrn words_per_line,g_int_vec

;variables to be remembered about the graphics board states

alu db 0 ;current ALU state
cchp1 db 0 ;cursor/character
cchp2 db 0 ; size definition
cchp3 db 0 ; parameter bytes
cur10 db 0 ;cursor - low byte
cur11 db 0 ; location - middle byte
cur12 db 0 ; storage - high bits & dot address
dc dw 0 ;figs command dc parameter
d dw 0 ;figs command d parameter
d2 dw 0 ;figs command d2 parameter
d1 dw 0 ;figs command d1 parameter
dir db 0 ;figs direction.
fg db 0 ;current foreground register
gbmskl db 0 ;graphics board mask register - low byte
gbmskh db 0 ; - high byte
gbmod db 0 ;graphics board mode register
gdcml db 0 ;GDC mask register bits - low byte
gdcmh db 0 ; - high byte
Initialization and Control

```
g_int_vec    dw  0  ; graphics option's interrupt vector
gtemp     dw  0  ; temporary storage
gtemp1    db  0  ; temporary storage
gtemp2    db  0  ; temporary storage
gtemp3    db  0  ; temporary storage
gtemp4    db  0  ; temporary storage
int_done  db  0  ; interrupt-done state
nmredl    db  0  ; number of read operations - low byte
nmredh    db  0  ; number of read operations - high byte
nmritl    db  0  ; number of GDC writes - low byte
nmrith    db  0  ; number of GDC writes - high byte
num_planes db  0  ; number of planes in current resolution
old_int_seg dw  0  ; old interrupt segment
old_int_off dw  0  ; old interrupt offset
p1        db  16 dup (?) ; shadow write buffer & GDC parameters
prdata    db  0  ; pattern register data
prmult    db  0  ; pattern register multiplier factor
scrltb    db  100h dup (?) ; scroll map shadow area
si_temp   dw  0  ; register for start address of display
startl    db  0  ; direction for text mode write operation
starth    db  0  ; shift factor for one line of words
words_per_line db  0  ; words/scan line for current resolution
xinit     dw  0  ; x initial position
yinit     dw  0  ; y initial position
xfinal    dw  0  ; x final position
yfinal    dw  0  ; y final position
xmax      dw  0
ymax      dw  0
dseg      ends
```

end
Controlling Graphics Output

There will be occasions when you will want to control the graphics output to the monitors. The procedure varies according to the monitor configuration. The following two examples illustrate how graphics output can be turned on and off in a single monitor system. The same procedures can be used to turn graphics output on and off in a dual monitor system. However, in a dual monitor configuration, you may want to display graphics output only on the color monitor and continue to display VT102 VSS text output on the monochrome monitor. This can be accomplished by loading an 83h into 0Ah instead of an 87h.

Example of Enabling a Single Monitor

```assembly
; procedure graphics_on
; purpose: enable graphics output on single color monitor
; entry: gbmod contains mode register shadow byte
; exit: none
; register usage: ax

; dseg segment byte public 'datasg'
extrn gbmod:byte ;defined in procedure 'init_option'
dseg ends
cseg segment byte public 'codesg'
extrn imode:near ;defined in procedure 'init_option'
public graphics_on
assume cs:cseg,ds:dseg,es:dseg,ss:nothing

graphics_on proc near
  mov al,87h
  out 0ah,al ;enable graphics on monochrome line
  or byte ptr gbmod,080h ;enable graphics output in gbmod
  call imode ;assert new mode register
  ret
endp
```

5-24
Example of Disabling a Single Monitor

```assembly
;********************************************************************
; procedure graphics_off
; purpose: disable graphics output to single (color) monitor
; entry: gbmod contains mode register shadow byte
; exit: none
; register usage: ax
;********************************************************************

dseg    segment byte public 'datasg'
extrn   gbmod:byte ; defined in procedure 'init_option'
dseg    ends
cseg    segment byte public 'codesg'
extrn   imode:near ; defined in procedure 'init_option'
           public graphics_off
           assume cs:cseg,ds:dseg,es:dseg,ss:nothing
;
graphics_off proc near
    byte ptr gbmod,07fh ; disable graphics output in gbmod
    call     imode ; assert new mode register
    mov      al,83h
    out      Oah,al ; turn off graphics on monochrome line
ret
graphics_off endp
```

Modifying and Loading the Color Map

For an application to modify the Color Map, it must first select the Color Map by way of the Indirect Register (write DFh to port 53h). This will also clear the Color Map Index Counter to zero so loading always starts at the beginning of the map.

Loading the Color Map is done during vertical retrace so there will be no interference with the normal refreshing of the bitmap. To ensure that there is sufficient time to load the Color Map, you must catch the beginning of a vertical retrace. First, check for vertical retrace going inactive (bit 5 of the GDC Status Register = 0). Then, look for the vertical retrace to start again (bit 5 of the GDC Status Register = 1).
To modify only an entry or two, the use of a shadow color map is suggested. Changes can first be made anywhere in the shadow map and then the entire shadow map can be loaded into the Color Map. The next section is an example of modifying a shadow color map and then loading the data from the shadow map into the Color Map.

**Example of Modifying and Loading Color Data in a Shadow Map**

```assembly
;******************************************************************************
; procedure change_color_map
; purpose: change a color in the colormap
; entry:   ax = new color (0 = highest intensity)
;          (F = lowest intensity)
;          al = high nibble = red data
;          low nibble = green data
;          ah = high nibble = gray data
;          low nibble = blue data
;          bx = palette entry number
; exit:
; register usage:   ax,bx,si
;******************************************************************************

cseg segment byte public 'codesg'
extrn assert_colormap:near ;defined in 'init_option'
public change_colormap
assume cs:cseg,ds:dseg,es:dseg,ss:nothing

change_color_map proc near
  mov   si,offset clmpda ;colormap shadow
  mov   [si+bx],al ;store the red and green data
  add   bx,16 ;increment to gray and blue data
  mov   [si+bx],ah ;store the gray and blue data
  call  assert_colormap ;defined in 'init_option'

change_color_map endp
cseg ends

dseg segment byte public 'datasg'
public clmpda
```

5-26
Information in the Color Map is stored as 16 bytes of red and green data followed by 16 bytes of monochrome and blue data. For each color entry, a 0 specifies full intensity and 0fh specifies zero intensity.

A sample set of color map entries for a Model 100-B system with a monochrome monitor in medium resolution (16 shades) would look as follows in the shadow area labelled CLMPDA:

```plaintext
;colormap:
;---------
;no red or green data

;clmpda
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
  db 0fff
```

monochrome data, no blue data

```
  db 0fff ;black
  db 00fh ;white
  db 01fh .
  db 02fh .
  db 03fh ;light monochrome
  db 04fh .
  db 05fh .
  db 06fh .
  db 07fh ;medium monochrome
  db 08fh .
  db 09fh .
  db 0afh .
  db 0bfh ;dark monochrome
  db 0c0h .
  db 0d0h .
  db 0efh .
```
On a Model 100-A system, only the lower two bits of the monochrome nibble are significant. This allows only four monochrome shades as opposed to 16 shades on the Model 100-B system in medium resolution mode. The following sample set of data applies to both the Model 100-A monochrome-only system in either medium or high resolution mode, as well as the Model 100-B monochrome-only system in high resolution mode.

: no red or green data

clmpda

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

db 0ffh

: monochrome data, no blue data


db 0ffh ; black

db 0ffh ; white

db 05fh ; light monochrome

db 0afh ; dark monochrome

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

db 0ffh ; black

5-28
In a dual monitor configuration, with a Model 100-B system in medium resolution mode, all four components of each color entry are present: red, green, blue and monochrome. A sample set of color data would be as follows:

;red and green data

clmpda       db 0ffh ;black
             db 000h ;white
             db 0f0h ;cyan
             db 00fh ;magenta
             db 000h ;yellow
             db 00fh ;red
             db 0ffh ;blue
             db 0f0h ;green
             db 0aah ;dk gray
             db 0f8h ;dk cyan
             db 08fh ;dk magenta
             db 088h ;dk yellow
             db 08fh ;dk red
             db 0ffh ;dk blue
             db 0f8h ;dk green
             db 077h ;gray

;monochrome and blue data

             db 0ffh ;black  black
             db 000h ;white  white
             db 010h ;.      cyan
             db 020h ;.      magenta
             db 03fh ;light mono. yellow
             db 04fh ;.      red
             db 050h ;.      blue
             db 06fh ;.      green
             db 07ah ;med. mono. dk gray
             db 0f8h ;.      dk cyan
             db 098h ;.      dk magenta
             db 0afh ;.      dk yellow
             db 0bfh ;dark mono. dk red
             db 0c8h ;.      dk blue
             db 0dfh ;.      dk green
             db 0e7h ;.      gray
On a Model 100-A dual monitor configuration, in medium resolution mode, all 16 color entries are displayable. However, only two bits of monochrome data are available allowing for only 4 monochrome shades.

On a Model 100-A dual monitor configuration, in high resolution mode, there are four displayable colors and again, four monochrome shades.

On a Model 100-B dual monitor configuration, in high resolution mode, there also are four displayable colors and four monochrome shades.

In a color monitor only system, the green data must be mapped to the monochrome output. For a Model 100-B single color monitor system, in medium resolution mode, a sample color map would be as shown below:

```
NOTE

The following sample color map will be assembled with this example. If this is not appropriate, substitute one of the other samples or generate one that is custom tailored to the application.

;red data, green data mapped to mono.

clmpda
```

```asm
    db 0ffh ;black
    db 00fh ;white
    db 0ffh ;cyan
    db 00fh ;magenta
    db 00fh ;yellow
    db 00fh ;red
    db 0ffh ;blue
    db 0ffh ;green
    db 0fah ;dk gray
    db 0ffh ;dk cyan
    db 08fh ;dk magenta
    db 08fh ;dk yellow
    db 08fh ;dk red
    db 0ffh ;dk blue
    db 0ffh ;dk green
    db 07fh ;gray
```
Initialization and Control

; green data, blue data
;
  db 0ffh ; black
  db 000h ; white
  db 000h ; cyan
  db 0f0h ; magenta
  db 00fh ; yellow
  db 0ffh ; red
  db 0f0h ; blue
  db 00fh ; green
  db 0aah ; dk gray
  db 088h ; dk cyan
  db 0f8h ; dk magenta
  db 08fh ; dk yellow
  db 0ffh ; dk red
  db 0f8h ; dk blue
  db 08fh ; dk green
  db 077h ; gray

; For a Model 100-A single color monitor system, in either high or
; medium resolution mode, only the lower two bits of the monochrome
; output are significant. Therefore, you can only display four
; intensities of green since the green data must be output through
; the monochrome line. The same applies to a Model 100-B single
; color monitor system in high resolution mode.
;
; dseg    ends
  end

5-31
Loading the ALU/PS Register

The ALU/PS Register data determines which bitmap planes will be written to during a Read/Modify/Write (RMW) cycle and also sets the operation of the logic unit to one of three write modes.

Bits 0 through 3 enable or disable the appropriate planes and bits 4 and 5 set the writing mode to REPLACE, COMPLEMENT, or OVERLAY. Bits 6 and 7 are not used. Bit definitions for the ALU/PS Register are in Part III of this manual.

Write an EFh to port 53h to select the ALU/PS Register and write the data to port 51h.

Example of Loading the ALU/PS Register

```
;***************************************************************
; ; procedure alups
; ; purpose: set the ALU/PS register
; ; entry: bl = value to set ALU/PS register to
; ; exit: update ALU/PS shadow byte
; ; register usage: ax,
;***************************************************************
```
Loading the Foreground/Background Register

The data byte in the Foreground/Background Register determines whether bits are set or cleared in each of the bitmap planes during a bitmap write (RMW) operation. Bit definitions for the Foreground/Background Register are in Part III of this manual.

Write an F7h to port 53h to select the Foreground/Background Register and write the data byte to port 51h.

Example of Loading the Foreground/Background Register

```
//********************************************************************
*                                                                 *
* procedure    fgbg                                             *
*                                                                 *
* purpose:      set the foreground / background register          *
*                                                                 *
* entry:        bl = value to set fgbg register to                *
* exit:         update fgbg shadow byte                           *
* register usage: ax                                            *
********************************************************************
```
; dseg   segment byte  public 'datasg'
extrn  fg:byte
dseg  ends

cseg  segment byte  public 'codesg'
extrn  gdc_not_busy:near
public  fgbg
assume  cs:cseg,ds:dseg,es:dseg,ss:nothing

; fgbg  proc  near
call  gdc_not_busy ;defined in 'init_option'
mov  al,0f7h ;select the foreground/background
out  53h,al ; register
mov  byte ptr fg,bl ;update shadow byte with new value
mov  al,bl
out  51h,al ;load new value into fgbg register
ret

fgbg  endp

cseg  ends
end
This chapter contains examples that illustrate displaying 64K bytes of memory, and clearing a rectangular area of the screen to a given color.

Display Data from Memory

In the following example, video data in a 64K byte area of memory is loaded into the bitmap in order to display it on the monitor. The last byte of the memory area specifies the resolution to be used. A value of zero means use medium resolution mode. A value other than zero means use high resolution mode. In medium resolution mode, the 64K bytes are written to four planes in the bitmap; in high resolution mode, the 64K bytes are written to two planes.

Example of Displaying Data from Memory

```
;**************************************************************************;
; procedure ritvid
; purpose: restore a graphics screen save in a 64k segment of main memory by the procedure ritvid.
;**************************************************************************
```
Area Write Operations

; dseg segment byte public 'datasg'
extrn gbmod:byte,gtemp:word,num_planes:byte,cur1:byte,gtemp1:byte
dseg ends

vidseg segment byte public 'vseg'
extrn viddata:byte
vidseg ends

cseg segment byte public 'codesg'
extrn init_option:near,fgbg:near,gdc_not_busy:near,alups:near
extrn imode:near
public ritvid
assume cs:cseg,ds:dseg,es:dseg,ss:nothing

; ritvid proc near
;
; The video data is in vidseg. The last byte in vidseg is the
; resolution flag. If flag is 0 the option is in medium resolution
; mode; otherwise it is in high resolution mode. Initialize the
; option to that resolution.
;
    mov    ax,es
    mov    word ptr cs:segment_save,ax ;save es
    call   gdc_not_busy ;wait till GDC is free
    mov    ax,vidseg ;set es to point to video buffer
    mov    es,ax
    mov    si,0ffffh ;fetch the resolution flag from
    mov    al,es:[si] ; the last byte of vidbuf
    test   al,0ffh ;is it high resolution?
    jnz    rt1 ;jump if yes.
    mov    dx,1
    jmp    rt2

rt1:   mov    dx,2
rt2:    mov    ax,word ptr cs:segment_save
    mov    es,ax ;restore old es
    call   init_option ;assert the new resolution.

; init-option leaves us in text mode with fg=f0 and alups=0.
;
    and    byte ptr gbmod,0fdh
    or     byte ptr gbmod,010h
    call   imode ;make sure we're in text mode
    mov    bl,0fh ;put 1's into bg and 0's into fg
    call   fgbg ;because write buffer inverts data
    test   byte ptr gbmod,1 ;high resolution?
    jnz    rt3 ;jump if yes.
    mov    word ptr gtemp,1024 ;8 wrd-writes/plane (med res)
    jmp    rt4

rt3:   mov    word ptr gtemp,2048 ;8 wrd-writes/plane (high res)
Area Write Operations

rt4:  mov  d1,0 ;start at beginning of vidbuf.
mov  ax,vidseg ;set es to point to video buffer
mov  es,ax
mov  cl,byte ptr num_planes ;fetch number of planes
xor  ch,ch ; to be written

;Enable a plane to be written.

rt5:  mov  word ptr gtemp1,cx ;save plane writing counter
mov  bl,byte ptr num_planes ;select plane to write enable
sub  bl,cl ;this is plane to write enable
mov  cl,bl
mov  bl,0feh ;put a 0 in that plane’s select position
rol  bl,cl
and  bl,0fh ;keep in REPLACE mode
call  alups ;assert the new ALU/PS

;Fill that plane with data, 8 words at a time, from vidseg.

mov  word ptr curl0,0 ;start write at top left
mov  cx,word ptr gtemp ;number of 8 word writes

rt6:  push  cx ; to fill plane
call  gdc_not_busy ;wait until GDC has finished
mov  al,0feh ; previous write
out  53h,al
out  51h,al
mov  cx,16 ;fetch 16 bytes

rt7:  mov  al,es:[d1] ;fill ptable with data
inc  d1 ; to be written
out  52h,al
loop  rt7
mov  al,49h ;assert the position to
out  57h,al ; start the write
mov  ax,word ptr curl0
out  56h,al
mov  al,ah
out  56h,al
mov  al,04ah ;init the mask to 0ffffff
out  57h,al
mov  al,0ffh
out  56h,al
out  56h,al
xor  al,al
out  54h,al
out  55h,al
mov  al,4ch
out  57h,al ;now start the write
mov  al,2 ;direction is down
Set a Rectangular Area to a Color

The example that follows illustrates how to set a rectangular area of the screen to some specified color. Input data consists of the coordinates of the upper left and lower right corners of the area (in pixels) plus the color specification (a 4-bit index value). The special case of setting the entire screen to a specified color is included in the example as a subroutine that calls the general routine.

Example of Setting a Rectangular Area to a Color

```asm
out 56h,al
mov al,7
:do 8 writes
out 56h,al
xor al,al
out 56h,al
mov al,22h
: start the write
out 57h,al
mov al,0ffh
out 56h,al
out 56h,al
add word ptr curl0,08 ; next location to be written
pop cx
loop rt6 ; loop to complete this plane
mov cx,word ptr gtemp1 ; keep looping until all
loop rt5 ; planes are written
mov ax,word ptr cs:segment_save
mov es,ax
ret
ritvid endp
segment_save dw 0
cseg ends
end
```

```asm
; **************************************************************
; procedure set_all_screen
; purpose: set entire screen to a user defined color
; entry: di is the color to clear the screen to
; exit: fgbg and alups shadow bytes updated
; register usage: ax, bx, cx, dx, si, di
; **************************************************************

cseg segment byte public 'codesg'
extrn fgbg:near, gdc_not_busy:near, imode:near, alups:near
public set_all_screen, set_rectangle
assume cs:cseg, ds:dseg, es:nothing, ss:nothing
```

7-4
Area Write Operations

\begin{verbatim}
;
set_all_screen proc near
    mov word ptr xstart,0 ;start at the top left corner
    mov word ptr ystart,0
    mov ax,word ptr xmax
    mov word ptr xstop,ax ;fetch the bottom right corner
    mov ax,word ptr ymax
    mov word ptr ystop,ax ;coordinates.
    jmp set_rectangle
set_all_screen endp
;
;********************************************************************
; procedure set_rectangle
;
; purpose:
; set a user defined screen rectangle to a user defined color
;
; entry:
; xstart has the start x in pixels
; ystart has the start y in scan lines
; xstop has the stop x in pixels
; ystop has the stop y in scan lines
; di is the color to clear the screen to
;
; exit:
; register usage: ax,bx,cx,dx,di,si,xstart is altered
;********************************************************************
;
set_rectangle proc near
;
;No validity checks are being made on start and stop coordinates.
;
; xstart must be <= xstop
; ystart must be <= ystop
;
;Assert the new screen color to both nibbles of the the foreground/
;background register. Put the option into REPLACE mode with all
;planes enabled and in write-enabled word mode.
;
    mov bx,di ;di has the color; only low nibble valid
    mov bh,bl ;combine color number into both fg and bg
    mov cl,4 ;shift the color up to the high nibble
    shl bh,cl
    or bl,bh ;combine high nibble with old low nibble
    call fgbg ;assert new value to fgbg register
    xor bl,bl ;set up REPLACE mode, all planes
    call alups ;assert new value to ALU/PS register
    and byte ptr gbm0,0fdh ;set up text mode
    or byte ptr gbm0,10h ;set up write enable mode
    call imode ;assert new value to mode register

\end{verbatim}
Area Write Operations

;Do the rectangle write.

;Do the write one column at a time. Since the GDC is a word device,
;we have to take into account that we might have our write window
;start on a pixel that isn’t on a word boundary. The graphics
;options write mask must be set accordingly. Do a write buffer
;write to all of the rectangle as defined by start,stop. Calculate
;the first curr0. Calculate the number of scans per column to be
;written.

mov ax,word ptr xstart ;turn pixel address into
mov cl,4 ; word address
shr ax,cl
mov dx,word ptr ystart ;turn scan start to words/line*y
mov cl,byte ptr shifts_per_line ;number of shifts
shl dx,cl
add dx,ax ;combine x and y word addresses
mov word ptr curr0,dx ;first curr0.
mov ax,word ptr ystop ;subtract start from stop.
sub ax,word ptr ystart
mov word ptr mnr01l,ax

;Program the text mask.

;There are four possible write conditions-

a - partially write disabled to left
b - completely write enabled
c - partially disabled to the right
d - partially disabled to both left and right

;The portion to be write disabled to the left will be the current
;xstart pixel information. As we write a column, we update the
;current xstart location. Only the first xstart will have a left
;hand portion write disabled. Only the last will have a right
;hand portion disabled. If the first is also the last, a portion
;of both sides will be disabled.

cls1: mov bx,0ffffh ;calculate the current write mask
mov cx,word ptr xstart
and cx,0fh ;eliminate all but pixel information
shr bx,cl ;shift in a 0 for each left pixel
; to be disabled
Write buffer write is done by columns. Take the current xstart and use it as the column to be written to. When the word address of xstart is greater than the word address of xstop, we are finished. There is a case where the current word address of xstop is equal to the current word address xstart. In that case we have to be concerned about write disabling the bits to the right. When xstop becomes less than xstart then we are done.

```asm
mov ax,word ptr xstart ;test if word xstop is equal and ax,0ff00h ; to word xstart mov cx,word ptr xstop and cx,0ff00h cmp ax,cx ;below? jb cls3 ;jump if yes je cls2 ;jump if equal - do last write jmp exit ;all done - exit
```

We need to set up the right hand write disable. This is also the last write. bx has the left hand write enable mask in it. Preserve and combine with the right hand mask which will be (f-stop pixel address) bits on the right.

```asm
cls2: mov cx,word ptr xstop ;strip pixel info out of xstop and cx,0fh inc cx ;make endpoint inclusive of write mov ax,0ffffh ;shift the disable mask shr ax,cl ;wherever there is a one, we xor ax,0ffffh ;want to enable writes and bx,ax ;combine right and left masks
```

bx currently has the mask bytes in it. Where we have a one, we want to make a zero so that particular bit will be write enabled.

```asm
cls3: xor bx,0ffffh ;invert to get zeros for ones
```

Assert the new write mask. Make sure that the GDC is not busy before we change the mask.

```asm
cls4: call gdc_not_busy ;check that the GDC isn’t busy mov al,bh ;assert the upper write mask out 55h,al mov al,bl ;assert the lower write mask out 54h,al
```

Position the GDC at the top of the column to be written. This address was calculated earlier and the word need only be fetched and applied. The number of scans to be written has already been calculated.
Area Write Operations

mov al,49h ;assert the GDC cursor address
out 57h,al
mov ax,word ptr cur10 ;assert word address low byte
out 56h,al
mov al,dh ;assert word address high byte
out 56h,al

;Start the write operation. Textmask, alups, gbmod and fgbg are
;already set up. GDC is positioned.

mov al,4ch ;assert figs to GDC
out 57h,al
xord al,al ;direction is down
out 56h,al
mov ax,word ptr nmritl
out 56h,al ;assert number of write
mov al,ah ; operations to perform
out 56h,al
mov al,22h ;assert wdat
out 57h,al
mov al,0ffh
out 56h,al
out 56h,al

;Update the starting x coordinate for the start of the next
;column write. Strip off the pixel information and then add 16
;pixels to it to get the next word address.

and word ptr xstart,0ff00h ;strip off pixel info
add word ptr xstart,16 ;address the next word
inc word ptr cur10
jmp cls1 ;check for another column to clear

exit: ret

set_rectangle endp
cseg ends
dseg segment byte public 'datasg'
eextr cur10:word,gbmod:byte,xmax:word,ymax:word
eextr shifts_per_line:byte
public xstart,xstop,ystart,ystop
xstart dw 0
xstop dw 0
ystart dw 0
ystop dw 0
nmritl dw 0
dseg ends end

7-8
The examples in this chapter illustrate some basic vector write operations. They cover setting up the Pattern Generator and drawing a single pixel, a line, and a circle.

**Setting Up the Pattern Generator**

When operating in Vector Mode, all incoming data originates from the Pattern Generator. The Pattern Generator is composed of a Pattern Register and a Pattern Multiplier. The Pattern Register supplies the bit pattern to be written. The Pattern Multiplier determines how many times each bit is sent to the bitmap write circuitry before being recirculated.

**NOTE**
The Pattern Multiplier must be loaded before loading the Pattern Register.

**Example of Loading the Pattern Register**

The Pattern Register is an 8-bit register that is loaded with a bit pattern. This bit pattern, modified by a repeat factor stored in the Pattern Multiplier, is the data sent to the bitmap write circuitry when the option is in Vector Mode.
Vector Write Operations

procedure pattern_register

purpose: set the pattern register

entry: bl = pattern data

exit: update pattern register shadow byte

register usage: ax

cautions: you must set the pattern multiplier before setting the pattern register

The pattern register contains a 16-bit pixel pattern that is written to the bitmap when the Graphics Option is in Vector Mode.

Sample register values and corresponding patterns are:

<table>
<thead>
<tr>
<th>register value</th>
<th>pattern output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0ffh</td>
<td>11111111</td>
</tr>
<tr>
<td>0aah</td>
<td>10101010</td>
</tr>
<tr>
<td>0f0h</td>
<td>11110000</td>
</tr>
<tr>
<td>0cdh</td>
<td>11001101</td>
</tr>
</tbody>
</table>

The above assumes that the Pattern Multiplier has been set to multiply the pattern by 1. If the Pattern Multiplier had been set to multiply the pattern by 3, the above examples, when output to the bitmap would look as follows:

<table>
<thead>
<tr>
<th>register value</th>
<th>pattern output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0ffh</td>
<td>111111111111111111111111</td>
</tr>
<tr>
<td>0aah</td>
<td>110000110001100011100011100</td>
</tr>
<tr>
<td>0f0h</td>
<td>111111111100000000000000</td>
</tr>
<tr>
<td>0cdh</td>
<td>111111000000111111000111</td>
</tr>
</tbody>
</table>

dseg segment byte public 'dataseg'
eextrn prdata:byte
dseg ends
cseg segment byte public 'codeseg'
eextrn gdc_not_busy:near
public pattern_register
assume cs:cseg,ds:dseg,es:dseg,ss:nothing

8-2
pattern_register proc near
    call gdc_not_busy ; defined in 'init_option'
    mov al,0fbh ; select the pattern register
    out 53h,al
    mov byte ptr prdata,bl ; update shadow byte
    mov al,bl
    out 51h,al ; load the pattern register
    ret
pattern_register endp

Example of Loading the Pattern Multiplier

The Graphics Option expects to find a value in the Pattern Multiplier such that sixteen minus that value is the number of times each bit in the Pattern Register is repeated. In the following example, you supply the actual repeat factor and the coding converts it to the correct value for the Graphics Option.
Display a Pixel

The following example displays a single pixel at a location specified by a given set of x and y coordinates. Coordinate position 0,0 is in the upper left corner of the screen. The x and y values are in pixels and are positive and zero-based. Valid values are:

\[ x = 0 - 799 \text{ for high resolution} \]
\[ 0 - 383 \text{ for medium resolution} \]
\[ y = 0 - 239 \text{ for high or medium resolution} \]

Also, in the following example, it is assumed that the Mode, ALU/PS, and Foreground/Background registers have already been set up for a vector write operation.

Example of Displaying a Single Pixel

```assembly

;********************************************************************
;                      procedure pixel
;                        purpose:  draw a pixel
;                        entry:  xinit = x location
;                                yinit = y location
;                                valid x values = 0-799 high resolution
;                                = 0-383 medium resolution
;                                valid y values = 0-239 med. or high res.
;********************************************************************
```
; Do a vector draw of one pixel at coordinates in xinit,yinit. Assume
; that the Graphics Option is already set up in terms of Mode Register,
; Foreground/Background Register, and ALU/PS Register.

; dseg   segment byte   public 'datasg'
extrn gbmod:byte,curl0:byte,curl1:byte,curl2:byte,xinit:word
extrn yinit:word
dseg   ends

cseg   segment byte   public 'codesg'
extrn cxy2cp:near,gdc_not_busy:near
public pixel
assume cs:cseg,ds:dseg,es:dseg,ss:nothing

pixel   proc   near
    call gdc_not_busy
    call cxy2cp      ;convert x,y to a cursor position
    mov al,49h       ;send out the cursor command byte
    out 57h,al
    mov ax,word ptr curl0 ;assert cursor location low byte
    out 56h,al
    mov al,ah         ;assert cursor location high byte
    out 56h,al
    mov al,byte ptr curl2 ;assert cursor pixel location
    out 56h,al
    mov al,4ch         ;assert the figs command
    out 57h,al
    mov al,02h         ;line direction - to the right
    out 56h,al
    mov al,6ch         ;tell GDC to draw pixel when ready
    out 57h,al
    ret
pixel   endp
cseg   ends
end

Display a Vector

The example in this section will draw a line between two points specified by x and y coordinates
given in pixels. The valid ranges for these coordinates are the same as specified for the previous
example. Again it is assumed that the Mode, ALU/PS, and Foreground/Background registers have
already been set up for a vector write operation. In addition, the Pattern Generator has been set up
for the type of line to be drawn between the two points.
Vector Write Operations

Example of Displaying a Vector

;****************************************************************************
; procedure vector
; purpose: draw a vector
; entry: xinit = starting x location
; yinit = starting y location
; xfinal= ending x location
; yfinal= ending y location
; valid x values = 0 - 799 high resolution
; 0 - 383 medium resolution
; valid y values = 0 - 239 high or med. res.
; exit:
; register usage: ax
;****************************************************************************

dseg segment byte public 'datasg'
extrn curl0:byte,curl1:byte,curl2:byte,d:word,d2:word
extrn d1:word,dir:byte,xinit:word,yinit:word,xfinal:word
extrn yfinal:word,gbmod:byte,p1:byte
dseg ends
cseg segment byte public 'codesg'
extrn gdc_not_busy:near,cxy2cp:near
public vector
assume cs:cseg,ds:dseg,es:dseg,ss:nothing
vector proc near

;Draw a vector.
;Assume the start and stop coordinates to be in xinit, yinit,
;xfinal, and yfinal. The Foreground/Background, ALU/PS, Mode,
;and Pattern Registers as well as the GDC PRAM bytes and all other
;incidental requirements such as "gdc_not_busy" have been taken
;care of already. This routine positions the cursor, computes the
;draw direction, dc, d, d2, d1 and then implements the actual figs
;and figd commands.

call gdc_not_busy
call cxy2cp ;convert starting x,y to a cursor position
mov al,49h ;set cursor location from curl0,1,2
out 57h,al ;issue the GDC cursor location command
mov al,curl0 ;fetch word - low address
out 56h,al
mov al,curl1 ;fetch word - middle address
out 56h,al

8-6
Vector Write Operations

```assembly
mov al,cur12 ; dot address (top 4 bits)/high address
out 56h,al
mov ax,word ptr xinit ; start and stop points the same?
cmp ax,word ptr xfinal ; jump if not
jnz v1
mov ax,word ptr yinit ; might be - check the y's
cmp ax,word ptr yfinal
jnz v1 ; jump if not
mov al,04ch ; write single pixel - current vector write
out 057h,al ; can't handle a one pixel write
mov al,2
out 056h,al
mov al,06ch
out 057h,al
ret
v1: mov bx,yfinal ; compute delta y
sub bx,yinit ; delta y negative now?
jns quad34 ; jump if not (must be quad 3 or 4)
quad12: neg bx ; delta y is negative, make absolute
mov ax,xfinal ; compute delta x
sub ax,xinit ; delta x negative?
js quad2 ; jump if yes
quad1: cmp ax,bx ; octant 2?
jbe oct3 ; jump if not
oct2: mov p1,02 ; direction of write
jmp vxind ; abs(deltax)>>abs(deltay), independent axis=x-axis
oct3: mov p1,03 ; direction of write
jmp vyind ; abs(deltax)<<abs(deltay), independent axis=y-axis
quad2: neg ax ; delta x is negative, make absolute
cmp ax,bx ; octant 4?
jae oct5 ; jump if not
oct4: mov p1,04 ; direction of write
jmp vyind ; abs(deltax)>>abs(deltay), independent axis=y-axis
oct5: mov p1,05 ; direction of write
jmp vxind ; abs(deltax)<<abs(deltay), independent axis=x-axis
quad34: mov ax,xfinal ; compute delta x
sub ax,xinit ; jump if delta x is positive
quad3: neg ax ; make delta x absolute instead of negative
cmp ax,bx ; octant 6?
jbe oct7 ; jump if not
oct6: mov p1,06 ; direction of write
jmp vxind ; abs(deltax)>>abs(deltay), independent axis=x-axis
oct7: mov p1,07 ; direction of write
jmp vyind ; abs(deltax)<<abs(deltay), independent axis=y-axis
```

8-7
Vector Write Operations

quad4: cmp ax,bx ;octant 0?
jae oct1 ;jump if not
oct0: mov p1,0 ;direction of write
jmp vyind ;abs(deltax)-abs(deltay), independent axis=y-axis
oct1: mov p1,01 ;direction of write
jmp vxind ;abs(deltax)>deltay, independent axis=x-axis
vyind: xchg ax,bx ;put independent axis in ax, dependent in bx
vxind: and ax,03fffh ;limit to 14 bits
mov dc,ax ;dc=abs(deltax)
push bx ;save abs(deltay)
shl bx,1
sub bx,ax
and bx,03fffh ;limit to 14 bits
mov d,bx ;d=2*abs(deltay)-abs(deltax)
pop bx ;restore (abs(deltay))
push bx ;save abs(deltay)
sub bx,ax
shl bx,1
and bx,03fffh ;limit to 14 bits
mov d2,bx ;d2=2*(abs(deltay)-abs(deltax))
pop bx
shl bx,1
dec bx
and bx,03fffh ;limit to 14 bits
mov d1,bx ;d1=2*abs(deltay)-1
vdo: mov al,04ch ;issue the figs command
out 57h,al
mov al,08 ;construct p1 of figs command
or al,p1
out 56h,al ;issue a parameter byte
mov si,offset dc
mov cx,08 ;issue the 8 bytes of dc,d,d2,d1
vdo1: lodsb ;fetch byte
out 56h,al ;issue to the GDC
loop vdo1 ;loop until all 8 done
mov al,06ch ;start the drawing process in motion
out 57h,al ;by issuing figd
ret
vector endp

cseg ends
Display a Circle

The example in this section will display a circle, given the radius and the coordinates of the center in pixels. The code is valid only if the option is in medium resolution mode. If this code is executed in high resolution mode, the aspect ratio would cause the output to be generated as an ellipse. As in the previous examples, the option is assumed to have been set up for a vector write operation with the appropriate type of line programmed into the Pattern Generator.

Example of Drawing a Circle

```
;*******************************************************
; procedure circle
;
; purpose:     draw a circle in medium resolution mode
;
; entry:       xinit = circle center x coordinate (0-799)
;              yinit = circle center y coordinate (0-239)
;              radius = radius of the circle in pixels
;
; caution:     This routine will only work in medium resolution mode. Due to the aspect ratio of high resolution mode, circles appear as ellipses.
;
;*******************************************************

;Draw an circle.
;This routine positions the cursor, computes the draw direction, dc, d, d2, d1 and implements the actual figs and figd commands.
;The Mode Register has been set up for graphics operations, the write mode and planes select is set up in the ALU/PS Register, the Foreground/Background Register is loaded with the desired foreground and background colors and the Pattern Multiplier/Pattern Register is loaded. In graphics mode, all incoming data comes from the Pattern Register. We have to make sure that the GDC's PRAM 8 and 9 are all ones so that it will try to write all ones to the bitmap. The external hardware intervene and put the pattern register's data into the bitmap.
```
Vector Write Operations

```
extrn gbmod:byte, curl0:byte, curl1:byte, curl2:byte, xinit:word
extrn yinit:word, dir:byte, shifts_per_line:byte
dseg segment byte public 'datasg'
  public radius, xad, yad
dc dw 0
d dw 0
d2 dw 0
d1 dw 0
dm dw 0
xad dw 0
yad dw 0
radius dw 0
dseg ends
cseg segment byte public 'codesg'
extrn gdc_not_busy:near
public circle
assume cs:cseg, ds:dseg, es:dseg, ss:nothing

; circle proc near
  call gdc_not_busy
  mov a1, 78h
  out 57h, a1 ; set pram bytes 8 and 9
  mov a1, 0ffh
  out 56h, a1
  out 56h, a1
  mov word ptr d1, -1 ; set figs d1 parameter
  mov word ptr dm, 0 ; set figs d2 parameter
  mov bx, word ptr radius ; get radius
  mov ax, 0b505h ; get 1/1.41
  inc bx
  mul bx
  mov word ptr dc, dx ; set figs dc parameter
  dec bx
  mov word ptr d, bx ; set figs d parameter
  shl bx, 1
  mov word ptr d2, bx ; set figs d2 parameter
  mov ax, word ptr xinit ; get center x
  mov word ptr xad, ax ; save it
  mov ax, word ptr yinit ; get center y
  sub ax, word ptr radius ; subtract radius
  mov word ptr yad, ax ; save it
  call acvt ; position cursor
  mov byte ptr dir, 00h ; arc 1
  call avdo ; draw it
  call acvt ; position cursor
  mov byte ptr dir, 06h ; arc 6
  call avdo ; draw it
```
Vector Write Operations

mov ax,word ptr xinit ;get center x
mov word ptr xad,ax ;save it
mov ax,word ptr yinit ;get center y
add ax,word ptr radius ;add in radius
mov word ptr yad,ax ;save it
call acvt ;position cursor
mov byte ptr dir,02h ;arc 2
call avdo ;draw it
call acvt ;position cursor
mov byte ptr dir,05h ;arc 5
call avdo ;draw it
mov ax,word ptr xinit ;get center x
sub ax,word ptr radius ;subtract radius
mov word ptr xad,ax ;save it
mov ax,word ptr yinit ;get center y
mov word ptr yad,ax ;save it
call acvt ;position cursor
mov byte ptr dir,03h ;arc 3
call avdo ;draw it
call acvt ;position cursor
mov byte ptr dir,00h ;arc 0
call avdo ;draw it
mov ax,word ptr xinit ;get center x
add ax,word ptr radius ;add in radius
mov word ptr xad,ax ;save it
mov ax,word ptr yinit ;get center y
mov word ptr yad,ax ;save it
call acvt ;position cursor
mov byte ptr dir,07h ;arc 7
call avdo ;draw it
call acvt ;position cursor
mov byte ptr dir,04h ;arc 4
call avdo ;draw it
ret

;Convert the starting x,y coordinate pair into a cursor position
;word value.
Vector Write Operations

; acvt:
mov cl, byte ptr shifts_per_line ; set up for 32/16-bit
xor dx, dx ; math - clear upper 16 bit
mov ax, word ptr yad
shr ax, cl
mov bx, ax ; save lines * word/line
mov ax, word ptr xad ; compute number of words on last line
mov cx, 16 ; 16 bits/word
div cx ; ax has number of extra words to add in
add ax, bx ; dx has the <16 dot address left over
mov cur10, al ; this is the new cursor memory address
mov cur11, ah
mov cl, 04 ; dot address is high nibble of byte
shr dl, cl
mov curl2, dl
mov al, 49h ; set cursor location to curl0,1,2
out 57h, al ; issue the GDC cursor location command
mov al, cur10 ; fetch word - low address
out 56h, al
mov al, cur11 ; fetch word - middle address
out 56h, al
mov al, cur12 ; dot address (top 4 bits)/high address
out 56h, al
ret

avdo: call gdc_not_busy ; issue the figs command
mov al, 4ch
out 57h, al
mov al, 020h ; construct p1 of figs command
mov si, offset dc
mov cx, 10 ; issue the 10 bytes of dc,d,d2,d1
mov cx, 10
out 56h, al
inc si ; point to next in list
loop avdo1 ; loop until all 10 done
mov al, 6ch ; start drawing process in motion
out 57h, al
ret

circle endp
cseg ends
end
In this chapter the examples illustrate coding for writing byte-aligned $8 \times 10$ characters, determining type and position of the cursor, and writing bit-aligned vector (stroked) characters.

Write a Byte-Aligned Character

This example uses a character matrix that is eight pixels wide and ten scan lines high. The characters are written in high resolution mode and are aligned on byte boundaries. The inputs are the column and row numbers that locate the character, the code for the character, and the color attribute.

Example of Writing a Byte-Aligned Character

```
procedure gtext

purpose: write 8 pixels wide x 10 scan lines
         graphics text in high resolution

entry:   ax is the column location of the character
         bx is the row location of the character
         dl is the character
         dh is the fgbg

segment byte public 'datasg'
```
Text Write Operations

extrn curl0:byte,curl2:byte,gbmod:byte,fg:byte

; This table has the addresses of the individual text font characters.
; Particular textab addresses are found by taking the offset of the
; textab, adding in the ASCII offset of the character to be printed
; and loading the resulting word. This word is the address of the
; start of the character’s text font.
;
gbmskl db 0
gbmskh db 0
textab dw 0
dw 10
  dw 20
  dw 30
  dw 40
  dw 50
  dw 60
  dw 70
  dw 80
  dw 90
  dw 100
  dw 110
  dw 120
  dw 130
  dw 140
  dw 150
  dw 160
  dw 170
  dw 180
  dw 190
  dw 200
  dw 210
  dw 220
  dw 230
  dw 240
  dw 250
  dw 260
  dw 270
  dw 280
  dw 290
  dw 300
  dw 310
  dw 320
  dw 330
  dw 340
  dw 350
  dw 360
  dw 370
Text Write Operations

dw 380
dw 390
dw 400
dw 410
dw 420
dw 430
dw 440
dw 450
dw 460
dw 470
dw 480
dw 490
dw 500
dw 510
dw 520
dw 530
dw 540
dw 550
dw 560
dw 570
dw 580
dw 590
dw 600
dw 610
dw 620
dw 630
dw 640
dw 650
dw 660
dw 670
dw 680
dw 690
dw 700
dw 710
dw 720
dw 730
dw 740
dw 750
dw 760
dw 770
dw 780
dw 790
dw 800
dw 810
dw 820
dw 830
dw 840
dw 850
Text Write Operations

dw 860
dw 870
dw 880
dw 890
dw 900
dw 910
dw 920
dw 930
dw 940

; text font

; space db 11111111b
db 0ffh
db 0ffh
db 0ffh
db 0ffh
db 0ffh
db 0ffh
db 0ffh
db 0ffh
db 11111111b

; exlam db 11111111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11111111b
db 11100111b
db 11111111b
db 11111111b
db 11111111b

; quote db 11111111b
db 0d7h
db 0d7h
db 0d7h
db 0ffh
db 0ffh
db 0ffh
db 0ffh
db 0ffh
db 11111111b
num    db   11111111b
       db   11010111b
       db   11010111b
       db   00000001b
       db   11010111b
       db   00000001b
       db   11010111b
       db   11010111b
       db   11111111b
       db   11111111b

dollar db   11111111b
         db   11101111b
         db   10000001b
         db   01101111b
         db   11101111b
         db   11100011b
         db   11110011b
         db   11100011b
         db   11111111b
         db   11111111b

percent db   11111111b
            db   00111101b
            db   00110111b
            db   11110111b
            db   11010111b
            db   11111111b
            db   11111111b
            db   11111111b
            db   11111111b
            db   11111111b

amp      db   11111111b
          db   10000111b
          db   01111011b
          db   10110111b
          db   11001111b
          db   10110101b
          db   01111011b
          db   10000100b
          db   11111111b
          db   11111111b
Text Write Operations

apos  db  11111111b
       db  11011111b
       db  11111111b
       db  11111111b
       db  11111111b
       db  11111111b
       db  11111111b
       db  11111111b

lefpar db  11111111b
        db  11110011b
        db  11100111b
        db  11001111b
        db  11001111b
        db  11001111b
        db  11001111b
        db  11100111b
        db  11110011b

ritpar db  11111111b
        db  11001111b
        db  11100111b
        db  11110011b
        db  11110011b
        db  11110011b
        db  11110011b
        db  11110011b

aster db  11111111b
        db  11111111b
        db  10110111b
        db  11010111b
        db  000000001b
        db  11010111b
        db  10110111b
        db  11111111b
        db  11111111b
        db  11111111b

plus db 11111111b
    db 11111111b
    db 11101111b
    db 11101111b
    db 00000001b
    db 11101111b
    db 11101111b
    db 11111111b
    db 11111111b

comma db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11001111b
    db 11100111b
    db 11001111b
    db 11111111b

minus db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b

period db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
    db 11111111b
<table>
<thead>
<tr>
<th>operation</th>
<th>db values</th>
</tr>
</thead>
<tbody>
<tr>
<td>slash</td>
<td>11111111b, 1111101b, 1111001b, 1110011b, 1100111b, 1001111b, 0011111b, 1111111b</td>
</tr>
<tr>
<td>zero</td>
<td>11111111b, 11000101b, 10010001b, 10010001b, 10001001b, 10001001b, 10011001b, 10100011b, 11111111b</td>
</tr>
<tr>
<td>one</td>
<td>11111111b, 11100111b, 11000111b, 11100111b, 11100111b, 11100111b, 11100111b, 10000001b, 11111111b</td>
</tr>
<tr>
<td>two</td>
<td>11111111b, 11000011b, 10011001b, 11111001b, 11100011b, 11001111b, 11001111b, 10011111b, 10000001b, 11111111b</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
three

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>10000001b</td>
</tr>
<tr>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td>db</td>
<td>11111001b</td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

four

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td>db</td>
<td>11100011b</td>
</tr>
<tr>
<td>db</td>
<td>11001001b</td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td>db</td>
<td>10000011b</td>
</tr>
<tr>
<td>db</td>
<td>11111001b</td>
</tr>
<tr>
<td>db</td>
<td>11111001b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

five

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>10000011b</td>
</tr>
<tr>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td>db</td>
<td>10000111b</td>
</tr>
<tr>
<td>db</td>
<td>11111001b</td>
</tr>
<tr>
<td>db</td>
<td>11111001b</td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

six

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td>db</td>
<td>10000111b</td>
</tr>
<tr>
<td>db</td>
<td>10001001b</td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>
Text Write Operations

seven db 1111111b
db 1000001b
db 1111001b
db 1111001b
db 1100111b
db 1100111b
db 1100111b
db 1100111b
db 1100111b

eight db 1111111b
db 1100011b
db 1001101b
db 1001101b
db 1100011b
db 1100011b
db 1100011b
db 1100011b
db 1100011b

nine db 1111111b
db 1100011b
db 1001101b
db 1001101b
db 1100001b
db 1100001b
db 1100001b
db 1100001b
db 1100001b

colon db 1111111b
db 1111111b
db 1111111b
db 1110011b
db 1110011b
db 1110011b
db 1110011b
db 1110011b
db 1110011b
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>scolon</td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11001111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>less</td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111001b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11001111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11001111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>1000001b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>1000001b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>greater</td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11001111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111001b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11001111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Text Write Operations

<table>
<thead>
<tr>
<th>Area</th>
<th>DB1</th>
<th>DB2</th>
<th>DB3</th>
<th>DB4</th>
<th>DB5</th>
<th>DB6</th>
<th>DB7</th>
<th>DB8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ques</td>
<td>11111111b</td>
<td>11000011b</td>
<td>10011001b</td>
<td>11111001b</td>
<td>11110011b</td>
<td>11100111b</td>
<td>11111111b</td>
<td>11111111b</td>
</tr>
<tr>
<td>at</td>
<td>11111111b</td>
<td>11000011b</td>
<td>10011001b</td>
<td>10011001b</td>
<td>10010001b</td>
<td>10010011b</td>
<td>10011111b</td>
<td>11000001b</td>
</tr>
<tr>
<td>capa</td>
<td>11111111b</td>
<td>11100111b</td>
<td>11000011b</td>
<td>11100011b</td>
<td>11011001b</td>
<td>10011001b</td>
<td>10010001b</td>
<td>10010011b</td>
</tr>
<tr>
<td>capb</td>
<td>11111111b</td>
<td>10000011b</td>
<td>10011001b</td>
<td>10011001b</td>
<td>10010011b</td>
<td>10010001b</td>
<td>10011001b</td>
<td>10010011b</td>
</tr>
</tbody>
</table>

---

9-12
### Text Write Operations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>capc</td>
<td>db 1111111b</td>
</tr>
<tr>
<td></td>
<td>db 1100001b</td>
</tr>
<tr>
<td></td>
<td>db 1001100b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1100001b</td>
</tr>
<tr>
<td></td>
<td>db 1111111b</td>
</tr>
<tr>
<td>capd</td>
<td>db 1111111b</td>
</tr>
<tr>
<td></td>
<td>db 1000001b</td>
</tr>
<tr>
<td></td>
<td>db 1001100b</td>
</tr>
<tr>
<td></td>
<td>db 1001100b</td>
</tr>
<tr>
<td></td>
<td>db 1001100b</td>
</tr>
<tr>
<td></td>
<td>db 1001100b</td>
</tr>
<tr>
<td></td>
<td>db 1001100b</td>
</tr>
<tr>
<td></td>
<td>db 1000001b</td>
</tr>
<tr>
<td></td>
<td>db 1111111b</td>
</tr>
<tr>
<td></td>
<td>db 1111111b</td>
</tr>
<tr>
<td>cape</td>
<td>db 1111111b</td>
</tr>
<tr>
<td></td>
<td>db 1000001b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1000001b</td>
</tr>
<tr>
<td></td>
<td>db 1000001b</td>
</tr>
<tr>
<td></td>
<td>db 1111111b</td>
</tr>
<tr>
<td></td>
<td>db 1111111b</td>
</tr>
<tr>
<td>capf</td>
<td>db 1111111b</td>
</tr>
<tr>
<td></td>
<td>db 1000001b</td>
</tr>
<tr>
<td></td>
<td>db 1001110b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1000011b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1001111b</td>
</tr>
<tr>
<td></td>
<td>db 1111111b</td>
</tr>
<tr>
<td></td>
<td>db 1111111b</td>
</tr>
</tbody>
</table>
Text Write Operations

capg  db  11111111
       db  11000011
       db  10011001
       db  10011001
       db  10011111
       db  10010001
       db  10011001
       db  11000011
       db  11111111

caph  db  11111111
       db  10011001
       db  10011001
       db  10011001
       db  10000001
       db  10011001
       db  10011001
       db  10011001
       db  11111111

capi  db  11111111
       db  11000011
       db  11100111
       db  11100111
       db  11100111
       db  11100111
       db  11100111
       db  11000011
       db  11111111

capj  db  11111111
       db  11100001
       db  11110011
       db  11110011
       db  11110011
       db  11110011
       db  11110011
       db  11000011
       db  11111111

9-14
<table>
<thead>
<tr>
<th>capk</th>
<th>db 11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db 10011001b</td>
</tr>
<tr>
<td></td>
<td>db 10010011b</td>
</tr>
<tr>
<td></td>
<td>db 10000111b</td>
</tr>
<tr>
<td></td>
<td>db 10001111b</td>
</tr>
<tr>
<td></td>
<td>db 10000111b</td>
</tr>
<tr>
<td></td>
<td>db 10010011b</td>
</tr>
<tr>
<td></td>
<td>db 10011001b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capl</th>
<th>db 11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db 10000111b</td>
</tr>
<tr>
<td></td>
<td>db 11001111b</td>
</tr>
<tr>
<td></td>
<td>db 11001111b</td>
</tr>
<tr>
<td></td>
<td>db 11001111b</td>
</tr>
<tr>
<td></td>
<td>db 11001111b</td>
</tr>
<tr>
<td></td>
<td>db 11001101b</td>
</tr>
<tr>
<td></td>
<td>db 10000001b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capm</th>
<th>db 11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db 00011001b</td>
</tr>
<tr>
<td></td>
<td>db 00010001b</td>
</tr>
<tr>
<td></td>
<td>db 00101001b</td>
</tr>
<tr>
<td></td>
<td>db 00101001b</td>
</tr>
<tr>
<td></td>
<td>db 00111001b</td>
</tr>
<tr>
<td></td>
<td>db 00110011b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capn</th>
<th>db 11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db 10011001b</td>
</tr>
<tr>
<td></td>
<td>db 10001001b</td>
</tr>
<tr>
<td></td>
<td>db 10001001b</td>
</tr>
<tr>
<td></td>
<td>db 10000001b</td>
</tr>
<tr>
<td></td>
<td>db 10010001b</td>
</tr>
<tr>
<td></td>
<td>db 10010001b</td>
</tr>
<tr>
<td></td>
<td>db 10011001b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
<tr>
<td></td>
<td>db 11111111b</td>
</tr>
</tbody>
</table>
### Text Write Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>DBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>capo</td>
<td>db 11111111b db 1100011b db 10011001b db 10011001b db 10011001b db 10011001b db 11000011b db 11111111b db 11111111b</td>
</tr>
<tr>
<td>capp</td>
<td>db 11111111b db 1000011b db 10011001b db 10011001b db 10000011b db 10011111b db 10011111b db 10011111b db 11111111b</td>
</tr>
<tr>
<td>capq</td>
<td>db 11111111b db 1100011b db 10011001b db 10011001b db 10011001b db 10011001b db 1001001b db 10011001b db 11000001b</td>
</tr>
<tr>
<td>capr</td>
<td>db 111111111b db 1000011b db 10011001b db 10011001b db 1000011b db 1000011b db 10010011b db 10011001b db 11111111b</td>
</tr>
</tbody>
</table>

9-16
<table>
<thead>
<tr>
<th>caps</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11000111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td>capt</td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10000001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td>capu</td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td>capv</td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>
Text Write Operations

capw  db  11111111b
      db  00111001b
      db  00111001b
      db  00111001b
      db  00111001b
      db  00101001b
      db  00000001b
      db  00111001b
      db  11111111b
      db  11111111b

capx  db  11111111b
      db  10011001b
      db  10011001b
      db  11000011b
      db  11100111b
      db  11000011b
      db  10011001b
      db  10011001b
      db  11111111b
      db  11111111b

capy  db  11111111b
       db  10011001b
       db  10011001b
       db  11000011b
       db  11100111b
       db  11100111b
       db  11100111b
       db  11000011b
       db  11111111b
       db  11111111b

capz  db  11111111b
       db  10000001b
       db  11110011b
       db  11100111b
       db  11100111b
       db  11000011b
       db  11111111b
       db  11111111b
       db  11111111b
       db  11111111b

9-18
<table>
<thead>
<tr>
<th>lbrak</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>10000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bslash</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11001111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11100111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rbrak</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>10000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11110011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>caret</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>11101111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11010111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10110111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>
Text Write Operations

underl db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 111110111b

dlsguot db 11111111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b

lita db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 10000011b
db 1111001b
db 1100001b
db 10011001b

litb db 11111111b
db 10011111b
db 10011111b
db 10011111b
db 10011111b
db 10011111b
db 10011111b
db 11111111b
<table>
<thead>
<tr>
<th>litc</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>litd</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11111001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>1100001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>1001001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>1001001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>1001001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>1100001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lite</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10011001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11000011b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>litf</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>11110011b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11001001b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11001111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>10000011b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11001111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11001111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11001111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>11111111b</td>
<td></td>
</tr>
</tbody>
</table>
Text Write Operations

litg      db  11111111b
            db  11111111b
            db  11111001b
            db  11000001b
            db  10010011b
            db  10010011b
            db  11000011b
            db  11110011b
            db  11000111b
            db  11000111b

lith      db  11111111b
            db  11111111b
            db  10011111b
            db  10011111b
            db  10000011b
            db  10001001b
            db  10011001b
            db  10011001b
            db  10011001b
            db  11111111b
            db  11111111b

litl      db  11111111b
            db  11111111b
            db  11111111b
            db  11110011b
            db  11100111b
            db  11100111b
            db  11100111b
            db  11100111b
            db  10000001b
            db  11111111b
            db  11111111b

litj      db  11111111b
            db  11111111b
            db  11110011b
            db  11110011b
            db  11110011b
            db  11110011b
            db  11110011b
            db  11110011b
            db  11110011b
            db  10010011b
            db  11000111b

9-22
litk db 11111111b
db 10011111b
db 10011111b
db 10010011b
db 10000111b
db 10000111b
db 10010011b
db 11111111b
db 11111111b

litl db 11111111b
db 11000111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11000011b
db 11111111b
db 11111111b

litm db 11111111b
db 11111111b
db 11111111b
db 10010011b
db 0010001b
db 0010001b
db 0010001b
db 00111001b
db 11111111b
db 11111111b

litm db 11111111b
db 11111111b
db 11111111b
db 10100011b
db 10010011b
db 10010011b
db 10010011b
db 10010011b
db 11111111b
db 11111111b
### Text Write Operations

<table>
<thead>
<tr>
<th>lito</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>litp</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10100011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10001001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10001001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10000011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>litq</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>1100001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10010001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10010001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>1100001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111001b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>litr</th>
<th>db</th>
<th>11111111b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10100011b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011001b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>10011111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>11111111b</td>
</tr>
</tbody>
</table>

9-24
Text Write Operations

lits  db  11111111b
      db  11111111b
      db  11111111b
      db  1100001tb
      db  10011111b
      db  11000011b
      db  11111001b
      db  10000011b
      db  11111111b
      db  11111111b

litt  db  11111111b
      db  11111111b
      db  11001111b
      db  10000011b
      db  11001111b
      db  11001111b
      db  11001001b
      db  11100011b
      db  11111111b
      db  11111111b

litu  db  11111111b
      db  11111111b
      db  11111111b
      db  10011001b
      db  10011001b
      db  10011001b
      db  10011001b
      db  11000011b
      db  11111111b
      db  11111111b

litv  db  11111111b
      db  11111111b
      db  11111111b
      db  10011001b
      db  10011001b
      db  10011001b
      db  10011001b
      db  11011011b
      db  11100111b
      db  11111111b
      db  11111111b

9-25
Text Write Operations

litw  db  11111111b
       db  11111111b
       db  11111111b
       db  00111001b
       db  00111001b
       db  00101001b
       db  10101011b
       db  10010011b
       db  11111111b
       db  11111111b

litx  db  11111111b
       db  11111111b
       db  11111111b
       db  10011001b
       db  11000011b
       db  11100111b
       db  11000011b
       db  10011001b
       db  11111111b
       db  11111111b

lity  db  11111111b
       db  11111111b
       db  11111111b
       db  10011001b
       db  10011001b
       db  10011001b
       db  11000011b
       db  11100111b
       db  11001111b
       db  10000001b
       db  11111111b

litz  db  11111111b
       db  11111111b
       db  11111111b
       db  10000001b
       db  11110011b
       db  11100111b
       db  11001111b
       db  10000001b
       db  11111111b
       db  11111111b
Text Write Operations

lsbrak db 11111111b
db 11110001b
db 11100111b
db 11001111b
db 10011111b
db 11001111b
db 11001111b
db 11100011b
db 11111111b
db 11111111b

vertl db 11111111b
db 11110011b
db 11110011b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b
db 11100111b

rsbrak db 11111111b
db 11100111b
db 11100111b
db 11110011b
db 11110011b
db 11110011b
db 11110011b
db 11110011b
db 11110011b
db 11110011b
db 11110011b

tilde db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b
db 11111111b

dseg ends
cseg segment byte public 'codesg'
    public gtext
extrn mode:near,gdc_not_busy:near
    assume cs:cseg,ds:dseg,es:dseg,ss:nothing
gtext    proc near
Text Write Operations

; We are going to assume that the character is byte-aligned. Anything else will be ignored with the char being written out to the integer of the byte address.

; Special conditions: if dl=0ffh - don’t print anything.

; 1) Make sure that the Graphics Option doesn’t have any pending operations to be completed.
; 2) Turn the x,y coordinates passed in ax,bx into a cursor word address to be saved and then asserted to the GDC.
; 3) If the current foreground/background colors are not those desired, assert the desired colors to the Foreground/Background Register.
; 4) Determine in which half of the word the character is to be written to and then enable that portion of the write.
; 5) Check to see if the character we are being requested to print is legal. Anything under 20h is considered to be unprintable and so we just exit. We also consider 0ffh to be unprintable since the Rainbow uses this code as a delete marker.
; 6) Turn the character’s code into a word offset. Use this offset to find an address in a table. This table is a table of near addresses that define the starting address of the ten bytes that is the particular character’s font. Fetch the first two bytes and assert to the screen. We have to assert write buffer counter reset because we are only using two of the words in the write buffer, not all 8.
; Each byte is loaded into both the left and right bytes of the write buffer word. The GDC is programmed to perform the two-scan-line write and we wait for the write to finish. The next 8 scan lines of the character font are loaded into both the left and right bytes of the write buffer and then written to the screen.

    push  ax
    call  gdc_not_busy
    pop   ax

; Ax = the column number of the character. Bx is the row number.
; In high resolution, each bx is = 640 words
; Cursor position = (ax/2)+10*(bx*scan line width in words)

    mov   di,ax  ; save the x so that we can check it later
    shr   ax,1   ; turn column position into a word address
    mov   cx,6   ; high resolution is 64 words per line
    shl   bx,cl  ; bx*scan line length
    mov   si,bx  ; save a copy of scan times count
    mov   cl,3   ; to get bx*10 first multiply bx by 8
    shl   bx,cl  ; then
    add   bx,si  ; add in the 2*bx*scan line length
add bx,sl ;this gives 10*bx*scan line length
add bx,ax ;combine x and y into a word address
mov word ptr curl0,bx ;position to write the word at

;Assert the colors attributes of the character to fgbg. Dh has the
;foreground and background attributes in it.
;
cmp dh,byte ptr fg ;is the fgbg color the one we want?
jz cont ;jump if yes
mov al,0f7h
out 53h,al
mov byte ptr fg,dh
mov al,dh
out 51h,al
;
;Assert the graphics board's text mask. The GDC does 16-bit writes
;in text mode but our characters are only 8 bits wide. We must enable
;half of the write and disable the other half. If the x was odd then
;enable the right half. If the x was even then enable the left half.
;
cont: test dl,1 ;is this a first byte?
jnz odd ;jump if not
mov word ptr gbmskl,00ffh
jmp com

odd: mov word ptr gbmskl,0ff00h
com: call stgbm ;assert the graphics board mask

;Only the characters below 127h are defined - the others are legal
;but not in the font table. After checking for a legal character
;fetch the address entry (character number - 20h) in the table.
;This is the address of the first byte of the character's font.
;
cmp dl,1fh ;unprintable character?
ja cont0 ;jump if not
jmp exit ;don't print illegal character

cont0: cmp dl,0ffh ;is this a delete marker?
jnz cont1 ;jump if not
jmp exit ;exit if yes

cont1: sub dl,20h ;table starts with a space
xor dh,dh ;at 0
mov bx,dx ;access table & index off bx
shl bx,1 ;byte to word address offset
mov si,offset textab

;Textab has the relative offsets of each character in it. All we have
;to do is add the start of the font table to the relative offset of
;the particular character.
Text Write Operations

; add si,offset space ;combine table offset with ;character offset
;
; Transfer the font from the font table into the write buffer.
; Write the first two scans, then do the last 8.
;
; cld
; make sure lodsb incs si.
 mov al,0feh
; reset the write buffer counter
 out 53h,al
 out 51h,al
 lodsw
; fetch both bytes.
 out 52h,al
 out 52h,al
; write buffer bytes
 mov al,ah
 out 52h,al
 out 52h,al
; write buffer bytes
 mov al,0feh
; reset the write buffer counter
 out 53h,al
 out 51h,al
;
; Check to see if already in text mode.
;
 test byte ptr gbmod,2
 jz textm ; jump if in text mode else
 and byte ptr gbmod,0fdh ; assert text mode
 call mode

textm: mov al,49h ; assert the cursor command
 out 57h,al
 mov ax,word ptr cur10
 out 56h,al
 mov al,ah
 out 56h,al
 mov al,4ah ; assert the mask command
 out 57h,al
 mov al,0ffh
 out 56h,al
 out 56h,al
 mov al,4ch ; assert the figs command
 out 57h,al
 xor al,al ; assert the down direction to write
 out 56h,al
 mov al,1 ; do it 2 write cycles
 out 56h,al
 xor al,al
 out 56h,al

9-30
mov al,22h ; assert the wdat command
out 57h,al
mov al,0ffh
out 56h,al
out 56h,al

; Wait for the first two scans to be written.

mov ax,422h ; make sure the GDC isn't drawing
out 57h,al ; write a wdat to the GDC
here1: in al,56h ; read the status register
test ah,al ; did the wdat get executed?
jz here1 ; jump if not

; si is still pointing to the next scan line to be fetched. Get the
; next two scan lines and then tell the GDC to write them. No new
; cursor, GDC mask, graphics mask or mode commands need be issued.

mov cx,8 ; eight scan lines
ldcr: lodsb ; fetch the byte
out 52h,al ; put the byte into both 1 and 2
out 52h,al ; write buffer bytes
loop ldcr
mov al,4ch ; assert the figs command
out 57h,al
xor al,al ; assert the down direction to write
out 56h,al
mov ax,7 ; do 8 write cycles
out 56h,al
mov al,ah
out 56h,al
mov al,22h ; assert the wdat command
out 57h,al
mov al,0ffh
out 56h,al
out 56h,al
exit: ret

stgbm: mov ax,word ptr gbmskl
out 54h,al
mov al,ah
out 55h,al
ret
gtext endp
cseg ends
end

9-31
Define and Position the Cursor

There are two routines in the following example. One sets the cursor type to no cursor, block, underscore, or block and underscore. It then sets up the current cursor location and calls the second routine. The second routine accepts new coordinates for the cursor and moves the cursor to the new location.

Example of Defining and Positioning the Cursor

```
;*******************************************************
; ; procedure gsettyp
; ; purpose: assert new cursor type
; ; entry: dl bits determine cursor style
; ; (if no bits set, no cursor is displayed)
; ; bit 0 = block
; ; bit 1 = undefined
; ; bit 2 = undefined
; ; bit 3 = underscore
; ;*******************************************************
;
 dseg segment byte public 'datalog'
 extrn curl0:byte,curl2:byte,gbmod:byte
 block db 0,0,0,0,0,0,0,0,0,0
 cdls db 0
 lastcl dw 0
 dw 0
 ocurs db 0
 newcur dw 0
 dw 0
 ncurs db 0
 unders db offh,offh,offh,offh,offh,offh,offh,offh,offh,offh
 userrd db 0,0,0,0,0,0,0,0,0,0
 dseg ends

 ;implements the new cursor type to be displayed. The current
 ;cursor type and location must become the old type and location.
 ;The new type becomes whatever is in dl. This routine will fetch
 ;the previous cursor type out of NCURS and put it into OCURS and
 ;then put the new cursor type into NCURS. The previous cursor
 ;coordinates are fetched and put into ax and bx. A branch to
 ;GSETPOS then erases the old cursor and displays the new cursor.
 ;Cursor type bits are not exclusive of each other. A cursor can
 ;be both an underscore and a block.
```
; dl = 0 - turns the cursor display off
; 1 - displays the insert cursor (full block)
; 8 - displays the overwrite cursor (underscore)
; 9 - displays a simultaneous underscore and block cursor

; cseg segment byte public ‘codesg’
extrn mode:near
assume cs:cseg,ds:dseg,es:dseg,ss:nothing
public gsettyp

; gsettyp proc near
mov al,byte ptr ncurs ; current cursor becomes
mov byte ptr ocurs,al ; old cursor type
mov byte ptr ncurs,dl ; pick up new cursor type
mov ax,word ptr newcl ; pick up current x and y
mov bx,word ptr newcl+2 ; cursor coordinates
jmp pos ; branch to assert new cursor

; type in old location

; ********************************************************************
; procedure gsetpos
; purpose: assert new cursor position
; entry: ax = x location
; bx = y location
; ********************************************************************

public gsetpos

gsetpos proc near

; Display the cursor. Cursor type was defined by GSETTYP. The
; cursor type is stored in NCURS. Fetch the type and address of the
; previous cursor and put it into OCURS and also into lastcl and
; lastcl+2. If a cursor is currently being displayed, erase it. If
; there is a new cursor to display, write it (or them) to the screen.
; A cursor may be a block or an underscore or both.
;
; The x and y coordinates of the cursor are converted into an address
; that the GDC can use. Either the left or the right half of the text
; mask is enabled, depending on whether the x is even or odd. The
; write operation itself takes places in complement mode so that no
; information on the screen is lost or obscured but only inverted in
; value. In order to ensure that all planes are inverted, a 0f0h is
; loaded into the Foreground/Background Register and all planes are
; write enabled. The cursor is written to the screen in two separate
; writes because the write buffer is eight, not ten, words long.

9-33
Text Write Operations

; Move current cursor type and location to previous type and location.

mov cl, byte ptr ncurs ; move current cursor type
mov byte ptr ocur, cl ; into old cursor type
pos:
cld
mov cx, word ptr newcl ; move current cursor
mov word ptr lastcl, cx ; location into old cursor
mov cx, word ptr newcl+2 ; location
mov word ptr lastcl+2, cx
mov word ptr newcl, ax ; save new cursor coordinates
mov word ptr newcl+2, bx ; in new cursor location

; Before doing anything to the graphics option we need to make sure
; that the option isn’t already in use. Assert a harmless command
; into the FIFO and wait for the GDC to execute it.

call not_busy

; Set up the graphics option. Put the Graphics Option in complement
; and text modes with all planes enabled. Assert fgbg and text mask.
; Calculate the write address and store in curIO, 1.

mov ax, 10efh ; address the ALU/PS
out 53h, al ; register
mov al, ah ; set complement mode with
out 51h, al ; all planes enabled

; Assert text mode with read disabled.

mov al, byte ptr gbmod ; get mode shadow byte
and al, 0fgh ; set text mode
or al, 10h ; set write enabled mode
cmp al, byte ptr gbmod ; is mode already asserted
jz gpos0 ; this way? if yes, jump
mov byte ptr gbmod, al ; update the mode register
call mode

gpos0: mov al, 0f7h ; set Foreground/Background
out 53h, al ; register to invert data
mov al, 0f0h
out 51h, al

; Is a cursor currently being displayed? If cdis<>0, then yes. Any
; current cursor will have to be erased before we display a new one.

gpos1: test byte ptr cdis, 1 ; if no old cursor to erase,
jz gpos2 ; just display old one

9-34
Text Write Operations

; This part will erase the old cursor.

mov byte ptr cdis,0 ; set no cursor on screen
mov dh,byte ptr lastcl ; fetch x and y, put into dx,
mov dl,byte ptr lastcl+2 ; and call dx2curl
call asmask ; assert the mask registers
call dx2curl1 ; turn dx into GDC address
test byte ptr ocurs,8 ; underline?
jz gspos1 ; jump if not
mov si,offset unders ; erase the underline
call discurs ; do the write
gspos1: test byte ptr ocurs,1 ; block?
jz gspos2 ; jump if not
call not_busy ; wait till done erasing underline
mov si,offset block ; erase the block
call discurs ; do the write

gspos2: cmp byte ptr ncurs,0 ; write a new cursor?
jz gspos5 ; jump if not
mov dh,byte ptr newcl ; fetch coordinates of
mov dl,byte ptr newcl+2 ; new cursor
call not_busy ; wait for erase to finish
call asmask ; assert the mask registers
call dx2curl1 ; turn dx into GDC address
test byte ptr ncurs,8 ; underscore cursor?
jz gspos3 ; jump if not
mov si,offset unders ; set up for underline cursor
call discurs ; do the write
gspos3: test byte ptr ncurs,1 ; block cursor?
jz gspos4 ; jump if not
call not_busy ; wait for any write to finish
mov si,offset block ; set up for block cursor
call discurs ; do the write.
gspos4: or byte ptr cdis,1 ; set cursor displayed flag
gspos5: call not_busy
ret

; Enable one byte of the text mask.

asmask: mov ax,0ffh ; set up the text mask
test dh,1 ; write to the right byte?
jz ritc4 ; jump if yes
mov ax,0ff00h
ritc4: out 55h,al ; issue low byte of mask
mov al,ah
out 54h,al ; issue high byte of mask
ret
Text Write Operations

; Display the cursor.
;
; Assume that the option is already set up in text mode, complement
; write and that the appropriate text mask is already set. The
; address of the cursor pattern is loaded into the si.
;
; discurs:

    mov al,0feh          ; select the write buffer and clear
    out 53h,al           ; the write buffer counter
    out 51h,al
    lodsb
    out 52h,al           ; feed the same byte to both halves
    out 52h,al           ; of the word to be written
    lodsb
    out 52h,al           ; feed the same byte to both halves
    out 52h,al           ; of the word to be written
    mov al,0feh          ; select the write buffer and clear
    out 53h,al           ; the write buffer counter
    out 51h,al
    mov al,49h           ; assert the position to write
    out 57h,al
    mov ax,word ptr curIO
    out 56h,al
    mov al,ah
    out 56h,al
    mov al,4ah           ; issue the GDC mask command to
    out 57h,al           ; set all GDC mask bits
    mov al,0ffh
    out 56h,al
    out 56h,al
    mov al,4ch           ; program a write of ten scans
    out 57h,al           ; first do two scans, then eight
    xor al,al
    out 56h,al
    mov al,1
    out 56h,al
    xor al,al
    out 56h,al
    mov al,22h           ; start the write
    out 57h,al
    mov al,0ffh
    out 56h,al
    out 56h,al
    call not_busy        ; wait for first two lines to finish
    mov cx,8             ; then write the next 8 scans
ritc6:  lodsb          ;fetch the cursor shape
       out 52h,al     ;feed the same byte to both halves
       out 52h,al    ; of the word
       loop rict6
       mov al,4ch     ;program a write of eight scans
       out 57h,al
       xor al,al
       out 56h,al
       mov al,7
       out 56h,al
       xor al,al
       out 56h,al
       mov al,22h     ;start the write
       out 57h,al
       mov al,0ffh
       out 56h,al
       out 56h,al
       ret
       ;Turn dh and dl into a word address (dl is the line and dh
       ;is the column). Store the result in word ptr cur10. Start with
       ;turning dl (line) into a word address.
       ;
       ; Word address = dl * number of words/line * 10
       ;
       ;Turn dh (column) into a word address.
       ;
       ; Word address = dh/2
       ;
       ;Combine the two. This gives the cur10 address to be asserted to
       ;the GDC.
       ;
       ;
       ;dx2curl:
       mov al,dh         ;store the column count
       mov cl,5           ;medium resolution = 32 words/line
       test byte ptr gbmod,1   ;is it high resolution?
       jz rictc5          ;jump if not
       inc cl             ;high resolution = 64 words/line
       rictc5: xor dh,dh
       shl dx,cl
       mov bx,dx          ;multiply dx by ten
       mov cl,3
       shl bx,1
       shl dx,cl
       add dx,bx          ;this is the row address
       shr al,1           ;this is the column number

9-37
Text Write Operations

xor ah,ah
add dx,ax ;this is the combined row and
mov word ptr cur10,dx ;column address
ret

;This is a quicker version of GDC_NOT_BUSY. We don’t waste time on
;some of the normal checks and things that GDC_NOT_BUSY does due to
;the need to move as quickly as possible on the cursor erase/write
;routines. This routine does the same sort of things. A harmless
;command is issued to the GDC. If the GDC is in the process of
;performing some other command, the WDAT we just issued
;will stay in the GDC’s command FIFO until such time as the GDC can
;get to it. If the FIFO empty bit is set, the GDC executed the
;WDAT command and must be finished with any previous operations
;programmed into it.

not_busy:
  mov ax,422h ;assert a WDAT
  out 57h,al
busy:
  in al,56h ;wait for FIFO empty bit
test ah,al
  jz busy
ret

getpos endp

cseg
ends

end

Write a Text String

The example in this section writes a string of ASCII text starting at a specified location and using a
specified scale factor. It uses the vector write routine from Chapter 8 to form each character.

Example of Writing a Text String

*************************************************************************
procedure vector_text
*************************************************************************

; entry: cx = string length
; text = pointer to externally defined array of
; ASCII characters
; scale = character scale
; xinit = starting x location
; yinit = starting y location
*************************************************************************
; cseg segment byte public 'codesg'
extrn imode:near,pattern_mult:near,pattern_register:near
extrn vector:near
public vector_text
assume cs:cseg,ds:dseg,es:dseg,ss:nothing
;
vector_text proc near
or byte ptr gbmod,082h
; ensure we're in graphics mode
mov al,4ah
out 57h,al
mov al,0ffh
out 56h,al
out 56h,al
; enable GDC mask data write
xor al,al
out 55h,al
out 54h,al
mov bl,1
; set pattern multiplier
mov bl,0ffh
; set pattern register
mov ax,word ptr xinit
; get initial x
mov word ptr xad,ax
; save it
mov ax,word ptr yinit
; get initial y
mov word ptr yad,ax
; save it
mov si,offset text
 ;get character
push si
push cx
; display it
mov ax,8
mov cl,byte ptr scale
;move over by cell value
mul cx
add word ptr xad,ax
pop cx
pop si
loop do_string ;loop until done
ret
 ;display character:
do_string:
    lodsb
    push si
    push cx
    call display_character
    mov ax,8
    mov cl,byte ptr scale
    ;move over by cell value
    mul cx
    add word ptr xad,ax
    pop cx
    pop si
    loop do_string
    ret
    display_character:
    cmp al,07fh
    ;make sure we're in font table
    jbe char_cont_1
    ;continue if we are
    ret
    char_cont_1:
    cmp al,20h
    ;check if we can print character
    ja char_cont
    ;continue if we can
    ret

9-39
char_cont:
    xor ah,ah          ; clear high byte
    shl ax,1           ; make it a word pointer
    mov si,ax
    mov si,font_table[sil] ; point si to font info
get_next_stroke:
    mov ax,word ptr xad
    mov word ptr xinit,ax
    mov ax,word ptr yad
    mov word ptr yinit,ax
    lodsb
    cmp al,endc         ; end of character ?
    jnz ret
    cont_1: mov bx,ax
    and ax,0fh          ; mask to y value
    test al,0fh          ; negative ?
    jz ct
    or ax,0fff0h         ; sign extend
ct:    mov cl,byte ptr scale
    xor ch,ch
    push cx
    imul cx
    sub word ptr yinit,ax ; subtract to y offset
    and bx,0f0h          ; mask to x value
    shr bx,1
    shr bx,1             ; shift to four least
    shr bx,1
    test bl,08h          ; negative ?
    jz ct1
    or bx,0fff0h         ; sign extend
ct1:   mov ax,bx
    pop cx
    imul cx
    add word ptr xinit,ax ; add to x offset
Text Write Operations

```
next_stroke:
    mov    ax,word ptr xad        ;set up xy offsets
    mov    word ptr xfinal,ax
    mov    ax,word ptr yad
    mov    word ptr yfinal,ax
lodsb
    cmp    al,endc                ;end of character?
    jz     display_char_exit      ;yes then leave
    cmp    al,endv                ;dark vector?
    jz     get_next_stroke
    mov    bx,ax
    and    ax,0fh                 ;mask to y value
    test   al,08h                 ;negative
    jz     ct2
    or     ax,0ff0h               ;sign extend
ct2:
    mov    cl,byte ptr scale      ;get scale information
    xor    ch,ch
    push   cx
    imul   cx                      ;multiply by scale
    sub    word ptr yfinal,ax     ;subtract to y offset
    and    bx,0f0h                ;mask to x value
    shr    bx,1                   ;shift to four least
    shr    bx,1
    shr    bx,1
    test   bl,08h                 ;negative?
    jz     ct3
    or     bx,0ff0h               ;sign extend
ct3:
    mov    ax,bx                   ;recover scale
    imul   cx                      ;multiply by scale
    add    word ptr xfinal,ax     ;add to x offset
    push   si                      ;save index to font info
    call   vector                 ;draw stroke
    pop    si                      ;recover font index
    mov    ax,word ptr xfinal     ;end of stroke becomes
    mov    word ptr xinit,ax      ;beginning of next stroke
    mov    ax,word ptr yfinal
    mov    word ptr yinit,ax
    jmp    next_stroke

display_char_exit:
    ret
vector_text endp
;
cseg    ends
dseg    segment byte    public 'datasg'
extrn   gbmod:byte,xinit:word,yinit:word,xfinal:word,yfinal:word
extrn   xad:word,yad:word,text:byte
public   scale
```
; The following tables contain vertex data for a stroked character set. The x and y coordinate information is represented by 4-bit, 2's-complement numbers in the range of + or - 7. The x and y bit positions are as follows:

; bit 7 6 5 4 3 2 1 0
; \ / \ / \ /
; x y

; End of character is represented by the value x = -8, y = -8.
; The dark vector is represented by x = -8, y = 0.

; ASCII characters are mapped into the positive quadrant, with the origin at the lower left corner of an upper case character.

; endc equ 10001000b ; end of character
endv equ 10000000b ; last vector of polyline

font_table dw offset font_00
        dw offset font_01
        dw offset font_02
        dw offset font_03
        dw offset font_04
        dw offset font_05
        dw offset font_06
        dw offset font_07
        dw offset font_08
        dw offset font_09
        dw offset font_0a
        dw offset font_0b
        dw offset font_0c
        dw offset font_0d
        dw offset font_0e
        dw offset font_0f
        dw offset font_10
        dw offset font_11
        dw offset font_12
        dw offset font_13
        dw offset font_14
        dw offset font_15

9-42
Text Write Operations

dw offset font_16

dw offset font_17

dw offset font_18

dw offset font_19

dw offset font_1a

dw offset font_1b

dw offset font_1c

dw offset font_1d

dw offset font_1e

dw offset font_1f

dw offset font_20 ;space

dw offset font_21 ;!

dw offset font_22

dw offset font_23

dw offset font_24

dw offset font_25

dw offset font_26

dw offset font_27

dw offset font_28

dw offset font_29

dw offset font_2a

dw offset font_2b

dw offset font_2c

dw offset font_2d

dw offset font_2e

dw offset font_2f

dw offset font_30

dw offset font_31

dw offset font_32

dw offset font_33

dw offset font_34

dw offset font_35

dw offset font_36

dw offset font_37

dw offset font_38

dw offset font_39

dw offset font_3a

dw offset font_3b

dw offset font_3c

dw offset font_3d

dw offset font_3e

dw offset font_3f

dw offset font_40

dw offset font_41

dw offset font_42

dw offset font_43
Text Write Operations

dw offset font_44

dw offset font_45

dw offset font_46

dw offset font_47

dw offset font_48

dw offset font_49

dw offset font_4a

dw offset font_4b

dw offset font_4c

dw offset font_4d

dw offset font_4e

dw offset font_4f

dw offset font_50

dw offset font_51

dw offset font_52

dw offset font_53

dw offset font_54

dw offset font_55

dw offset font_56

dw offset font_57

dw offset font_58

dw offset font_59

dw offset font_5a

dw offset font_5b

dw offset font_5c

dw offset font_5d

dw offset font_5e

dw offset font_5f

dw offset font_60

dw offset font_61

dw offset font_62

dw offset font_63

dw offset font_64

dw offset font_65

dw offset font_66

dw offset font_67

dw offset font_68

dw offset font_69

dw offset font_6a

dw offset font_6b

dw offset font_6c

dw offset font_6d

dw offset font_6e

dw offset font_6f

dw offset font_70

dw offset font_71

dw offset font_72

dw offset font_73
Text Write Operations

dw offset font_74
dw offset font_75
dw offset font_76
dw offset font_77
dw offset font_78
dw offset font_79
dw offset font_7a
dw offset font_7b
dw offset font_7c
dw offset font_7d
dw offset font_7e
dw offset font_7f

;
font_00 db endc
font_01 db endc
font_02 db endc
font_03 db endc
font_04 db endc
font_05 db endc
font_06 db endc
font_07 db endc
font_08 db endc
font_09 db endc
font_0a db endc
font_0b db endc
font_0c db endc
font_0d db endc
font_0e db endc
font_0f db endc
font_10 db endc
font_11 db endc
font_12 db endc
font_13 db endc
font_14 db endc
font_15 db endc
font_16 db endc
font_17 db endc
font_18 db endc
font_19 db endc
font_1a db endc
font_1b db endc
font_1c db endc
font_1d db endc
font_1e db endc
font_1f db endc
font_20 db endc

; space
Text Write Operations

font_21 db 20h, 21h, endv, 23h, 26h, endc
font_22 db 24h, 26h, endv, 54h, 56h, endc
font_23 db 20h, 26h, endv, 40h, 46h, endv, 04h, 64h, endv, 02h, 62h
    db endc
font_24 db 2fh, 27h, endv, 01h, 10h, 30h, 41h, 42h, 33h, 13h, 04h, 05h
    db 16h, 36h, 045h, endc
font_25 db 11h, 55h, endv, 14h, 15h, 25h, 24h, 14h, endv, 41h, 51h, 52h
    db 42h, 41h, endc
font_26 db 50h, 14h, 15h, 26h, 36h, 45h, 44h, 11h, 10h, 30h, 52h, endc
font_27 db 34h, 36h, endc
font_28 db 4eh, 11h, 14h, 47h, endc
font_29 db 0eh, 31h, 34h, 07h, endc
font_2a db 30h, 36h, endv, 11h, 55h, endv, 15h, 51h, endv, 03h, 63h
    db endc
font_2b db 30h, 36h, endv, 03h, 63h, endc
font_2c db 11h, 20h, 2fh, 0dh, endc
font_2d db 03h, 63h, endc
font_2e db 00h, 01h, 11h, 10h, 00h, endc
font_2f db 00h, 01h, 45h, 46h, endc
font_30 db 01h, 05h, 16h, 36h, 45h, 41h, 30h, 10h, 01h, endc
font_31 db 04h, 26h, 20h, endv, 00h, 040h, endc
font_32 db 05h, 16h, 36h, 45h, 44h, 00h, 40h, 041h, endc
font_33 db 05h, 16h, 36h, 45h, 44h, 33h, 42h, 41h, 30h, 10h, 01h, endv
    db 13h, 033h, endc
font_34 db 06h, 03h, 043h, endv, 20h, 026h, endc
font_35 db 01h, 10h, 30h, 41h, 42h, 33h, 03h, 06h, 046h, endc
font_36 db 02h, 13h, 33h, 42h, 41h, 30h, 10h, 01h, 05h, 16h, 36h, 045h
    db endc
font_37 db 06h, 46h, 44h, 00h, endc
font_38 db 01h, 02h, 13h, 04h, 05h, 16h, 36h, 45h, 44h, 33h, 42h, 41h
    db 30h, 10h, 01h, endv, 13h, 023h, endc
font_39 db 01h, 10h, 30h, 04h, 13h, 044h, endc
font_3a db 15h, 25h, 24h, 14h, 15h, endv, 12h, 22h, 21h, 11h, 12h
    db endc
font_3b db 15h, 25h, 24h, 14h, 15h, endv, 21h, 11h, 12h, 22h, 20h, 1fh
    db endc
font_3c db 30h, 03h, 036h, endc
font_3d db 02h, 042h, endv, 04h, 044h, endc
font_3e db 10h, 43h, 16h, endc
font_3f db 06h, 17h, 37h, 46h, 45h, 34h, 24h, 022h, endv, 21h, 020h
    db endc
font_40 db 50h, 10h, 01h, 06h, 17h, 57h, 66h, 63h, 52h, 32h, 23h, 24h
    db 35h, 55h, 064h, endc
font_41 db 00h, 04h, 26h, 44h, 040h, endv, 03h, 043h, endc
font_42 db 00h, 06h, 36h, 45h, 44h, 33h, 42h, 41h, 30h, 00h, endv
    db 03h, 033h, endc

9-46
Text Write Operations

font_43 db 45h, 36h, 16h, 05h, 01h, 10h, 30h, 041h, endc
font_44 db 00h, 06h, 36h, 45h, 41h, 30h, 00h, endc
font_45 db 40h, 00h, 06h, 046h, endv, 03h, 023h, endc
font_46 db 00h, 06h, 046h, endv, 03h, 023h, endc
font_47 db 45h, 36h, 16h, 05h, 01h, 10h, 30h, 41h, 43h, 023h, endc
font_48 db 00h, 06h, endv, 03h, 043h, endv, 40h, 046h, endc
font_49 db 10h, 030h, endv, 20h, 026h, endv, 16h, 036h, endc
font_4a db 01h, 10h, 30h, 41h, 046h, endc
font_4b db 00h, 06h, endv, 02h, 046h, endv, 13h, 040h, endc
font_4c db 40h, 00h, 06h, endc
font_4d db 00h, 06h, 24h, 046h, 040h, endc
font_4e db 00h, 06h, endv, 05h, 041h, endv, 40h, 046h, endc
font_4f db 01h, 05h, 16h, 36h, 45h, 41h, 30h, 10h, 01h, endc
font_50 db 00h, 06h, 36h, 45h, 44h, 33h, 03h, endc
font_51 db 12h, 30h, 10h, 01h, 05h, 16h, 36h, 45h, 41h, 30h, endc
font_52 db 00h, 06h, 36h, 45h, 44h, 33h, 03h, endv, 13h, 040h, endc
font_53 db 01h, 10h, 30h, 41h, 42h, 33h, 13h, 04h, 05h, 16h, 36h
    db 045h, endc
font_54 db 06h, 046h, endv, 20h, 026h, endc
font_55 db 06h, 01h, 10h, 30h, 41h, 046h, endc
font_56 db 06h, 02h, 20h, 42h, 046h, endc
font_57 db 06h, 00h, 22h, 40h, 046h, endc
font_58 db 00h, 01h, 45h, 046h, endv, 40h, 41h, 05h, 06h, endc
font_59 db 06h, 24h, 020h, endv, 24h, 46h, endc
font_5a db 06h, 46h, 45h, 01h, 00h, 40h, endc
font_5b db 37h, 17h, 1fh, 3fh, endc
font_5c db 06h, 05h, 41h, 40h, endc
font_5d db 17h, 37h, 3fh, 2fh, endc
font_5e db 04h, 26h, 044h, endc
font_5f db 0fh, 07fh, endc
font_60 db 54h, 36h, endc
font_61 db 40h, 43h, 34h, 14h, 03h, 01h, 10h, 30h, 041h, endc
font_62 db 06h, 01h, 10h, 30h, 41h, 43h, 34h, 14h, 03h, endc
font_63 db 41h, 30h, 10h, 01h, 03h, 14h, 34h, 043h, endc
font_64 db 46h, 41h, 30h, 10h, 01h, 03h, 14h, 34h, 43h, endc
font_65 db 41h, 30h, 10h, 01h, 03h, 14h, 34h, 43h, 42h, 02h, endc
font_66 db 20h, 25h, 36h, 46h, 55h, endv, 03h, 43h, endc
font_67 db 41h, 30h, 10h, 01h, 03h, 14h, 34h, 43h, 4fh, 3eh, 1eh
    db 0fh, endc
Text Write Operations

font_68 db 00h,06h,endv,03h,14h,34h,43h,40h,endc
font_69 db 20h,23h,endv,25h,26h,endc
font_6a db 46h,45h,endv,43h,4fh,3eh,1eh,0fh,endc
font_6b db 00h,06h,endv,01h,34h,endv,12h,30h,endc
font_6c db 20h,26h,endc
font_6d db 00h,04h,endv,03h,14h,23h,34h,43h,40h,endc
font_6e db 00h,04h,endv,03h,14h,34h,43h,40h,endc
font_6f db 01h,03h,14h,34h,43h,41h,30h,10h,01h,endc
font_70 db 04h,0eh,endv,01h,10h,30h,41h,43h,34h,14h
                              db 03h, endc
font_71 db 41h,30h,10h,01h,03h,14h,34h,43h,endv,44h
                              db 04h, endc
font_72 db 00h,04h,endv,03h,14h,34h,43h,40h,endc
font_73 db 01h,10h,30h,41h,32h,12h,03h,14h,34h
                              db 43h, endc
font_74 db 04h,44h,endv,26h,21h,30h,40h,51h,endc
font_75 db 04h,01h,10h,30h,41h,endv,44h,40h,4eh,endc
font_76 db 04h,02h,20h,42h,44h,endc
font_77 db 04h,00h,22h,40h,44h,endc
font_78 db 00h,44h,endv,04h,40h,4eh,endc
font_79 db 04h,01h,10h,30h,41h,endv,44h,4fh,3eh,1eh
                              db 0fh, endc
font_7a db 04h,44h,00h,40h,4eh,1eh
font_7b db 40h,11h,32h,03h,34h,46h,endc
font_7c db 20h,23h,endv,25h,27h,endc
font_7d db 00h,31h,12h,43h,14h,35h,06h,4eh,endc
font_7e db 06h,27h,46h,67h,endc
font_7f db 07,77, endc

scale db 0

dseg ends
end

9-48
The Read Process

Programming a read operation is simpler than programming a write operation. From the Graphics Option's point of view, only the Mode and ALU/PS registers need to be programmed. There is no need to involve the Foreground/Background Register, Text Mask, Write Buffer, or the Pattern Generator. GDC reads are programmed much like text writes except for the action command which in this case is RDAT. When reading data from the bitmap, only one plane can be active at any one time. Therefore, it can take four times as long to read back data as it did to write it in the first place.

Read the Entire Bitmap

In the following example, the entire bitmap, one plane at a time, is read and written into a 64K byte buffer in memory. This example compliments the example of displaying data from memory found in Chapter 7.
Example of Reading the Entire Bitmap

```assembly
;********************************************************************
; procedure redvid
;
; purpose: this routine will read out all of display memory, one plane at a time, then store that data in a 64k buffer in motherboard memory.
; entry:
; exit:
; register usage: ax,cx,di
;********************************************************************

dseg segment byte public 'datasg'
extrn num_planes:byte,gbmod:byte,nmredl:word,gtemp:word,curl10:word
dseg ends
vidseg segment byte public 'vseg'
    public viddata
    viddata db 0ffffh dup (?)
vidseg ends
cseg segment byte public 'codesg'
extrn gdc_not_busy:near,alups:near,fgbg:near,init_option:near
extrn mode:near
    assume cs:cseg,ds:dseg,es:dseg,ss:nothing
    public redvid

redvid proc near

;Set up to enable reads. The Graphics Option has to disable writes in the ALU/PS, enable a plane to be read in the Mode Register, and program the GDC to perform one plane’s worth of reads.
;GDC programming consists of issuing a CURSOR command of 0, a mask of FFFFh, a FIGS command with a direction to the right and a read of an entire plane, and finally the RDAT command to start the read in motion. Note that the GDC can’t read in all 8000h words of a high resolution plane but it doesn’t matter because not all 8000h words of a high resolution plane have useful information in them.
```

10-2
; clear the direction flag
; make sure the GDC is not busy
; disable all writes
; assume high resolution read
; actually high resolution?
; jump if yes
; medium resolution no. of reads
rd1: mov word ptr nmredl,ax

; Blank the screen. This will let the GDC have 100% use of the time to read the screen in.
; blank command
out 57h,al

; Set up to transfer data as it is being read from the screen into the VIDSEG data segment.
; set up the es register to point to the video buffer
; start at beginning of the buffer
; init routine sets this byte
; num_planes = 2 or 4
rd2: mov word ptr gtemp,cx ; save plane count
mov al,0bfh ; address the mode register
out 53h,al
mov ah,byte ptr gbmod ; mode byte = no graphics,
and ah,0e1h ; plane to read, write enable
or ah,al ; combine with plane to read
out 51h,al ; assert new mode
mov al,49h ; position the GDC cursor to top left
out 57h,al
xor al,al
out 56h,al
out 56h,al
mov al,4ah ; set all bits in GDC mask
out 57h,al
mov al,0ffh
out 56h,al
out 56h,al
Read Operations

mov al, 4ch ; assert the FIGS command
out 57h, al
mov al, 2 ; direction is to the right
out 56h, al
mov ax, word ptr nmredl ; number of word reads to do
out 56h, al
mov al, ah
out 56h, al
mov al, 0a0h ; start the read operation now
out 57h, al
mov cx, word ptr nmredl ; read in as they are ready.
shl cx, 1 ; bytes = 2 * words read
rd4: in al, 56h ; byte ready to be read?
test al, 1
jz rd4 ; jump if not
in al, 57h ; read the byte
stosb ; store in vid5eg
loop rd4

; We've finished reading all of the information out of that plane.
; If high resolution, increment di by a word because we were one
; word short of the entire 32k high resolution plane. Recover the
; plane to read count and loop if not done.

test byte ptr gbmod, 1 ; high resolution?
jz rd5 ; jump if not
stosw ; dummy stos to keep no. reads = words/plane
rd5: mov cx, word ptr gtemp
loop rd2 ; loop if more planes to be read

; We're done with the read.
; Restore video refresh and set the high/medium resolution flag byte
; at the end of vid5eg so that when it is written back into the video
; we do it in the proper resolution.

mov al, 0dh ; unblank the screen
out 57h, al
test byte ptr gbmod, 1 ; high res?
jnz rd6 ; jump if yes
xor al, al ; last byte = 0 for medium resolution
jmp rd7
rd6: mov al, 0ffh ; last byte = ff for high resolution
rd7: mov di, 0ffffh ; set the resolution flag
mov byte ptr es:[di], al
mov ax, dseg
mov es, ax ; restore es
ret
redvid endp

cseg ends

end
Pixel Write After a Read Operation

After a read operation has completed, the graphics option is temporarily unable to do a pixel write. (Word writes are not affected by preceding read operations.) However, the execution of a word write operation restores the option’s ability to do pixel writes. Therefore, whenever you intend to do a pixel write after a read operation, you must first execute a word write. This will ensure that subsequent vectors, arcs, and pixels will be enabled.

The following code sequence will execute a word write operation that will not write anything into the bitmap. The code assumes that the GDC is not busy since it has just completed a read operation. It also assumes that this code is entered after all the required bytes have been read out of the FIFO buffer.

```assembly
;********************************************************************
; procedure write_after_read
; purpose: Execute a no-op word write after read operation is completed.
;********************************************************************

public write_after_read
assume cs:cseg,ds:dseg,es:nothing,ss:nothing

proc near
mov al,Odh ;sometimes the GDC will not accept the first command after a read - this command can safely be missed and serves to ensure that the FIFO buffer is cleared and pointing in the right direction
xor bl,bl ;restore write enable replace mode to all planes in the ALU/PS Register
call alups ;planes in the ALU/PS Register
mov al,0ffh ;disable writes to all bits at the option's Mask Registers
out 55h,al
out 54h,al
or byte ptr gbmod,10h ;enable writes to Mode Register
call imode ;it is already in word mode
mov al,4ch ;unnecessary to assert cursor or mask since it doesn’t matter where you write - the operation will enable subsequent pixel writes
out 57h,al
xor al,al ;write is completely disabled anyway - just going through the word write
out 56h,al
out 56h,al

10-5
```
Read Operations

mov al, 22h
out 57h, al ; execute the write operation
ret

write_after_read

cseg ends
dseg segment byte
extrn gbmod: byte

dseg ends
end

10-6
Vertical Scrolling

The Scroll map controls the location of 64-word blocks of display memory on the video monitor. In medium resolution mode, this is two scan lines. In high resolution mode, this is one scan line. By redefining scan line locations in the Scroll Map, you effectively move 64 words of data into new screen locations.

All Scroll Map operations by the CPU start at location zero and increment by one with each succeeding CPU access. The CPU has no direct control over which Scroll Map location it is reading or writing. All input addresses are generated by an eight-bit index counter which is cleared to zero when the CPU first accesses the Scroll Map through the Indirect Register. There is no random access of a Scroll Map address.

Programming the Scroll Map involves a number of steps. First ensure that the GDC is not currently accessing the Scroll Map and that it won't be for some time (the beginning of a vertical retrace for example). Clearing bit 5 of the Mode Register to zero enables the Scroll Map for writing. Clearing bit 7 of the Indirect Register to zero selects the Scroll Map and clears the Scroll Map Counter to zero. Data can then be entered into the Scroll Map by writing to port 51h. When the programming operation is complete or just before the end of the vertical retrace period (whichever comes first) control of the Scroll Map addressing is returned to the GDC by setting bit 5 of the Mode Register to one.
If, for some reason, programming the Scroll Map requires more than one vertical retrace period, there is a way to break the operation up into two segments. A read of the Scroll Map increments the Scroll Map Index Counter just as though it were a write. You can therefore program the first half, wait for the next vertical retrace, read the first half and then finish the write of the last half.

Example of Vertical Scrolling One Scan Line

```asm
;********************************************************************
; procedure vscroll
; purpose: move the current entire screen up one scan line
; entry:
; exit:
; register usage: ax, cx, di, si
;********************************************************************

dseg segment byte public 'datasg'
extrn scrllb:byte, gtemp1:byte, start1:byte, gbmod:byte ;see Example 3
dseg ends
cseg segment byte public 'codesg'
extrn ascrn:near ;defined in Example 3.
assume cs:cseg,ds:dseg,es:dseg,ss:nothing
public vscroll

vscroll proc near

iThe scrollmap controls which 64 word display memory segment will be
displayed on a particular screen line. The scroll map will display
on the top high resolution scan line the 64-word segment denoted by
the data loaded into location 0. If the data is 0, the first
64-word segment is accessed. If the data is a 10, the 11th 64-word
segment is displayed. By simply rewriting the order of 64-word
segments in the scroll map, the order in which they are displayed is
;correspondingly altered. If the entire screen is to be scrolled up
one line, the entire scroll map’s contents are moved up one location.
;Data at address 1 is moved into address 0, data at address 2 is moved
;into address 1 and so on. A split screen scroll can be accomplished
;by keeping the stationary part of the screen unchanged in the scroll
;map while loading the appropriate information into the moving window.
;If more than one scroll map location is loaded with the same data,
;the corresponding scan will be displayed multiple times on the screen.
```

11-2
; Note that the information in the bitmap hasn’t been changed, only the
; location where the information is displayed on the video monitor has
; been changed. When the lines that used to be off the bottom of the
; screen scroll up and become visible, they will have in them whatever
; had been written there before. If a guaranteed clear scan line is
; desirable, the off-screen lines should be cleared with a write before
; the scroll takes place.
;
; In medium resolution, only the first 128 scroll map entries have
; meaning because while each medium resolution scan is 32 words long,
; each scroll map entry controls the location of 64 words of data. In
; medium resolution, this is the same as two entire scans. The scroll
; map acts as if the most significant bit of the scroll map entries was
; always 0. Loading an 80h into a location is the same as loading a 0.
; Loading an 81h is the equivalent to writing a 1. The example shown
; below assumes a high resolution, 256 location, scrollmap. Had it
; been medium resolution, only the first 128 scans would have been
; moved. The other 128 scroll map locations still exist but are of no
; practical use to the programmer. What this means to the applications
; programmer is that in medium resolution, after the scroll map has
; been initialized, the first 128 entries are treated as if they were
; the only scroll map locations in the table.
;
; Save the contents of the first section of the scroll table to be
; overwritten, fetch the data from however many scans away we want to
; scroll by, then move the contents of the table in a circular fashion.
; The last entry to be written is the scan we first saved. After the
; shadow scroll table has been updated, it can then be asserted by a
; call to the "ascrol" routine in the "init_option" procedure.
;
; mov si, offset scrltb ; set the source of the data
; mov di, si ; set the destination of the data
; lodsb ; fetch the first scan
; mov byte ptr gtemp1, al ; and save it
; mov cx, 255 ; move the other 255 scroll
; rep movsw ; table bytes
; mov al, byte ptr gtemp1 ; recover the first scan and put
; stosb ; it into scan 256 location
; call ascrol ; assert updated scroll table
; ret ; to scroll map

vscroll endp

cseg ends

end
Horizontal Scrolling

Not only can the video display be scrolled up and down but it can also be scrolled from side to side as well. The GDC can be programmed to start video action at an address other than location 0000. Using the PRAM command to specify the starting address of the display partition as 0002 will effectively shift the screen two words to the left. Since the screen display width is not the same as the number of words displayed on the line there is a section of memory that is unrefreshed. The data that scrolls off the screen leaves the refresh area and it will also be unrefreshed. To have the data rotate or wrap around the screen and be saved requires that data be read from the side about to go off the screen and be written to the side coming on to the screen. If the application is not rotating but simply moving old data out to make room for new information, the old image can be allowed to disappear into the unrefreshed area.

Although the specifications for the dynamic RAMs only guarantee a data persistence of two milliseconds, most of the chips will hold data much longer. Therefore, it is possible to completely rotate video memory off one side and back onto the other. However, applications considering using this characteristic should be aware of the time dependency and plan accordingly.

Example of Horizontal Scrolling One Word

```plaintext
; The GDC is programmable (on a word boundary) as to where it starts ; displaying the screen. By incrementing or decrementing that starting ; address word we can redefine the starting address of each scan line ; and thereby give the appearance of horizontal scrolling. Assume that ; this start window display address is stored in the variables: startl ; and starth. Let's further assume that we want to limit scrolling to ; one scan line's worth. Therefore, in high resolution we can never ; issue a starting address higher than 63; in medium resolution, none ; higher than 31.
```

11-4
; dseg segment byte public 'dataseg'
extrn scroll:byte,gtmp1:byte,start1:byte,gbmod:byte
dseg ends

cseg segment byte public 'codeseg'
extrn gdc_not_busy:near
assume cs:cseg,ds:dseg,es:dseg,ss:nothing
public hscroll
;
; hscroll proc near
or al, al ; move screen to left?
jz hs1 ; jump if not
dec byte ptr startl ; move screen to right
jmp hs2
hs1: inc byte ptr startl ; move screen to left
hs2: test byte ptr gbmod, 1 ; high res?
jnz hs3 ; jump if yes
and byte ptr startl, 31 ; limit to 1st medium
jmp hs4 ; resolution scan
hs3: and byte ptr startl, 63 ; limit to 1st high
; resolution scan
;
; Assert the new start1, starth to the GDC. Assume that starth is
; always going to be 0 although this is not a necessity. Issue the
; PRAM command and rewrite the starting address of the GDC display
; window 0.
;
hs4: call gdc_not_busy ; make sure the GDC is not busy
mov al, 70h ; issue the PRAM command
out 57h, al
mov al, byte ptr startl ; fetch low byte of the starting
out 56h, al ; address
xor al, al ; assume high byte is always 0
out 56h, al
ret
hscroll endp
cseg ends
end
Shadow Areas

Most of the registers in the Graphics Option control more than one function. In addition, the registers are write-only areas. In order to change selected bits in a register while retaining the settings of the rest, shadow images of these registers should be kept in motherboard memory. The current contents of the registers can be determined from the shadow area, selected bits can be set or reset by ORing or ANDing into the shadow area, and the result can be written over the existing register.

Modifying the Color Map and the Scroll Map is also made easier using a shadow area in motherboard memory. These are relatively large areas and must be loaded during the time that the screen is inactive. It is more efficient to modify a shadow area in motherboard memory and then use a fast move routine to load the shadow area into the Map during some period of screen inactivity such as a vertical retrace.

Bitmap Refresh

The Graphics Option uses the same memory accesses that fill the screen with data to also refresh the memory. This means that if the screen display stops, the dynamic video memory will lose all the data that was being displayed within two milliseconds. In high resolution, it takes two scan lines to refresh the memory (approximately 125 microseconds). In medium resolution, it takes four scan lines to refresh the memory (approximately 250 microseconds). During vertical retrace (1.6 milliseconds) and horizontal retrace (10 microseconds) there is no refreshing of the memory. Under a worst case condition, you can stop the display for no more than two milliseconds minus four medium resolution scans minus vertical retrace or just about 150 microseconds. This is particularly important when programming the Scroll Map.
All write and read operations should take place during retrace time. Failure to limit reads and writes to retrace time will result in interference with the systematic refreshing of the dynamic RAMs as well as not displaying bitmap data during the read and write time. However, the GDC is usually programmed to limit its bitmap accesses to retrace time as part of the initialization process.

**Software Reset**

Whenever you reset the GDC by issuing the RESET command (a write of zero to port 57h), the Graphics Option must also be reset (a write of any data to port 50h). This is to synchronize the memory operations of the Graphics Option with the read/modify/write operations generated by the GDC. A reset of the Graphics Option by itself does not reset the GDC; they are separate reset operations.

**Setting Up Clock Interrupts**

With the Graphics Option installed on a Rainbow system, there are two 60 hz clocks available to the programmer—one from the motherboard and one from the Graphics Option. The motherboard clock is primarily used for a number of system purposes. However, you can intercept it providing that any routine that is inserted be kept short and compatible with the interrupt handler. Refer to the “init_option” procedure in Chapter 5 for a coding example of how to insert a new interrupt vector under MS-DOS.

Clock interrupt types and vector addresses differ depending on the model of the motherboard as well as whether the interrupt is for the Graphics Option or for the motherboard. (Refer to Table 3.)

It is important to keep all interrupt handlers short! Failure to do so can cause a system reset when the motherboard’s MHFU line goes active. New interrupt handlers should restore any registers that are altered by the routine.

**Table 3. Clock Interrupt Parameters**

<table>
<thead>
<tr>
<th>MOTHERBOARD MODEL</th>
<th>INTERRUPT TYPE</th>
<th>VECTOR ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAPHICS</td>
<td>A  22h</td>
<td>88h</td>
</tr>
<tr>
<td>OPTION</td>
<td>B  A2h</td>
<td>288h</td>
</tr>
<tr>
<td>MOTHERBOARD</td>
<td>A  20h</td>
<td>80h</td>
</tr>
<tr>
<td></td>
<td>B  A0h</td>
<td>280h</td>
</tr>
</tbody>
</table>
Operational Requirements

All data modifications to the bitmap are performed by hardware that is external to the GDC. In this environment, it is a requirement that the GDC be kept in graphics mode and be programmed to write in Replace mode. Also, the internal write data patterns of the GDC must be kept as all ones for the external hardware to function correctly. The external hardware isolates the GDC from the data in the bitmap such that the GDC is not aware of multiple planes or incoming data patterns.

Although it is possible to use the GDC's internal parameter RAM for soft character fonts and graphics characters, it is faster to use the option's Write Buffer. However, to operate in the GDC's native mode, the Write Buffer and Pattern Generator should be loaded with all ones, the Mode Register should be set to graphics mode, and the Foreground/Background Register should be loaded with FOh.

When the Graphics Option is in Word Mode, the GDC's mask register should be filled with all ones. This causes the GDC to go on to the next word after each pixel operation is done. The external hardware in the meantime, has taken care of all sixteen bits on all four planes while the GDC was taking care of only one pixel.

When the option is in Vector Mode, the GDC is also in graphics mode. The GDC's mask register is now set by the third byte of the cursor positioning command (CURS). The GDC will be able to tell the option which pixel to perform the write on but the option sets the mode, data and planes.

Set-Up Mode

When you press the SET-UP key on the keyboard, the system is placed in set-up mode. This, in turn, suspends any non-interrupt driven software and brings up a set-up screen if the monitor is displaying VT102 video output. If, however, the system is displaying graphics output, the fact that the system is in set-up mode will not be apparent to a user except for the lack of any further interaction with the graphics application that has been suspended. The set-up screen will not be displayed.

Users of applications that involve graphics output should be warned of this condition and cautioned not to press the SET-UP key when in graphics output mode. Note also that pressing the SET-UP key a second time will resume the execution of the suspended graphics software.

In either case, whether the set-up screen is displayed or not, set-up mode accepts any and all keyboard data until the SET-UP key is again pressed.
Timing Considerations

It is possible for an application to modify the associated hardware that is external to the GDC (registers, buffers, maps) before the GDC has completed all pending operations. If this should occur, the pending operations would then be influenced by the new values with unwanted results.

Before changing the values in the registers, buffers, and color map, you must ensure that the GDC has completed all pending operations. The “gdc_not_busy” subroutine in the “init_option” procedure in Chapter 5 is one method of checking that the GDC has completed all pending operations.
Part III
Reference Material
PART III

Chapter 13. Option Registers, Buffers, and Maps  13-1

  I/O Ports  13-1
  Indirect Register  13-3
  Write Buffer  13-4
  Write Mask Registers  13-5
  Pattern Register  13-6
  Pattern Multiplier  13-7
  Foreground/Background Register  13-8
  ALU/PS Register  13-9
  Color Map  13-10
  Mode Register  13-11
  Scroll Map  13-12

Chapter 14. GDC Registers and Buffers  14-1

  Status Register  14-1
  FIFO Buffer  14-2

Chapter 15. GDC Commands  15-1

  Introduction  15-1
  Video Control Commands  15-2
    CCHAR – Specify Cursor and Character Characteristics  15-2
    RESET – Reset the GDC  15-3
    SYNC – Sync Format Specify  15-6
    VSYNC – Vertical Sync Mode  15-8
Contents

Display Control Commands  15-8
  BCTRL – Control Display Blanking  15-8
  CURS – Specify Cursor Position  15-9
  PITCH – Specify Horizontal Pitch  15-10
  PRAM – Load the Parameter RAM  15-10
  START – Start Display and End Idle Mode  15-12
  ZOOM – Specify the Zoom Factor  15-12

Drawing Control Commands  15-13
  FIGD – Start Figure Drawing  15-13
  FIGS – Specify Figure Drawing Parameters  15-14
  GCHRD – Start Graphics Character Draw and Area Fill  15-16
  MASK – Load the Mask Register  15-16
  WDAT – Write Data into Display Memory  15-17

DATA READ COMMANDS  15-18
  RDAT – Read Data from Display Memory  15-18
Option Registers, Buffers, and Maps

The Graphics Option uses a number of registers, buffers, and maps to generate graphic images and control the display of these images on a monochrome or color monitor. Detailed discussions of these areas may be found in Chapter 3 of this manual.

I/O Ports

The CPUs on the Rainbow system’s motherboard use the following I/O ports to communicate with the Graphics Option:

<table>
<thead>
<tr>
<th>Port</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>50h</td>
<td>Graphics option software reset and resynchronization.</td>
</tr>
<tr>
<td>51h</td>
<td>Data input to area selected through port 53h.</td>
</tr>
<tr>
<td>52h</td>
<td>Data input to the Write Buffer.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>53h</td>
<td>Area select input to Indirect Register.</td>
</tr>
<tr>
<td>54h</td>
<td>Input to low-order byte of Write Mask.</td>
</tr>
<tr>
<td>55h</td>
<td>Input to high-order byte of Write Mask.</td>
</tr>
<tr>
<td>56h</td>
<td>Parameter input to GDC - Status output from GDC.</td>
</tr>
<tr>
<td>57h</td>
<td>Command input to GDC - Data output from GDC.</td>
</tr>
</tbody>
</table>
Indirect Register

The Indirect Register is used to select one of eight areas to be written into.

Load Data: Write data byte to port 53h.

<table>
<thead>
<tr>
<th>Data Byte</th>
<th>Active Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEh</td>
<td>0</td>
<td>selects the Write Buffer</td>
</tr>
<tr>
<td>FDh</td>
<td>1</td>
<td>selects the Pattern Multiplier. (Pattern Multiplier must always be loaded before the Pattern Register)</td>
</tr>
<tr>
<td>FBh</td>
<td>2</td>
<td>selects the Pattern Register.</td>
</tr>
<tr>
<td>F7h</td>
<td>3</td>
<td>selects the Foreground/Background Register.</td>
</tr>
<tr>
<td>EFh</td>
<td>4</td>
<td>selects the ALU/PS Register.</td>
</tr>
<tr>
<td>DFh</td>
<td>5</td>
<td>selects the Color Map and resets the Color Map Address Counter to zero.</td>
</tr>
<tr>
<td>BFh</td>
<td>6</td>
<td>selects the Graphics Option Mode Register.</td>
</tr>
<tr>
<td>7Fh</td>
<td>7</td>
<td>selects the Scroll Map and resets the Scroll Map Address Counter to zero.</td>
</tr>
</tbody>
</table>

NOTE

If more than one bit is set to zero, more than one area will be selected and the results of subsequent write operations will be unpredictable.
Write Buffer

The Write Buffer is the incoming data source when the Graphics Option is in Word Mode.

Select Area: write FEh to port 53h
Clear Counter: write any value to port 51h
Load Data: write up to 16 bytes to port 52h

<table>
<thead>
<tr>
<th>BYTE</th>
<th>AS THE CPU ACCESSES IT (16 X 8-BIT RING BUFFER)</th>
<th>WORD</th>
<th>AS THE GDC ACCESSES IT (8 X 16-BIT WORDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>0,1</td>
<td>0</td>
<td>0,1</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
<td>1</td>
<td>2,3</td>
</tr>
<tr>
<td>4,5</td>
<td>4,5</td>
<td>2</td>
<td>4,5</td>
</tr>
<tr>
<td>6,7</td>
<td>6,7</td>
<td>3</td>
<td>6,7</td>
</tr>
<tr>
<td>8,9</td>
<td>8,9</td>
<td>4</td>
<td>8,9</td>
</tr>
<tr>
<td>10,11</td>
<td>10,11</td>
<td>5</td>
<td>10,11</td>
</tr>
<tr>
<td>12,13</td>
<td>12,13</td>
<td>6</td>
<td>12,13</td>
</tr>
<tr>
<td>14,15</td>
<td>14,15</td>
<td>7</td>
<td>14,15</td>
</tr>
</tbody>
</table>
Write Mask Registers

The Write Mask Registers control the writing of individual bits in a bitmap word.

Select Area: no selection required
Load Data: write low-order data byte to port 54h
write high-order data byte to port 55h

where:

- bit = 0 enables a write in the corresponding bit position of the word being displayed.
- bit = 1 disables a write in the corresponding bit position of the word being displayed.
Pattern Register

The Pattern Register provides the incoming data when the Graphics Option is in Vector Mode.

Select Area: write FBh to port 53h
Load Data: write data byte to port 51h

where:

Pattern is the pixel data to be displayed by the option when in Vector Mode.
Pattern Multiplier

The Pattern Multiplier controls the recirculating frequency of the bits in the Pattern Register.

Select Area: write FDh to port 53h
Load Data: write data byte to port 51h

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNUSED</td>
<td>VALUE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where:

- **value** is a number in the range of 0 through 15 such that 16 minus this value is the factor that determines when the Pattern Register is shifted.
Forefront/Background Register

The Foreground/Background Register controls the bit/plane input to the bitmap.

Select Area: write F7h to port 53h
Load Data: write data byte to port 51h

<table>
<thead>
<tr>
<th>DATA BYTE</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOREGROUND REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACKGROUND REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where:

**Bits**

- **0-3** are the bits written to bitmap planes 0-3 respectively when the option is in REPLACE mode and the incoming data bit is a zero.
- If the option is in OVERLAY or COMPLEMENT mode and the incoming data bit is a zero, there is no change to the bitmap value.
- **4-7** are the bits written to bitmap planes 4-7 respectively when the option is in REPLACE or OVERLAY mode and the incoming data bit is a one.
- If the option is in COMPLEMENT mode and the incoming data bit is a one, the Foreground bit determines the action. If it is a one, the bitmap value is inverted; if it is a zero, the bitmap value is unchanged.
ALU/PS Register

The ALU/PS Register controls the logic used in writing to the bitmap and the inhibiting of writing to specified planes.

Select Area: write EFh to port 53h
Load Data: write data byte to port 51h

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>enable writes to plane 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>inhibit writes to plane 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>enable writes to plane</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>inhibit writes to plane 1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>enable writes to plane 2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>inhibit writes to plane 2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>enable writes to plane 3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>inhibit writes to plane 3</td>
</tr>
<tr>
<td>5,4</td>
<td>00</td>
<td>place option in REPLACE mode</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>place option in COMPLEMENT mode</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>place option in OVERLAY mode</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Unused</td>
</tr>
<tr>
<td>7,6</td>
<td></td>
<td>Unused</td>
</tr>
</tbody>
</table>
Color Map

The Color Map translates bitmap data into the monochrome and color intensities that are applied to the video monitors.

Select Area: write DFh to port 53h (also clears the index counter)
Coordinate: wait for vertical sync interrupt
Load Data: write 32 bytes to port 51h

<table>
<thead>
<tr>
<th>2ND 16 BYTES LOADED BY THE CPU</th>
<th>1ST 16 BYTES LOADED BY THE CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONO. DATA</td>
<td>BLUE DATA</td>
</tr>
<tr>
<td>BYTE 17</td>
<td>BYTE 1</td>
</tr>
<tr>
<td>BYTE 18</td>
<td>BYTE 2</td>
</tr>
<tr>
<td>BYTE 19</td>
<td>BYTE 3</td>
</tr>
<tr>
<td>BYTE 20</td>
<td>BYTE 4</td>
</tr>
<tr>
<td>BYTE 21</td>
<td>BYTE 5</td>
</tr>
<tr>
<td>BYTE 22</td>
<td>BYTE 6</td>
</tr>
<tr>
<td>BYTE 23</td>
<td>BYTE 7</td>
</tr>
<tr>
<td>BYTE 32</td>
<td>BYTE 16</td>
</tr>
</tbody>
</table>
Mode Register

The Mode Register controls a number of the Graphics Option's operating characteristics.

Select Area: write BFh to port 53h
Load Data: write data byte to port 51h

where:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>place option in medium resolution mode</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>place option in high resolution mode</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>place option into word mode</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>place option into vector mode</td>
</tr>
<tr>
<td>3,2</td>
<td>00</td>
<td>select plane 0 for readback operation</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>select plane 1 for readback operation</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>select plane 2 for readback operation</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>select plane 3 for readback operation</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>enable readback operation</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>enable write operation</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>enable writing to the Scroll Map</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>disable writing to the Scroll Map</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>disable vertical sync interrupts to CPU</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>enable vertical sync interrupts to CPU</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>disable video output from Graphics Option</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>enable video output from Graphics Option</td>
</tr>
</tbody>
</table>

**NOTE**
The Mode Register must be reloaded following any write to port 50h (software reset).
Scroll Map

The Scroll Map controls the location of each line displayed on the monitor screen.

Preliminary: enable Scroll Map writing (Mode Register bit 5 = 0)
Select Area: write 7Fh to port 53h (also clears the index counter)
Coordinate: wait for vertical sync interrupt
Load Data: write 256 bytes to port 51h
Final: disable Scroll Map writing (Mode Register bit 5 = 1)

where:

- **GDC Line Address** is the line address as generated by the GDC and used as an index into the Scroll Map.
- **Bitmap Line Address** is the offset line address found by indexing into the Scroll Map. It becomes the new line address of data going into the bitmap.
GDC Registers and Buffers

The GDC has an 8-bit Status Register and a 16 x 9-bit first-in, first-out (FIFO) Buffer that provide the interface to the Graphics Option. The Status Register supplies information on the current activity of the GDC and the status of the FIFO Buffer. The FIFO Buffer contains GDC commands and parameters when the GDC is in write mode. It contains bitmap data when the GDC is in read mode.

Status Register

The GDC's internal status can be interrogated by doing a read from port 56h. The Status Register contents are as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Status</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DATA READY</td>
<td>When set, data is ready to be read from the FIFO.</td>
</tr>
<tr>
<td>1</td>
<td>FIFO FULL</td>
<td>When set, the command/parameter FIFO is full.</td>
</tr>
<tr>
<td>2</td>
<td>FIFO EMPTY</td>
<td>When set, the command/parameter FIFO is completely empty.</td>
</tr>
<tr>
<td>3</td>
<td>DRAWING IN PROGRESS</td>
<td>When set, the GDC is performing a drawing function. Note, however, that this bit can be cleared before the DRAW command is fully completed. The GDC does not draw continuously and this bit is reset during interrupts to the write operation.</td>
</tr>
<tr>
<td>4</td>
<td>DMA EXECUTE</td>
<td>Not used.</td>
</tr>
<tr>
<td>5</td>
<td>VERTICAL SYNC ACTIVE</td>
<td>When set, the GDC is doing a vertical sync.</td>
</tr>
<tr>
<td>6</td>
<td>HORIZONTAL SYNC ACTIVE</td>
<td>When set, the GDC is doing a horizontal sync.</td>
</tr>
<tr>
<td>7</td>
<td>LIGHT PEN DETECTED</td>
<td>Not used.</td>
</tr>
</tbody>
</table>
**FIFO Buffer**

You can both read from and write to the FIFO Buffer. The direction that the data takes through the buffer is controlled by the Rainbow system using GDC commands. GDC commands and their associated parameters are written to ports 57h and 56h respectively. The GDC stores both in the FIFO Buffer where they are picked up by the GDC command processor. The GDC uses the ninth bit in the FIFO Buffer as a flag bit to allow the command processor to distinguish between commands and parameters. Contents of the bitmap are read from the FIFO using reads from port 57h.

![Diagram of FIFO Buffer](image)

where:

- `flg` is a flag bit to be interpreted as:
  - 0 - data byte is a parameter
  - 1 - data byte is a command

- `data byte` is a GDC command or parameter

When you reverse the direction of flow in the FIFO Buffer, any pending data in the FIFO is lost. If a read operation is in progress and a command is written to port 56h, the unread data still in the FIFO is lost. If a write operation is in progress and a read command is processed, any unprocessed commands and parameters in the FIFO Buffer are lost.
Introduction

This chapter contains detailed reference information on the GDC commands and parameters supported by the Graphics Option. The commands are listed in alphabetical order within functional category as follows:

- **Video Control Commands**
  - CCHAR
    - Specifies the cursor and character row heights
  - RESET
    - Resets the GDC to its idle state
  - SYNC
    - Specifies the video display format
  - VSYNC
    - Selects Master/Slave video synchronization mode

- **Display Control Commands**
  - BCTRL
    - Controls the blanking/unblanking of the display
  - CURS
    - Sets the position of the cursor in display memory
  - PITCH
    - Specifies the width of display memory
  - PRAM
    - Defines the display area parameters
  - START
    - Ends idle mode and unblanks the display
  - ZOOM
    - Specifies zoom factor for the graphics display
GDC Commands

• Drawing Control Commands
  
  FIGD – Draws the figure as specified by FIGS command
  FIGS – Specifies the drawing controller parameters
  GCHRD – Draws the graphics character into display memory
  MASK – Sets the mask register contents
  WDAT – Writes data words or bytes into display memory

• Data Read Commands
  
  RDAT – Reads data words or bytes from display memory
Video Control Commands

CCHAR – Specify Cursor and Character Characteristics

Use the CCHAR command to specify the cursor and character row heights and characteristics.

**COMMAND BYTE**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**PARAMETER BYTES**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>DC</td>
<td>0</td>
<td>0</td>
<td>LR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>BR(LO)</td>
<td>SC</td>
<td>CTOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>CBOT</td>
<td>BR(HI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where:

- **DC** controls the display of the cursor
  - 0 – do not display cursor
  - 1 – display the cursor
- **LR** is the number of lines per character row, minus 1
- **BR** is the blink rate (5 bits)
- **SC** controls the action of the cursor
  - 0 – blinking cursor
  - 1 – steady cursor
- **CTOP** is the cursor’s top line number in the row
- **CBOT** is the cursor’s bottom line number in the row
  (CBOT must be less than LR)
RESET – Reset the GDC

Use the RESET command to reset the GDC. This command blanks the display, places the GDC in idle mode, and initializes the FIFO buffer, command processor, and the internal counters. If parameter bytes are present, they are loaded into the sync generator.

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETER BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>P4</td>
</tr>
<tr>
<td>P5</td>
</tr>
<tr>
<td>P6</td>
</tr>
<tr>
<td>P7</td>
</tr>
<tr>
<td>P8</td>
</tr>
</tbody>
</table>
where:

CG sets the display mode for the GDC
00 – mixed graphics and character mode
01 – graphics mode only
10 – character mode only
11 – invalid

IS controls the video framing for the GDC
00 – noninterlaced
01 – invalid
10 – interlaced repeat field for character displays
11 – interlaced

D controls the RAM refresh cycles
0 – no refresh – static RAM
1 – refresh – dynamic RAM

F controls the drawing time window
0 – drawing during active display time and retrace blanking
1 – drawing only during retrace blanking

AW active display words per line minus 2; must be an even number

HS horizontal sync width minus 1

VS vertical sync width

HFP horizontal front porch width minus 1

HBP horizontal back porch width minus 1

VFP vertical front porch width

AL active display lines per video field

VBP vertical back porch width
SYNC – Sync Format Specify

Use the SYNC command to load parameters into the sync generator. The GDC is neither reset nor placed in idle mode.

![Command Byte Diagram]

where:

- **DE** controls the display
  - 0 – disables (blanks) the display
  - 1 – enables the display

![Parameter Bytes Diagram]
where:

CG sets the display mode for the GDC
  00 – mixed graphics and character mode
  01 – graphics mode only
  10 – character mode only
  11 – invalid

IS controls the video framing for the GDC
  00 – noninterlaced
  01 – invalid
  10 – interlaced repeat field for character displays
  11 – interlaced

D controls the RAM refresh cycles
  0 – no refresh – static RAM
  1 – refresh – dynamic RAM

F controls the drawing time window
  0 – drawing during active display time and retrace blanking
  1 – drawing only during retrace blanking

AW active display words per line minus 2; must be an even number

HS horizontal sync width minus 1

VS vertical sync width

HFP horizontal front porch width minus 1

HBP horizontal back porch width minus 1

VFP vertical front porch width

AL active display lines per video field

VBP vertical back porch width
VSYNC – Vertical Sync Mode

Use the VSYNC command to control the slave/master relationship whenever multiple GDC’s are used to contribute to a single image.

**COMMAND BYTE**

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>
```

where:

- **M** sets the synchronization status of the GDC
  - 0 – slave mode (accept external vertical sync pulses)
  - 1 – master mode (generate and output vertical sync pulses)
Display Control Commands

BCTRL – Control Display Blanking

Use the BCTRL command to specify whether the display is blanked or enabled.

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 0 0 0 1 1 0 DE</td>
</tr>
</tbody>
</table>

where:

DE controls the display
- 0 – disables (blanks) the display
- 1 – enables the display
CURS – Specify Cursor Position

Use the CURS command to set the position of the cursor in display memory. In character mode the cursor is displayed for the length of the word. In graphics mode the word address specifies the word that contains the starting pixel of the drawing; the dot address specifies the pixel within that word.

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 0 0 1 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETER BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>P1 EAD(LO)</td>
</tr>
<tr>
<td>P2 EAD(MID)</td>
</tr>
<tr>
<td>P3 dAD 0 0 EAD(HI)</td>
</tr>
</tbody>
</table>

where:

- EAD is the execute word address (18 bits)
- dAD is the dot address within the word
PITCH – Specify Horizontal Pitch

Use the PITCH command to set the width of the display memory. The drawing processor uses this value to locate the word directly above or below the current word. It is also used during display to find the start of the next line.

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 0 0 0 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETER BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>P1 P</td>
</tr>
</tbody>
</table>

where:

P is the number of word addresses in display memory in the horizontal direction
**PRAM – Load the Parameter RAM**

Use the PRAM command to load up to 16 bytes of information into the parameter RAM at specified adjacent locations. There is no count of the number of parameter bytes to be loaded; the sensing of the next command byte stops the load operation. Because the Graphics Option requires that the GDC be kept in graphics mode, only parameter bytes one through four, nine, and ten are used.

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 1 1 SA</td>
</tr>
</tbody>
</table>

where:

- **SA** is the start address for the load operation (Pn - 1)
PARAMETER BYTES

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>SAD(LO)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>SAD(MID)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>LEN(LO)</td>
<td>0</td>
<td>0</td>
<td>SAD(HI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>WD</td>
<td>IM</td>
<td>LEN(HI)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>UNUSED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>UNUSED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P11</td>
<td>UNUSED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P16</td>
<td>UNUSED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where:

SAD is the start address of the display area (18 bits)
LEN is the number of lines in the display area (10 bits)
GDC Commands

WD  sets the display width
    0 – one word per memory cycle (16 bits)
    1 – two words per memory cycle (8 bits)

IM  sets the current type of display when the GDC is in mixed graphics and character mode
    0 – character area
    1 – image or graphics area

NOTE
When the GDC is in graphics mode, the IM bit must be a zero.
START – Start Display and End Idle Mode

Use the START command to end idle mode and enable the video display.

```
COMMAND BYTE
  7 6 5 4 3 2 1 0
  0 1 1 0 1 0 1 1
```
GDC Commands

ZOOM – Specify the Zoom Factor

Use the ZOOM command to set up a magnification factor of 1 through 16 (using codes 0 through 15) for the display and for graphics character writing.

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETER BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 DISP GCHR</td>
</tr>
</tbody>
</table>

where:

DISP is the zoom factor (minus one) for the display
GCHR is the zoom factor (minus one) for graphics character writing and area fills
Drawing Control Commands

FIGD – Start Figure Drawing

Use the FIGD command to start drawing the figure specified with the FIGS command. This command causes the GDC to:

- load the parameters from the parameter RAM into the drawing controller, and
- start the drawing process at the pixel pointed to by the cursor: Execute Word Address (EAD) and Dot Address within the word (dAD)

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 1 0 1 1 0 0</td>
</tr>
</tbody>
</table>

LJ-0250
FIGS – Specify Figure Drawing Parameters

Use the FIGS command to supply the drawing controller with the necessary figure type, direction, and drawing parameters needed to draw figures into display memory.

**COMMAND BYTE**

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**PARAMETER BYTES**

```
P1
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL</td>
<td>R</td>
<td>A</td>
<td>GC</td>
<td>L</td>
<td>DIR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P2
| DC(LO) |

P3
| 0 | GD |

P4
| DC(HI) |

P5
| D(LO) |

P6
| D(LO) |

P7
| D(HI) |

P8
| D2(HI) |

P9
| D1(HI) |

P10
| DM(LO) |

P11
| DM(HI) |
```
where:

SL  Slanted Graphics Character
R   Rectangle
A   Arc/Circle
GC  Graphics Character
L   Line (Vector)

DIR is the drawing direction base (see definitions below)
DC is the DC drawing parameter (14 bits)
GD is the graphic drawing flag used in mixed graphics and character mode
D  is the D drawing parameter (14 bits)
D2 is the D2 drawing parameter (14 bits)
D1 is the D1 drawing parameter (14 bits)
DM is the DM drawing parameter (14 bits)

FIGURE TYPE SELECT BITS [VALID COMBINATIONS]

<table>
<thead>
<tr>
<th>SL R A GC L</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>CHARACTER DISPLAY MODE DRAWING, INDIVIDUAL DOT DRAWING, WDAT, AND RDAT</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>STRAIGHT LINE DRAWING</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>GRAPHICS CHARACTER DRAWING AND AREA FILL WITH GRAPHICS CHARACTER PATTERN</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>ARC AND CIRCLE DRAWING</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>RECTANGLE DRAWING</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
<td>SLANTED GRAPHICS CHARACTER DRAWING AND SLANTED AREA FILL</td>
</tr>
</tbody>
</table>
DRAWING DIRECTION BASE (DIR)

[101]  [100]  [011]

[110]→[START]→[010]

[111]  [000]  [001]
GCHRD – Start Graphics Character Draw and Area Fill

Use the GCHRD command to initiate the drawing of the graphics character or area fill pattern that is stored in the Parameter RAM. The drawing is further controlled by the parameters loaded by the FIGS command. Drawing begins at the address in display memory pointed to by the Execute Address (EAD) and Dot Address (dAD) values.

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 1 0 1 0 0 0</td>
</tr>
</tbody>
</table>

**GDC Commands**
MASK – Load the Mask Register

Use the MASK command to set the value of the 16-bit Mask Register that controls which bits of a word can be modified during a Read/Modify/Write (RMW) cycle.

where:

- \( M \) is the bit configuration to be loaded into the Mask Register (16 bits). Each bit in the Mask Register controls the writing of the corresponding bit in the word being processed as follows:
  - 0 – disable writing
  - 1 – enable writing
WDAT – Write Data Into Display Memory

Use the WDAT command to perform RMW cycles into display memory starting at the location pointed to by the cursor Execute Word Address (EAD). Precede this command with a FIGS command to supply the writing direction (DIR) and the number of transfers (DC).

Command Byte

<table>
<thead>
<tr>
<th>COMMAND BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 0 1 TYPE 0 MOD</td>
</tr>
</tbody>
</table>

where:

TYPE is the type of transfer
00 – word transfer (first low then high byte) 
01 – invalid 
10 – byte transfer (low byte of the word only) 
11 – byte transfer (high byte of the word only) 

MOD is the RMW memory logical operation
00 – REPLACE with Pattern 
01 – COMPLEMENT 
10 – RESET to Zero 
11 – SET to One

Parameter Bytes

<table>
<thead>
<tr>
<th>PARAMETER BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
</tbody>
</table>

where:

WORD is a 16-bit data value 
BYTE is an 8-bit data value
Data Read Commands

RDAT – Read Data From Display Memory

Use the RDAT command to read data from display memory and pass it through the FIFO buffer and microprocessor interface to the host system. Use the CURS command to set the starting address and the FIGS command to supply the direction (DIR) and the number of transfers (DC). The type of transfer is coded in the command itself.

**COMMAND BYTE**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>TYPE</td>
<td>0</td>
<td>MOD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where:

**TYPE** is the type of transfer
- 00 – word transfer (first low then high byte)
- 01 – invalid
- 10 – byte transfer (low byte of the word only)
- 11 – byte transfer (high byte of the word only)

**MOD** is the RMW memory logical operation
- 00 – REPLACE with Pattern
- 01 – COMPLEMENT
- 10 – RESET to Zero
- 11 – SET to One

**NOTE**
The MOD field should be set to 00 if no modification to the video buffer is desired.
Part IV
Appendixes
Contents

PART IV

Appendix A. Option Specification Summary  A-1
  Physical Specifications  A-1
  Environmental Specifications  A-1
    Temperature  A-1
    Humidity  A-1
    Altitude  A-2
  Power Requirements  A-2
  Standards and Regulations  A-2
  Part and Kit Numbers  A-3

Appendix B. Rainbow Graphics Option — Block Diagram  B-1

Appendix C. Getting Help  C-1
Option Specification Summary

Physical Specifications

The Graphics Option Video Subsystem is a 5.7” × 10.0”, high density, four-layer PCB with one 40-pin female connector located on side 1. This connector plugs into a shrouded male connector located on the system module. The option module is also supported by two standoffs.

Environmental Specifications

Temperature

- Operating ambient temperature range is 10 to 40 degrees C.
- Storage temperature is –40 to 70 degrees C.

Humidity

- 10% to 90% non-condensing
- Maximum wet bulb, 28 degrees C.
- Minimum dew point, 2 degrees C.
Altitude

- Derate maximum operating temperature 1 degree per 1,000 feet elevation
- Operating limit: 22.2 in. Hg. (8,000 ft.)
- Storage limit: 8.9 in Hg. (30,000 ft.)

Power Requirements

<table>
<thead>
<tr>
<th></th>
<th>Calculated Typical</th>
<th>Calculated Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5V DC (+/-5%)</td>
<td>3.05 amps</td>
<td>3.36 amps</td>
</tr>
<tr>
<td>+12V DC (+/-10%)</td>
<td>180 mA</td>
<td>220 mA</td>
</tr>
</tbody>
</table>

Standards and Regulations

The Graphics Option module complies with the following standards and recommendations:

- DEC Standard 119 – Digital Product Safety (covers UL 478, UL 114, CSA 22.2 No. 154, VDE 0806, and IEC 380)
- IEC 485 – Safety of Data Processing Equipment
- EIA RS170 – Electrical Performance Standards – Monochrome Television Studio Facilities
- CCITT Recommendation V.24 – List of Definitions for Interchange Circuit Between Data Terminal Equipment and Data Circuit Terminating Equipment
- CCITT Recommendation V.28 – Electrical Characteristics for Unbalanced Double-Current Interchange Circuits
Option Specification Summary

Part and Kit Numbers

Graphics Option

Hardware:
- Printed Circuit Board
- Color RGB Cable

Software and Documentation:
- Rainbow Color/Graphics Option Installation Guide
- Rainbow GSX-86 Getting Started
- Rainbow Diagnostic/GSX-86 Diskette
- Rainbow 100 CP/M-86/80 V1.0 Technical Documentation
- Rainbow 100 MS-DOS V2.01 Technical Documentation

PC1XX-BA
54-15688
BCC17-06
EK-PCCOL-IN-001
AA-AE36A-TV
AA-V526A-TV
AA-W964A-TV
BL-W965A-RV
QV043-GZ
QV025-GZ
Figure 15. Rainbow Graphics Option — Block Diagram
## Help Line Phone Numbers

<table>
<thead>
<tr>
<th>Country</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.S.A.</td>
<td>(800) DEC-8000</td>
</tr>
<tr>
<td>Canada</td>
<td>(800) 267-5251</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>(0256) 59 200</td>
</tr>
<tr>
<td>Belgium</td>
<td>(02)-24 26 790</td>
</tr>
<tr>
<td>West Germany</td>
<td>(089) 95 91 66 44</td>
</tr>
<tr>
<td>Italy</td>
<td>(02)-617 53 81 or 617 53 82</td>
</tr>
<tr>
<td>Japan</td>
<td>(0424) 64-3302</td>
</tr>
<tr>
<td>Denmark</td>
<td>(04)-30 10 05</td>
</tr>
<tr>
<td>Spain</td>
<td>(1)-73 34 307</td>
</tr>
<tr>
<td>Finland</td>
<td>(90)-42 33 32</td>
</tr>
<tr>
<td>Holland</td>
<td>(1820)-31 100</td>
</tr>
<tr>
<td>Switzerland</td>
<td>(01)-810 51 21</td>
</tr>
<tr>
<td>Sweden</td>
<td>(08)-98 88 35</td>
</tr>
<tr>
<td>Norway</td>
<td>(02)-25 64 22</td>
</tr>
<tr>
<td>France</td>
<td>(1)-687 31 52</td>
</tr>
<tr>
<td>Austria</td>
<td>(222)-67 76 41 extension 444</td>
</tr>
<tr>
<td>Australia</td>
<td></td>
</tr>
<tr>
<td>Sydney</td>
<td>(02) 412-5555</td>
</tr>
<tr>
<td>All other areas</td>
<td>(008) 226377</td>
</tr>
</tbody>
</table>
A
Address conversion
  from pixel coordinates 3–5
Address logic 3–2
Altitude specifications 1–2
ALU functions
  COMPLEMENT 4–8, 4–18
  OVERLAY 4–9, 4–19
  REPLACE 4–8, 4–18
ALU/PS Register 4–8, 6–1
  bit definitions 13–9
  load data 13–9
  select 13–9
Arithmetic Logic Unit 4–8

B
Background Register 4–6
BCTRL command 15–9
Bit definitions
  ALU/PS Register 13–9
  BCTRL command 15–9
  CCHAR command 15–3
  CURS command 15–10
  FIFO Buffer 14–2
  FIGS command 15–18
  Foreground/Background Register 13–8
  GDC Status Register 14–1
  Indirect addressing 4–2
  Indirect Register 13–3
  MASK command 15–22
  Mode Register 13–11
  PITCH command 15–11
  PRAM command 15–12
  RDAT command 15–24
  RESET command 15–4
  Status Register 14–1
  SYNC command 15–6
  VSYNC command 15–8
  WDAT command 15–23
  Write Mask Registers 13–5
  ZOOM command 15–16
Bitmap 1–2
  data 3–6
  line address 13–12
  modifications 12–3
  organization 3–5
  reading from 10–1
  refreshing 12–1
Bitmap planes
  high resolution 3–6
  medium resolution 3–6
C
CCHAR command 15-3
   initial value 5-8
Character
   characteristics 15-3
Characteristics of
   character 15-3
cursor 15-3
Circle
   display a 8-9
Clear index counter
   Color Map 13-10
   Scroll Map 13-12
   Write Buffer 13-4
Clock interrupt
   parameters 12-2
types 12-2
   vector addresses 12-2
Clock interrupts 12-2
Clocks
   Graphics Option 12-2
   motherboard 12-2
Color intensities 4-9
   available 1-1
   conversion to drive voltages 4-13
displayed 1-1
Color Map 3-6, 4-9
   high resolution 4-11
   load data 13-10
   loading 4-12
   medium resolution 4-10
   select 13-10
Color monitor 2-3
Components
   hardware 1-1
Configuration
   Color Map 4-9
Configurations
   color monitor 2-3
dual monitors 2-4
   monochrome monitor 2-2
Control display blanking 15-9
Control graphics output 5-24
Control multiple GDCs 15-8
Conversion
   color intensities to drive voltages 4-13

Conversion table
   color intensities to drive voltages 4-13
CURS command 15-10
Cursor
   characteristics 15-3
   positioning 15-10

D
Data flow in FIFO Buffer 14-2
Data logic 3-2
Data path
   color monitor 2-3
dual monitors 2-4
   monochrome monitor 2-2
Data patterns 3-2
Data read commands 15-2
Digital-to-analog converters 4-13
Disable
   individual bits 4-4
   plane writes 4-8
Display
   a circle 8-9
   a pixel 8-4
   a vector 8-5
Display blanking 15-9
Display control commands 15-1
Display logic 3-6
Display memory 1-2, 3-2
   GDC access to 3-3
   organization 3-5
Display planes 1-2
Displaying data from memory 7-1
Drawing control commands 15-2
Dual monitors 2-4

E
Enable
   individual bits 4-4
   plane writes 4-8
End idle mode 15-15
Environmental specifications 1-1
Examples
   CCP/M version test 5-5
   CP/M version test 5-2
   disable monitor output 5-25
display a circle 8–9
display a pixel 8–4
display a vector 8–6
display data from memory 7–1
enable monitor output 5–24
horizontal scrolling 11–4
initialize Graphics Option 5–9
load Color Map 5–26
loading ALU/PS Register 6–1
loading Foreground/Background Register 6–2
loading Pattern Multiplier 8–3
loading Pattern Register 8–1
modify color data 5–26
MS-DOS version test 5–3
no-op word write 10–5
option present test 5–1
read entire bitmap 10–2
set area to a color 7–4
vertical scrolling 11–2
write a text string 9–38
writing byte-aligned character 9–1

F
FIFO Buffer 3–9, 14–2
bit definitions 14–2
data flow 14–2
flag bit 3–9
read mode 3–9
write mode 3–9
FIGD command 15–17
FIGS command 15–18
Figure drawing parameters 15–18
Foreground Register 4–6
Foreground/Background Register 4–6, 6–2
bit definitions 13–8
load data 13–8
select 13–8
Full-screen scrolling 4–16

G
GCHRD command 15–21
GDC 1–1
command processor 14–2
in native mode 12–3
initialize 5–7
GDC access to bitmap 3–7
GDC addresses 3–5
GDC buffers
reference data 14–1
GDC command bytes 3–9
GDC command logic 3–9
GDC commands 15–1
BCTRL 15–9
CCHAR 15–3
CURS 15–10
FIGD 15–17
FIGS 15–18
GCHRD 15–21
in FIFO Buffer 14–2
MASK 15–22
PITCH 15–11
PRAM 15–12
RDAT 15–24
RESET 12–2, 15–4
START 15–15
SYNC 15–6
VSYNC 15–8
WDAT 15–23
ZOOM 15–16
GDC functions 1–2
GDC line address 13–12
GDC Mask Register 15–22
GDC parameter bytes 3–9
GDC parameters
in FIFO Buffer 14–2
GDC registers
reference data 14–1
GDC reset 5–6, 12–2
parameters 5–6
GDC Status Register
bit definitions 14–1
Graphics Display Controller 1–1
Index

Graphics Option 1–1
   I/O ports 13–1
      in vector mode 12–3
      in word mode 12–3
   initialize 5–8
   regulations 1–2
   reset 12–2
   standards 1–2
Graphics option
   reference data 13–1
Graphics output
   control of 5–24

H
Hardware components 1–1
High resolution 1–3
   refresh 12–1
   Horizontal Back Porch 3–7
   Horizontal Front Porch 3–7
   Horizontal pitch 15–11
   Horizontal retrace 3–7
   Horizontal scrolling 11–4
   Humidity specifications 1–1

I
I/O ports 4–1, 13–1
   Index counter
      Write Buffer 4–2
   Indirect addressing 4–2
      bit definitions 4–2
   Indirect Register 4–2
      bit definitions 13–3
      load data 13–3
   Initial values
      CCHAR command 5–8
      PITCH command 5–8
      PRAM command 5–8
      ZOOM command 5–7
   Initialize
      GDC 5–7
      Graphics Option 5–8
   Intensity values
      conversion to drive voltages 4–13
   Interrupt control 4–15, 4–19

L
Line address
   bitmap 13–12
   GDC 13–12
Load
   ALU/PS Register 6–1
   Foreground/Background Register 6–2
   Pattern Multiplier 8–3
   Pattern Register 8–1
Load data
   ALU/PS Register 13–9
   Color Map 13–10
   Foreground/Background Register 13–8
   Indirect Register 13–3
   Mode Register 13–11
   Pattern Multiplier 13–7
   Pattern Register 13–6
   Scroll Map 13–12
   Write Buffer 13–4
   Write Mask Registers 13–5
Load GDC Mask Register 15–22
Load parameter RAM 15–12
Loading
   Color Map 4–12, 5–25
   Scroll Map 4–17
   Write Buffer 4–3
   Write Mask Registers 4–4

M
Magnification factor 15–16
MASK command 15–22
Medium resolution 1–3
   refresh 12–1
Mode
   readback 1–3
   scroll 1–3
   vector 1–3, 3–2
   word 1–3, 3–2
Mode Register 4–15, 4–19
   bit definitions 13–11
   load data 13–11
   select 13–11
Model A motherboard 1–1
Model B motherboard 1–1
Modify color data 5–26
Monitor configurations 2–1
Monochrome monitor 2–2
Motherboard
  Model A 1–1
  Model B 1–1
Multiple GDCs 15–8

O
Operating mode 4–15, 4–19
Operational requirements 12–3
Option
  components 4–1
  kit numbers 1–3
  part numbers 1–3
Option specifications
  altitude 1–2
  environmental 1–1
  humidity 1–1
  physical 1–1
  power requirements 1–2
  temperature 1–1
Organization
  bitmap 3–5
Overview 1–1

P
Parameter RAM 15–12
Parameters
  clock interrupt 12–2
Pattern Generator 4–5, 8–1
  schematic 4–5
  shift frequency 4–6
Pattern Multiplier 4–5
  load data 13–7
  loading 8–3
  select 13–7
Pattern Register 4–5, 8–1
  load data 13–6
  loading 8–1
  select 13–6
Persistence
  of screen data 11–4
Physical specifications 1–1
PITCH command 15–11
  initial value 5–8
Pixel
  address 3–5
  display a 8–4
Plane select function 4–8
Power requirement specifications 1–2
PRAM command 15–12
  initial value 5–8
Programming the Scroll Map 11–1

R
RDAT command 15–24
Read from display memory 15–24
Read operation 10–1
Readback mode 1–3, 4–15, 4–19
Reading
  entire bitmap 10–1
  precaution 10–5
Reference data
  GDC buffers 14–1
  GDC registers 14–1
  graphics option buffers 13–1
  graphics option maps 13–1
  graphics option registers 13–1
Refreshing
  bitmap 12–1
  in high resolution 12–1
  in medium resolution 12–1
Registers
  ALU/PS 4–8
  Foreground/Background 4–6
  Indirect 4–2
  Mode 4–15, 4–19
  Pattern 4–5
  Write Mask 4–4
Requirements
  operational 12–3
Reset
  GDC 12–2
  Graphics Option 12–2
RESET command 12–2, 15–4
Reset GDC 5–6
Reset the GDC 15–4
Resolution
  high 1–3
  medium 1–3
Resolution mode 4–15, 4–19
Index

S
Scan line
   definition 3-5
Screen control parameters 3-7
Screen data persistence 11-4
Screen logic 3-7
Scroll Map 3-5, 4-16
   load data 13-12
   loading 4-17
   operations 11-1
   programming 11-1
   select 13-12
   shadow image 4-17
Scroll Map control 4-15, 4-19
Scroll mode 1-3
Scrolling
   horizontal 11-4
   vertical 11-1
Select
   ALU/PS Register 13-9
   Color Map 13-10
   Foreground/Background Register 13-8
   Mode Register 13-11
   Pattern Multiplier 13-7
   Pattern Register 13-6
   Scroll Map 13-12
   Write Buffer 13-4
   Write Mask Registers 13-5
Set area to a color 7-4
SET-UP key 12-3
Set-up mode 12-3
Shadow areas 12-1
Shadow color map 5-26
Shadow image
   Scroll Map 4-17
Shadowing
   Color Map 12-1
   Scroll Map 12-1
Software logic 3-1
Split-screen scrolling 4-16
START command 15-15
Start display 15-15
Start figure drawing 15-17
Start graphics area fill 15-21
Start graphics character draw 15-21
Status Register
   bit definitions 14-1
SYNC command 5-8, 15-8
Sync format 15-6
System in set-up mode 12-3
System maintenance port 2-1
T
Temperature specifications 1-1
Test for motherboard version 5-2
Test for option present 5-1
Timing considerations 12-5
V
Vector
   display a 8-5
   Vector mode 1-3, 3-2
Vertical
   retrace 3-7
   scrolling 4-16, 11-1
   Vertical Back Porch 3-7
   Vertical Front Porch 3-7
   Video control commands 15-1
   Video display
      organization 3-2
   Video drive voltages 4-13
   Video output control 4-15, 4-20
   VSYNC command 5-8, 15-8
W
WDAT command 5-7, 15-23
Word address 3-5
Word mode 1-3, 3-2
Write Buffer 4-2
   clear index counter 13-4
   index counter 4-2
   load data 13-4
   loading 4-3
   output 4-3
   select 13-4
Write byte-aligned character 9-1
Write Mask Registers 3-5, 4-4
   bit definitions 13-5
   load data 13-5
   loading 4-4
   select 13-5
Write mode 4–15, 4–19
Write operations 3–1
Write text string 9–38
Write to display memory 15–23
Writing depth 3–1
Writing length 3–1
Writing time 3–1
Writing width 3–1

Z

ZOOM command 15–16
  initial value 5–7
Zoom factor 15–16
HOW TO ORDER ADDITIONAL DOCUMENTATION

If you want to order additional documentation by phone:

And you live in: Call: Between the hours of:
New Hampshire, Alaska or Hawaii 603-884-6660 8:30 AM and 6:00 PM Eastern Time
Continental USA or Puerto Rico 1-800-258-1710 8:30 AM and 6:00PM Eastern Time
Canada (Ottawa-Hull) 613-234-7726 8:00 AM and 5:00 PM Eastern Time
Canada (British Columbia) 1-800-267-6146 8:00 AM and 5:00 PM Eastern Time
Canada (all other) 112-800-267-6146 8:00 AM and 5:00 PM Eastern Time

If you want to order additional documentation by direct mail:

And you live in: Write to:
USA or Puerto Rico DIGITAL EQUIPMENT CORPORATION
ATTN: Peripherals and Supplies Group
P.O. Box CS2008
Nashua, NH 03061
NOTE: Prepaid orders from Puerto Rico must be placed with the local DIGITAL subsidiary
(Phone 809-754-7575)
Canada DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: P&SG Business Manager
Other than USA, Puerto Rico or Canada DIGITAL EQUIPMENT CORPORATION
Peripherals and Supplies Group
P&SG Business Manager
c/o Digital's local subsidiary or approved distributor

TO ORDER MANUALS WITH EK PART NUMBERS WRITE OR CALL

P&CS PUBLICATIONS
Circulation Services
10 Forbes Road
NR03/W3
Northboro, Massachusetts 01532
(617)351-4325
READER'S COMMENTS

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

Did you find errors in this manual? If so, specify the error and the page number.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

Please indicate the type of reader that you most nearly represent.

☐ First-time computer user
☐ Experienced computer user
☐ Application package user
☐ Programmer
☐ Other (please specify) ________________________________________________

Name ________________________________________________

Date ________________________________________________

Organization _________________________________________

Street _______________________________________________

City _________________________________________________

State ________________________________________________

Zip Code _____________________________________________
or Country ___________________________________________