Programming with DOMAIN Graphics Primitives

Order No. 005808
Revision 00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: July, 1985
Updated: January, 1987

This document was produced using the Interleaf Workstation Publishing Software (WPS). Interleaf and WPS are trademarks of Interleaf, Inc.

UPDATING INSTRUCTIONS

The information in this package supersedes the contents of *Programming with DOMAIN Graphics Primitives*, Order Number 005808, Revision 00. To update your manual, remove and insert the new sheets as listed below. Insert this sheet behind the title page as a record of the changes.

NOTE: **The table of contents and the index will be updated when the manual (005808) is revised**

REMOVE	INSERT
Title/Disclaimer	Title/Disclaimer
	Appendix E (tab divider)
	Appendix E
	Appendix F (tab divider)
	Appendix F

Preface

Programming With DOMAIN Graphics Primitives describes the DOMAIN® graphics primitive system. This manual shows how to use graphics primitive routines in application programs. For a detailed description of these routines see the *DOMAIN System Call Refrence (Volume I)*.

Audience

This manual is for programmers who use the DOMAIN Graphics Primitives to develop application programs. It is assumed that users of this manual have some knowledge of computer graphics and have experience using the DOMAIN system.

All the programming examples used in this manual are presented in Pascal with translations into FORTRAN and C in the appendices. In addition, all the programming examples are also on-line. You can retrieve an example on-line by

Organization

This manual contains nine chapters and four appendices.

Chapter 1	Presents an overview of the graphics primitives package and a comparison with other DOMAIN graphics packages.
Chapter 2	Describes display configurations, formats, and the modes within which the graphics routines can operate.
Chapter 3	Describes the essentials of writing GPR™ application programs.
Chapter 4	Describes how to use GPR drawing and text routines.
Chapter 5	Describes cursor control and input operations.
Chapter 6	Describes the various types of bitmaps and the attribute blocks associated with them.
Chapter 7	Discusses bitmaps outside display memory. It demonstrates how to use bit-block transfers to copy information from one bitmap to another or from one location to another location in the same bitmap.
Chapter 8	Describes color configurations, formats, color maps, and the operation modes for color graphics.
Chapter 9	Discusses Graphics Map Files.
Appendix A	Presents a glossary of graphics terms in relation to the Graphics Primitives package.

Appendix B Illustrates the 880 and low-profile keyboard and keyboard charts.

Appendix C Presents the Pascal program examples used in the manual translated into C.

Appendix D Presents the Pascal program examples used in the manual translated into FORTRAN.

Additional Reading

For information about using DOMAIN Graphics Metafiles, see *Programming With DOMAIN 2D Graphics Metafiles Resource.* For information about using the DOMAIN system, see the *DOMAIN System Command Reference Manual* and the *DOMAIN System User's Guide.* For information about the software components of the operating system and user-callable system routines, see the *DOMAIN System Call Reference (Volumes I and II).* For language-specific information, see the *DOMAIN FORTRAN User's Guide,* the *DOMAIN Pascal User's Guide,* and the *DOMAIN C User's Guide.* For information about the high-level language debugger, see the *Language Level Debugger Manual.*

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

UPPERCASE	Uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.
lowercase	Lowercase words or characters in formats and command descriptions represent values that you must supply.
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions. In sample Pascal statements, braces assume their Pascal meanings.
< >	Angle brackets enclose a key to be pressed.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down the <CTRL> key while typing the character.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference* manual. Refer to the CRUCR (Create User Change Request) Shell command description. You can view the same information on-line by typing:

```
$ HELP CRUCR <RETURN>
```

For your comments on documentation, a Reader's Response form is located at the back of this manual.

Contents

Chapter 1 Introduction to Graphics Primitives 1-1

1.1. Uses of Graphics Primitives 1-1
1.1.1. Characteristics of Graphics Primitives 1-2

Chapter 2 Displaying Graphics with GPR 2-1

2.1. Displaying Graphic Images 2-1
2.1.1. Pixels and Pixel Values 2-2
2.1.2. Bitmap Dimensions 2-2
2.1.3. The Display Controller 2-2
2.2. Display Devices 2-2
2.3. Generating Images Using a Bit-mapped Raster Scan Device 2-3
2.4. Operation Modes 2-3
2.5. Selecting an Operation Mode 2-4
2.5.1. Borrow-Display Mode 2-4
2.5.2. Direct Mode 2-5
2.5.3. Frame Mode 2-5
2.5.4. No-Display Mode 2-6

Chapter 3 GPR Programming Basics 3-1

3.1. Writing GPR Application Programs 3-1
3.1.1. Insert Files 3-1
3.1.2. Variables 3-1
3.1.3. Initializing the Graphics Package 3-2
3.1.4. Error Reporting 3-3
3.1.5. Developing an Algorithm to Perform a Task 3-3
3.1.6. Terminating a GPR Session 3-3
3.2. Examples Of Initializing GPR 3-3
3.2.1. Pascal Example to Initialize GPR 3-4
3.2.2. FORTRAN Example to Initialize GPR 3-5
3.2.3. C Example to Initialize GPR 3-6

Chapter 4 Drawing and Text Operations 4-1

4.1. The GPR Coordinate System 4-1
4.1.1. Current Position 4-2
4.2. GPR Drawing Routines 4-2
4.3. Line-drawing Examples 4-3
4.3.1. A Program to Draw a Single Line 4-4
4.3.2. A Program to Draw Connected Lines 4-6
4.3.3. A Program to Draw Disconnected Lines 4-8
4.3.4. A Program to Draw an Unfilled Circle 4-10

4.4. GPR Fill Routines 4-12
4.5. Fill Examples 4-13
4.5.1. A Program to Draw and Fill a Triangle 4-14
4.5.2. A Program to Draw and Fill a Polygon 4-16
4.6. A Program to Draw Two Diagonal Lines 4-18
4.6.1. Extending the Line-Drawing Program 4-19
4.7. A Program to Draw a Simple Design 4-20
4.7.1. Extending the Design Program 4-22
4.8. Text Operations 4-23
4.9. A Program Using Text 4-24

Chapter 5 The Cursor and Input Events 5-1

5.1. Using Cursor Control 5-1
5.2. Implementation Restrictions On The Cursor 5-1
5.3. Display Mode and Cursor Control 5-1
5.4. Using Input Operations 5-2
5.4.1. Event Types 5-3
5.4.2. Event Reporting 5-3
5.4.3. Input Routine 5-4
5.5. A Program That Waits For An Event 5-5

Chapter 6 Initial Bitmaps and Attributes 6-1

6.1. Bitmap Structure 6-1
6.2. Bitmap Locations 6-1
6.3. Initial Bitmap Size 6-1
6.3.1. Initial Bitmap in Borrow Mode 6-1
6.3.2. Initial Bitmaps in Frame Mode 6-2
6.3.3. Initial Bitmap in Direct Mode 6-2
6.3.4. Initial Bitmap in No-Display Mode 6-3
6.4. The Current Bitmap 6-3
6.5. Bitmap Attributes 6-3
6.5.1. The Current Attribute Block 6-3
6.5.2. Creating Attribute Blocks 6-3
6.5.3. Making an Attribute Block the Current Attribute Block 6-4
6.6. Other Bitmaps 6-4
6.6.1. External Bitmaps 6-4
6.6.2. Hidden-Display-Memory Bitmaps 6-4
6.7. Listing of Bitmap Attributes and Bitmap Attribute Default Values 6-5
6.8. Changing Attributes 6-8
6.8.1. Retrieving Attributes 6-9
6.9. A Program Using Clipping 6-10
6.10. A Program To Demonstrate Rubberbanding 6-12

Chapter 7 Bitmaps and Bit Block Transfers 7-1

7.1. Bitmaps In Main-memory, Hidden-display Memory and On Disk 7-1
7.1.1. Allocating Bitmaps In Main Memory 7-1
7.1.2. Making Main-memory Bitmaps Current 7-1
7.2. Hidden-display-memory Bitmaps 7-2
7.3. External Bitmaps 7-2
7.4. Using Blts With External Bitmaps and Hidden-display Memory 7-3
7.4.1. Using a Plane Mask With a BLT 7-4
7.4.2. Using Raster Operations With a BLT 7-5
7.4.3. Example of a BLT Operation 7-5
7.5. Example of A Blt With A Raster Operation 7-5
7.6. A Program To Draw A Checker Board 7-7
7.6.1. Procedure check_on_disk 7-10
7.6.2. Procedure draw_design 7-11
7.6.3. Procedure blt_border 7-12
7.6.4. Procedure blt_checker_to_border 7-13

Chapter 8 Color Graphics 8-1

8.1. Display Configurations 8-1
8.1.1. Two-Board Configuration 8-2
8.1.2. Three-Board Configuration 8-2
8.2. Displaying Colors On The Screen 8-3
8.2.1. The Color Map: A Set of Color Values 8-3
8.2.2. The Size of a Color Map 8-5
8.2.3. Color Map for Color Displays: 4-Bit and 8-Bit Formats 8-5
8.2.4. Color Map for Color Displays: 24-Bit Imaging 8-5
8.3. Establishing A Color Map 8-5
8.3.1. Using a Color Map 8-7
8.3.2. FORTRAN Example to Establish a Color Value 8-7
8.3.3. Pascal Example to Establish a Color Value 8-8
8.3.4. Modifying a Color Table 8-9
8.3.5. Changing Pixel Values 8-9
8.3.6. Color Map for Monochromatic Displays 8-9
8.3.7. Saving/Restoring Pixel Values 8-9
8.4. Using Color Display Formats 8-10
8.4.1. Using Imaging Display Formats 8-10
8.4.2. Routines for Imaging Display Formats 8-10
8.5. Color Zoom Operations 8-11
8.6. Color Examples 8-12
8.6.1. A Program to Draw a Rectangle and Text in Color 8-13
8.6.2. A Program to Draw a Design in Color 8-14
8.6.3. A Program to Draw Concentric Circles in Color 8-16

Chapter 9 Graphics Map Files **9-1**

9.1. A Graphics Map File 9-1
9.2. Insert Files 9-1
9.3. Error Messages 9-2
9.4. Programming Example 9-2
9.4.1. Comments on Programming Example 9-2

Appendix A Glossary **A-1**

Appendix B Keyboard Charts **B-1**

Appendix C C Programs **C-1**

Appendix D FORTRAN Programs **D-1**

Index **Index-1**

Illustrations

Figure 2-1.	A Raster Graphic System	2-1
Figure 2-2.	DOMAIN Monochrome Display Configurations	2-3
Figure 4-1.	A 500 x 500 Bitmap	4-1
Figure 4-2.	A Single Line	4-5
Figure 4-3.	Connected Lines	4-7
Figure 4-4.	Disconnected Lines	4-9
Figure 4-5.	A Circle	4-11
Figure 4-6.	A Filled Triangle	4-15
Figure 4-7.	A Filled Polygon	4-17
Figure 4-8.	An "X" Across a Landscape Display	4-19
Figure 4-9.	Four Filled Rectangles within a Box	4-21
Figure 4-10.	Text On A Square	4-26
Figure 5-1.	Cursor Origin Example	5-2
Figure 6-1.	Frame Display	6-2
Figure 6-2.	Clipping Window On A Bitmap	6-6
Figure 7-1.	Information Required for Graphics BLT	7-5
Figure 7-2.	BLT Example: Intersection of Source Bitmap, Source Window, Destination Clipping Window	7-6
Figure 7-3.	Example of BLT with Raster Op Code = 1 (Logical "AND")	7-6
Figure 7-4.	Checker Board with Border	7-7
Figure 7-5.	Border Design	7-7
Figure 8-1.	Four Plane Color System	8-4
Figure 8-2.	Color Value Structure	8-4
Figure 8-3.	From Pixel to Color Map in 24-bit Imaging	8-6
Figure 8-4.	Color Zoom	8-11
Figure B-1.	Low-Profile Keyboard Chart - Translated User Mode	B-2
Figure B-2.	Low-Profile Keyboard	B-3
Figure B-3.	880 Keyboard	B-3
Figure B-4.	880 Keyboard Chart - Translated User Mode	B-4

Tables

Table 6-1.	Raster Operations and Their Functions	6-7
Table 6-2.	Raster Operations: Truth Table	6-8
Table 7-1.	GPR_$OPEN_BITMAP_FILE Access Table	7-3
Table 8-1.	Two-Board Configuration for Color Display	8-1
Table 8-2.	Three-Board Configuration for Color Display	8-2
Table 8-3.	Default Color Map for Monochromatic Displays	8-7
Table 8-4.	Default Color Map for Color Displays	8-8

Chapter 1
Introduction to Graphics Primitives

This chapter briefly outlines the uses and characteristics of the graphics primitives routines (GPR). The graphics primitives library is built into your DOMAIN system. The routines (primitives) that make up the library let you manipulate the least divisible graphic elements to develop high-speed graphics operations. These elements include lines and polylines, other drawing operations, text fonts, pixel values, display types, and bitmaps.

The DOMAIN system also provides the DOMAIN Graphics Metafile Resource system and an optional DOMAIN Core Graphics package.

The DOMAIN Graphics Metafile Resource package (GMR) is a collection of routines that provide the ability to create, display, edit, and store device-independent files of picture data. The package provides routines for developing and storing picture data and displaying the graphic output of that data. The graphics metafile package provides you with the necessary support to build a graphics system "with a memory." The package integrates graphics output capabilities with file handling and editing capabilities. For a detailed description, see *Programming with DOMAIN 2D Graphics Metafile Resource.*

Core, an optional package, which is designed to meet industry standards, provides a high-level graphics environment in which to build portable graphics application systems. For a detailed description of Core graphics, see *Programming With DOMAIN Core Graphics.*

1.1. Uses of Graphics Primitives

The graphics primitives include the following capabilities:

- Drawing lines, circles, and rectangles
- Loading text fonts and manipulating text
- Manipulating graphics with bit block transfers
- Filling polygon areas
- Accommodating input operations
- Setting attributes
- Sharing the display with other processes using windows
- Imaging with an extended color range
- Storing bitmaps externally.

The GPR package uses the following components of the DOMAIN system.

- A display

- Display memory
- Any portion of program memory
- A set of graphics primitive routines
- The Display Manager.

1.1.1. Characteristics of Graphics Primitives

Graphics primitives are device-dependent with respect to the display. However, they are independent of the various display environments. The operating system provides two other sets of calls to manipulate the display:

Display Manager interface
: These program calls, (which begin with PAD), allow you to manipulate pads and frames to display text. You cannot manipulate graphics using these calls.

Display driver interface
: For monochrome displays, there is a lower level of software called the display diriver (SMD) which can be used to do output to the display. Also, for all displays, SMD includes the basic support for keyboard input and cursor manipulation. Most of the display driver calls duplicate functions now provided by the graphic primitives package.

For a description of the calls to the Display Manager interface and the display driver interface see the *Programming With General System Calls*.

GPR routines are independent of the display environments in two ways. First, you can run a program which uses GPR routines on any of the displays.

Second, graphics primitives routines can issue calls to either the Display Manager or the display driver. Therefore, if you use the graphics primitives routines, you can easily change program execution from one display mode to another by changing one option in the initialization routine GPR_$INIT.

Chapter 2
Displaying Graphics with GPR

This chapter describes display configurations, formats, and the modes within which the graphics routines can operate.

2.1. Displaying Graphic Images

DOMAIN displays are bit-mapped, raster-scan devices consisting of three main components: display memory, also called a frame buffer, which stores a matrix of pixel values for images to be displayed; a display monitor that can be monochrome or color; and a display controller that converts digital data stored in frame buffers to video signals that can be displayed on the monitor. See Figure 2-1.

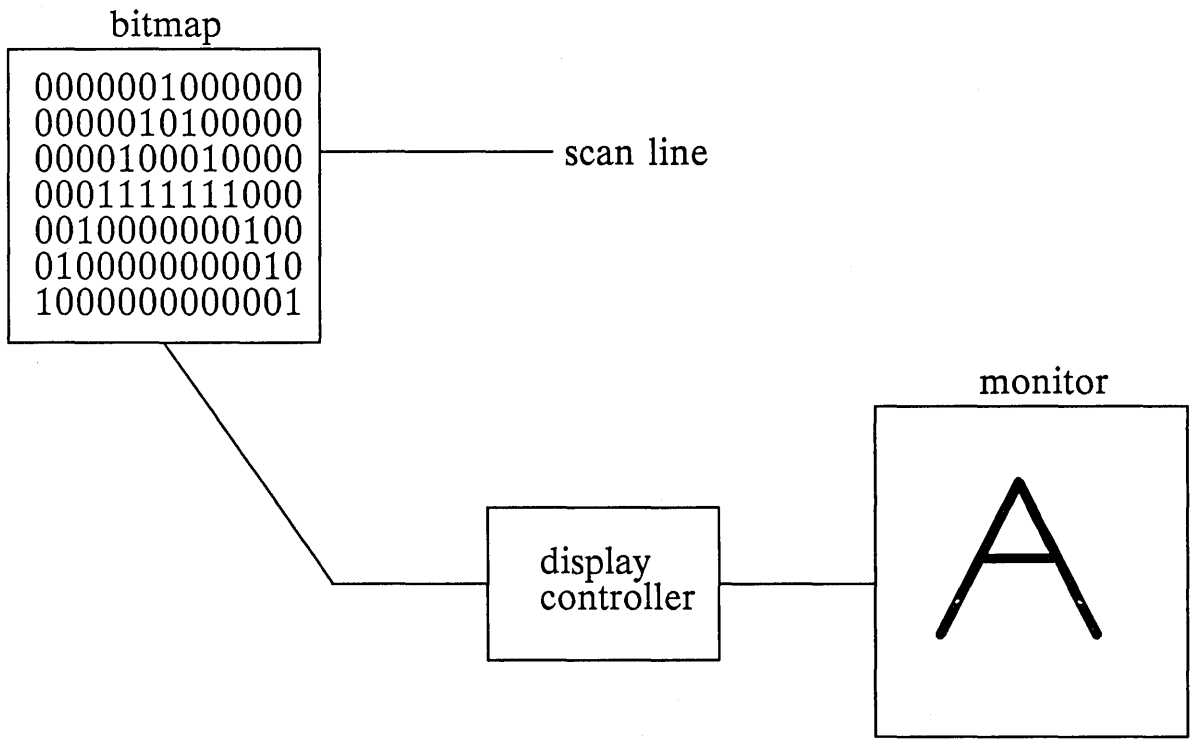


Figure 2-1. A Raster Graphic System

2.1.1. Pixels and Pixel Values

Within a bitmap, an image is stored as a matrix of pixel values. Each pixel value represents *one* addressable picture element or pixel in an array of pixels, which is a raster. For monochrome displays, possible pixel values are 0 and 1. A pixel value of 0 indicates that a particular pixel should not be illuminated on the screen. A pixel value of 1 indicates that a particular pixel should be illuminated. Obviously, only one bit is needed to store a pixel value for monochromatic displays. Pixel values for color displays require more than one bit and are discussed in Chapter 8.

2.1.2. Bitmap Dimensions

Bitmap width is represented by an x coordinate ranging from zero on the left to a maximum defined for the bitmap on the right. Bitmap height is represented by a y coordinate ranging from 0 on the top to a maximum defined for the bitmap on the bottom, in other words - upper-left-hand origin.

Bitmap depth specifies the number of bits of information associated with each pixel value. If a bitmap stores one bit of information for each pixel, as it does for a monochrome display, it need only be one plane deep. If more than one bit of information must be stored for each pixel, the bitmap is several planes deep: one plane for each bit of information. A color display that stores four bits of information for each pixel will use a bitmap four planes deep. A pixel value that uses four bits can have 16 unique values.

Chapter 6 contains more detailed information about bitmaps.

2.1.3. The Display Controller

The display controller is the interface between the bitmap and the display monitor or screen. Its function is to read successive bytes of data from the bitmap and convert this data (0's and 1's) to appropriate video signals which illuminate the pixels. To keep an image displayed, the display controller must continually scan the bitmap one row at a time converting and sending image information to the display. Each row of the bitmap is called a scan line.

2.2. Display Devices

Each Apollo## display device has either a monochrome display or a color display. Currently there are two types of monochrome display devices and two color display devices.

Monochromatic Display Devices

The two types of monochromatic display devices are:

- Monochromatic portrait display
- Monochromatic landscape display.

The monochromatic portrait display is either black and white or black and green. The landscape display is black and white. Each of these has a display memory 1024 pixels wide and 1024 pixels high; however, they differ in the portion of display memory that is visible. The portrait visual

display is 800 pixels wide and 1024 pixels high, while the landscape visual display is 1024 pixels wide and 800 pixels high. The portion of display memory that is not visible is called hidden-display memory (HDM). Figure 2-2 shows the two monochromatic display configurations. Color display configurations are covered in chapter eight.

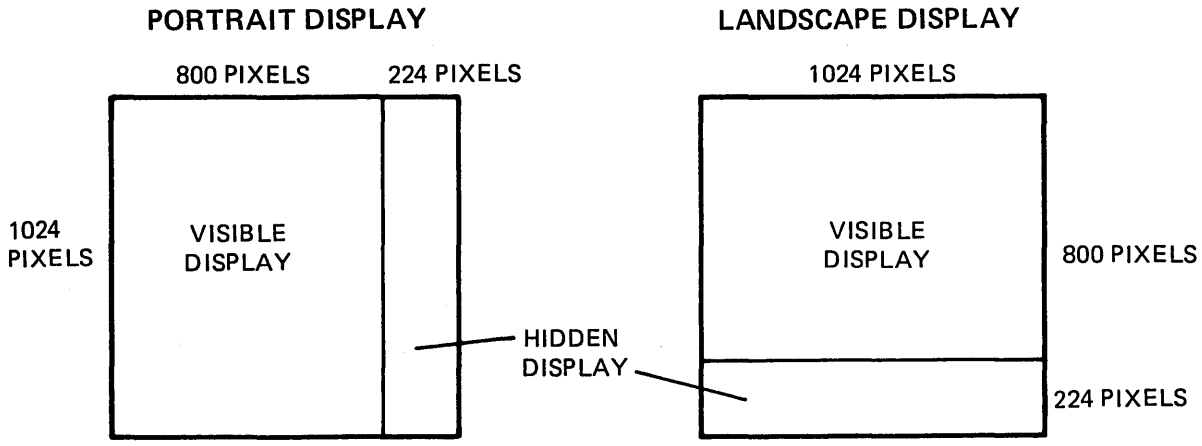


Figure 2-2. DOMAIN Monochrome Display Configurations

2.3. Generating Images Using a Bit-mapped Raster Scan Device

Images are generated on raster-scan devices by computing the position of each pixel to be illuminated in a raster and then illuminating it. This would be an enormous task without GPR routines designed to do most of the work. For example, you can draw a line by calling a GPR line-drawing routine. You supply the two end points of the line and GPR illuminates the correct pixels between the two end points. Geometric shapes are drawn in similiar fashon. You supply the coordinates of the figure to the appropriate GPR routine, and GPR does the rest.

2.4. Operation Modes

Displaying graphics on a screen or storing graphics in memory is the objective of DOMAIN graphics programs. The speed with which this process is accomplished and your ability to perform multiple tasks on your node rely on the operation mode you choose.

GPR has four operation modes. Three of them allow you to display graphics on a screen; one is used for storing graphic images in memory.

The four operation modes are the following:

- Borrow mode and borrow-nc mode
- Direct mode
- Frame mode
- No display mode.

Your choice of operation mode will depend on the advantages of each in relation to your graphics program and the display environment.

2.5. Selecting an Operation Mode

Programs select an operation mode when they initiate a graphics session with GPR_$INIT. Most of the graphics routines can operate within any of these modes, but there are some exceptions. For example, you cannot use clipping in frame mode.

2.5.1. Borrow-Display Mode

In borrow-display mode, the program borrows the full screen and the keyboard from the Display Manager and uses the display driver directly through GPR software. All Display Manager windows disappear from the screen. The Display Manager continues to run during this time. However, it does not write the output of any other processes to the screen or read any keyboard input until the borrowing program returns the display. Input typed ahead into input pads may be read while the display is borrowed. Borrow-display mode is useful for programs that require exclusive use of the entire screen.

A variant of borrow-display mode, borrow-nc ("no clear") mode, allows you to allocate a bitmap in display memory without setting all the pixels to zero. It is identical to borrow-display mode, except that it does not clear the screen. This is useful for copying what is on the screen into a file to save for later display or printing.

Advantages

- Borrow display mode usually provides the best graphics performance.
- You have the entire screen to use as a display area.
- You can use hidden-display memory.
- Borrow_nc mode gives you the option of not clearing the bitmap on initialization.

Disadvantage

- You lose the features offered by the Display Manager while your program is running. For example, you cannot have multiple windows displayed.

2.5.2. Direct Mode

Direct mode is similar to borrow-display mode, but the program borrows a window from the Display Manager instead of borrowing the entire display. The Display Manager relinquishes control of the window in which the program is executing, but continues to run, writing output and processing keyboard input for other windows on the screen. Direct mode offers a graphics application the performance and unrestricted use of display capabilities found in borrow-display mode and, in addition, permits the application to coexist with other activities on the screen. Direct mode should be the preferred mode for most interactive graphics applications.

In direct mode, the program repeatedly acquires and releases the display for brief periods for graphics operations. This gives the advantages of speed of operation while preserving the Display Manager's control over display functions such as changing the window size and scrolling.

Advantages

- Performance is almost as good as in borrow-display mode.
- You retain the use of the Display Manager.
- You can use any rectangular part of the screen.

Disadvantages

- You must synchronize with the Display Manager. (Calls are provided.)
- You must redraw the window when the screen is redrawn.

2.5.3. Frame Mode

Alternately, a graphics program that executes within a frame of a Display Manager pad calls the Display Manager, which interacts with the display driver. A graphics program executes more slowly in frame mode than in borrow-display or direct mode, but frame mode offers some additional Display Manager features:

- A frame provides a "virtual display" that can be larger than the window, allowing you to scroll the window over the frame.
- Frame mode makes it easier to perform ordinary stream I/O to input and transcript pads.
- In frame mode, the Display Manager reproduces the image when necessary.
- The program can leave the image in the pad upon exit so that users can view it at some later time.

Frame mode currently places some restrictions on the GPR operations that are allowed. The programmer's reference describes the individual routines, including their restrictions.

Advantages

- Easy to use: you take care of the graphics calls, and the Display Manager takes care of everything. It is appropriate for simple, noninteractive applications.
- Synchronization with other processes is handled by the DM.
- Reserves an area within a pad for graphics display.
- Allows you to scroll an image out of view. The Display Manager redraws the image when it is pushed or popped.
- Allows use of high-level I/O calls such as READ and WRITE.

Disadvantages

- Graphics programs run much slower than in the other modes.
- There are restrictions on the operations on bitmaps in a frame.
- "Player piano" effect: when an image has had many changes since the last call to GPR_$CLEAR, all such changes are played back. This playing back, which occurs when the window is redrawn for any reason, may take a noticeable period of time to complete.

2.5.4. No-Display Mode

When the program selects no-display at initialization, the GPR initialization routine allocates a bitmap in main memory. The program can then use GPR routines to perform graphic operations to the bitmap, bypassing any screen display entirely. Applications can use no-display mode to create a main memory bitmap, then call graphics map file routines(GMF calls) to write to a file, or send the bitmap to a peripheral device, such as a printer.

Advantages

- You can perform graphic operations to the bitmap while bypassing the display.
- You can create bitmaps larger than the display.

Disadvantages

- Images are not visible on the display.
- You can not use the display after initializing GPR in no-display mode until you terminate GPR and re-initialize it in one of the other modes.

Chapter 3
GPR Programming Basics

This chapter describes the essentials of writing GPR application programs.

3.1. Writing GPR Application Programs

Developing GPR application programs requires several steps. The following subsections describe the steps needed to produce an application program. Some GPR routines are presented in these sections along with brief explainations. For a complete description of these routines, see the *DOMAIN System Call Reference (Volume I).*

3.1.1. Insert Files

In order to write GPR application programs, you must include two insert files. The first one defines certain commonly used system declarations. It must be one of the following:

FORTRAN	Pascal	C
`/sys/ins/base.ins.ftn`	`/sys/ins/base.ins.pas`	`/sys/ins/base.ins.c`

The second insert file allows you to use GPR routines. It must be one of the following:

FORTRAN	Pascal	C
`/sys/ins/gpr.ins.ftn`	`/sys/ins/gpr.ins.pas`	`/sys/ins/gpr.ins.c`

At times you may need other insert files. For example, if you use pad calls within your GPR program, you have to include the appropriate pad insert file. You may also want to create your own insert files to facilitate variable declarations. If you consistently use a particular set of variables, you can put them in an insert file and then include the insert file in any program that uses those variables.

Many of the programming examples used in this manual include the following insert file:

FORTRAN	Pascal	C
`/sys/ins/time.ins.ftn`	`/sys/ins/time.ins.pas`	`/sys/ins/time.ins.c`

This enables the programs to use the TIME_$WAIT routine, which keeps an image displayed on the screen for a specified period of time.

3.1.2. Variables

Variables used as parameters in GPR calls must be declared to correspond to the data types used on our system. The documentation listed with each GPR call defines the data-types of the parameters. In cases where declaring the necessary variables might not be straightforward, for example record types or enumerated types in FORTRAN, you are directed to the data-type section at the beginning of the GPR calls in the *DOMAIN System Call Refrence (Volume I).*

3.1.3. Initializing the Graphics Package

To execute GPR calls in an application program, you must first initialize the package. You do this by calling the routine GPR_$INIT in the application program. You are allowed to perform non-GPR operations before initializing GPR, but you cannot execute any GPR routines except GPR_$INQ_CONFIG until GPR is initialized.

The form of GPR_$INIT is the following:

```
GPR_$INIT(op_mode, unit, size, hi_plane_id, init_bitmap_disc, status)
```

Input Parameters

op_mode — The operation mode for the application program. The possible values are the following:

```
GPR_$BORROW
GPR_$BORROW_NC
GPR_$DIRECT
GPR_$FRAME
GPR_$NO_DISPLAY
```

unit — The value for this parameter depends on the operation mode. The possible operation modes and the corresponding values that unit can hold are the following:

Operation Mode	UNIT
GPR_$BORROW	1
GPR_$BORROW_NC	1
GPR_$FRAME	Stream id of the window or window pane in which the graphics is to be performed.
GPR_$DIRECT	
GPR_$NO_DISPLAY	Any value

size — The width and height of the initial bitmap, in pixels.

hi_plane_id — The identifier of the bitmap's highest plane. Valid values are the following:

For display memory bitmaps:

```
0     For monochrome displays (1 plane)
0 - 3 For color displays in two-board configuration
      (1 - 4 planes)
0 - 7 For color displays in three-board configuration
      (1 - 8 planes).
```

For main memory bitmaps:

```
0 - 7 for all displays (1 - 8 planes).
```

Output Parameters

init_bitmap_desc
: A unique descriptor for the initial bitmap. All bitmaps have descriptors.

status
: The standard system error indicator.

3.1.4. Error Reporting

All GPR calls return a 32-bit status code, which indicates whether or not the call executed successfully. If the call succeeded, the value of the status code is STATUS_$OK (0). If the call failed, the returned value gives the nature of the failure and where it occurred.

The GPR insert file lists all the possible error codes. Error Reporting is covered in detail in the *Programming With General System Calls*.

3.1.5. Developing an Algorithm to Perform a Task

The next step in the development of a GPR application program is to prepare an algorithm using GPR routines to accomplish the task at hand. See Chapter 4 for some sample algorithms.

3.1.6. Terminating a GPR Session

Use GPR_$TERMINATE to terminate your GPR session . You can initialize and terminate GPR as often as you like within a graphics program. For example, you may initialize GPR in borrow mode and perform some task, terminate GPR, re-initialize GPR in direct mode, and perform some other task.

3.2. Examples Of Initializing GPR

Three language-specific examples to initialize GPR in borrow mode with a bitmap having dimensions of 500 x 500 are listed in this section.

3.2.1. Pascal Example to Initialize GPR

```
Program example;

 {insert files}
 %nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %list;


 var
   size        : gpr_$offset_t;   {size of the initial bitmap}

   init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}

   mode        : gpr_$display_mode_t;  {operation mode}

   hi_plane_id : gpr_$plane_t;  {highest plane in bitmap}

   delete_display : boolean; {This value is ignored in borrow mode.}

   status : status_$t; {error code}

 begin
      .
      .
      .
      size.x_size := 500;
      size.y_size := 500;
      hi_plane_id := 0; {bitmap with one plane}
      gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status);
      .
      .
      .
      gpr_$terminate(delete_display,status);

 end.
```

3.2.2. FORTRAN Example to Initialize GPR

```
Program example

 %include '/sys/ins/base.ins.ftn'
 %include '/sys/ins/gpr.ins.ftn'

    integer*2 size(2)  {array to hold the size of}
                       {the initial bitmap}

    integer*4 init_bitmap {descriptor of the inital bitmap}

    integer*2 mode {data structure to hold the operation}
                   {mode}

    integer*2 hi_plane_id {highest plane number in bitmap}

    integer*4 status {error code}

    logical delete_display {This value is ignored in borrow mode.}

      .
      .
      .

    size(1) = 500
    size(2) = 500
    mode = gpr_$borrow
    hi_plane_id = 0
    call gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status)
      .
      .
      .
    call gpr_$terminate(delete_display,status)

  end
```

3.2.3. C Example to Initialize GPR

```
/* Program example */

#nolist
#include "stdio.h"
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#list

gpr_$offset_t       size;  /* Size of initial bitmap */

linteger            init_bitmap;  /*descriptor of initial bitmap. */

gpr_$display_mode_t     mode;  /* operation mode */

gpr_$plane_t        hi_plane_id; /* highest plane number in bitmap */

status_$t           status;  /* error code */

boolean             delete_display;  /* This value is ignored in */
                                     /* borrow mode. */


main()
{


    size.x_size = 500;
    size.y_size = 500;
    mode = gpr_$borrow;
    hi_plane_id = 0;/

    gpr_$init(mode,(short)1,size,hi_plane_id,init_bitmap,status);
      .
      .
      .
    gpr_$terminate(delete_display,status);

}
```

Chapter 4
Drawing and Text Operations

This chapter introduces GPR drawing, filling and text routines. Several programming examples are presented in order to demonstrate some of these routines in actual GPR application programs.

4.1. The GPR Coordinate System

The GPR coordinate system places the coordinate origin at the top left-hand corner of a bitmap. The x values increase to the right, and y values increase downwards. Coordinates for all drawing operations are relative to the coordinate origin. You can change the coordinate origin using the routine: GPR_$SET_COORDINATE_ORIGIN.

If you initialize a 500 by 500 bitmap, the corners of your bitmap will have the coordinates displayed in Figure 4-1.

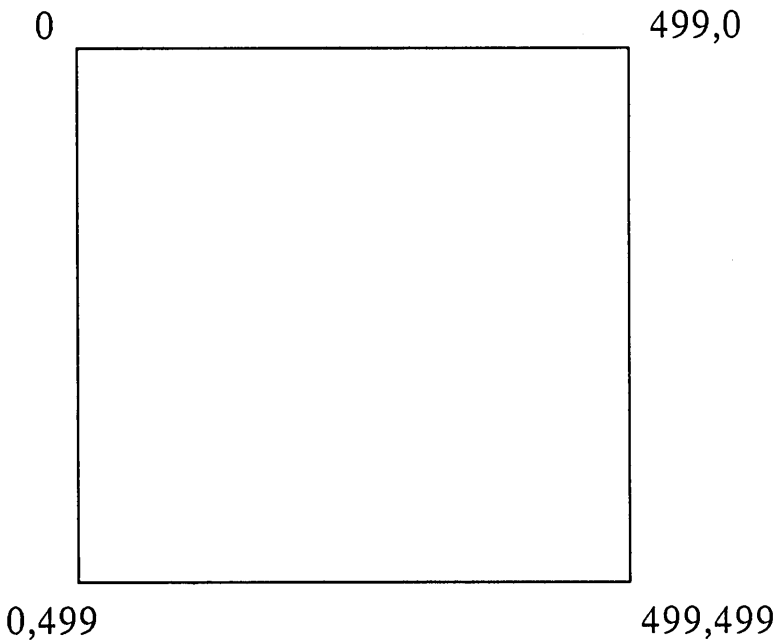


Figure 4-1. A 500 x 500 Bitmap

4.1.1. Current Position

All drawing and text operations begin at the current position. After an application program is initialized with GPR_$INIT, the current position is set to the coordinate origin (0,0). After you use some drawing or text operations, the current position gets updated to a new current position. (See examples in this chapter. Not all drawing routines update the current position.) The routine, GPR_$INQ_CP returns the x and y coordinates of the current position.

To begin a drawing or text operation at a specific point in a bitmap, it is often necessary to move the current position. The routine, GPR_$MOVE moves the current position to the coordinates specfied without drawing a line.

4.2. GPR Drawing Routines

GPR provides several routines to draw geometric figures, lines, and arcs. The calls are listed below followed by brief descriptions. The parameters have been omitted.

GPR_$ARC_3P
: Draws an arc from the current position, through two other points. The current position is updated to the coordinates of the second point, which is the last point on the arc.

GPR_$CIRCLE Draws a circle with a specified radius around a specified center point. This routine does not update the current position.

GPR_$DRAW_BOX
: Draws an unfilled box given two opposing corners. This routine does not update the current position.

GPR_$LINE Draws a line from the current position to the specified endpoint. The current position is updated to the coordinates of the specified endpoint.

GPR_$MULTILINE
: Draws a series of disconnected lines. The current position is updated with each line that is drawn.

GPR_$POLYLINE
: Draws a series of connected lines. The current position is updated with each line that is drawn.

GPR_$SPLINE_CUBIC_P
: Draws a parametric cubic spline from the current position through a list of control points. The current position is updated to the coordinates of the last control point.

GPR_$SPLINE_CUBIC_X
: Draws a cubic spline as a function of x from the current position through a list of control points. The current position is updated to the coordinates of the last control point.

GPR_$SPLINE_CUBIC_Y

Draws a cubic spline as a function of y from the current position through a list of control points. The current position is updated to the coordinates of the last control point.

4.3. Line-drawing Examples

Four programming examples are presented in this section to demonstrate how the following GPR drawing routines work in relation to the coordinate origin and the current position:

- GPR_$LINE
- GPR_$POLYLINE
- GPR_$MULTILINE
- GPR_$CIRCLE.

Notice that some routines change the current position and some do not. Also, notice that all drawing operations are relative to the coordinate origin. These examples are translated into FORTRAN and C in the language-specific appendices.

TIME_$WAIT, a call used in all the programs in this chapter and some of the programs in other chapters, is not a GPR routine. It is used to keep an image displayed on the screen for a specified period of time. This call is documented in the *DOMAIN System Call Refrence (Volume II).*

4.3.1. A Program to Draw a Single Line

This program draws a single line from the coordinate origin (0,0) to the endpoint with coordinates (400,500).

After drawing the line, the current position is updated to (400,500). See Figure 4-2.

```
Program draw_a_line;
 %nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %include '/sys/ins/time.ins.pas';
 %list;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;
 var
   init_bitmap_size : gpr_$offset_t;   {size of the initial bitmap}
   init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
   mode        : gpr_$display_mode_t := gpr_$borrow;
   hi_plane_id : gpr_$plane_t := 0;  {highest plane in bitmap}
   delete_display : boolean; {This value is ignored in borrow mode.}
   pause  : time_$clock_t;
   status : status_$t; {error code}
 begin
      init_bitmap_size.x_size := 700;
      init_bitmap_size.y_size := 700;
      gpr_$init(mode,1,init_bitmap_size,hi_plane_id,init_bitmap,status);
      gpr_$line(400,500,status);
      {Keep figure displayed on the screen for five seconds.}
      pause.low32  := five_seconds;
      pause.high16 := 0;
      time_$wait( time_$relative, pause, status );
      gpr_$terminate(delete_display,status); {Terminate gpr.}
 end.
```

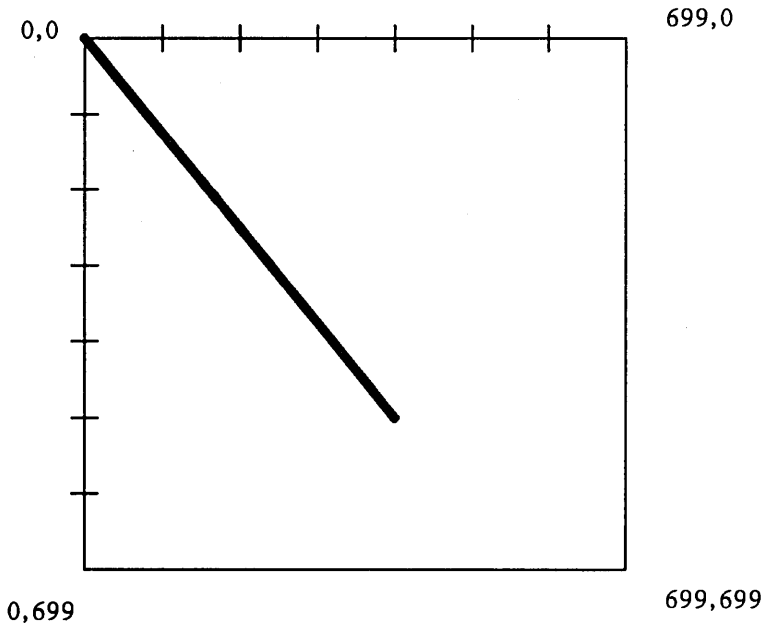


Figure 4-2. A Single Line

4.3.2. A Program to Draw Connected Lines

This program draws three connected lines. The first line begins at the point with coordinates (30,30), the second line at (200,300), and the third line at (400,400). The third line terminates at (300,200). See Figure 4-3.

The routine GPR_$POLYLINE requires that the x and y coordinates of successive coordinate positions be passed in two arrays. The number of coordinate positions is passed in a two-byte integer.

The current position is updated to (200,300), (400,400) and (300,200) respectively.

```
Program draw_connected_lines;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;
const
   one_second = 250000;
   five_seconds = 5 * one_second;
var
  size        : gpr_$offset_t;   {size of the initial bitmap}
  init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
  mode        : gpr_$display_mode_t := gpr_$borrow;
  hi_plane_id : gpr_$plane_t := 0;  {highest plane in bitmap}
  x : gpr_$coordinate_array_t := [200,400,300]; {an array of x coord.}
  y : gpr_$coordinate_array_t := [300,400,200]; {an array of y coord.}
  numb_of_pts     : integer := 3;
  delete_display : boolean; {This value is ignored in borrow mode.}
  pause  : time_$clock_t;
  status : status_$t; {error code}
begin
     size.x_size := 700;
     size.y_size := 700;
     gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status);
     gpr_$move(30,30,status);
     gpr_$polyline(x,y,numb_of_pts,status);
     {Keep figure displayed on the screen for five seconds.}
     pause.low32  := five_seconds;
     pause.high16 := 0;
     time_$wait( time_$relative, pause, status );
     gpr_$terminate(delete_display,status);
end.
```

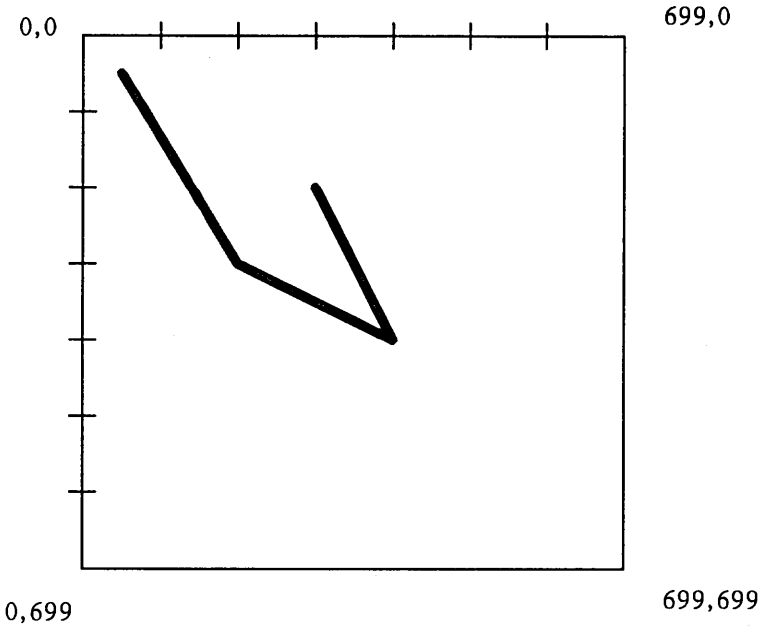


Figure 4-3. Connected Lines

4.3.3. A Program to Draw Disconnected Lines

This program draws three disconnected lines. The coordinates of the endpoints of the first line are (100,100), (400,100); the coordinates of the endpoints of the second line are (100,200), (400,200); and the coordinates of the endpoints of the third line are (100,300), (400,300). See Figure 4-4.

The routine GPR_$MULTILINE requires that the x and y coordinates of successive coordinate positions be passed in two separate arrays. The number of coordinate positions is passed in a two-byte integer. The current position is updated to (400,100), (400,200) and (400,300) respectively.

```
Program disconnected_lines;
 %nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %include '/sys/ins/time.ins.pas';
 %list;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;
 var
   size        : gpr_$offset_t;   {size of the initial bitmap}
   init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
   mode        : gpr_$display_mode_t := gpr_$borrow;
   hi_plane_id : gpr_$plane_t := 0;  {highest plane in bitmap}
   x : gpr_$coordinate_array_t:=[100,400,100,400,100,400];{an array of x coord.}
   y : gpr_$coordinate_array_t:=[100,100,200,200,300,300];{an array of y coord.}
   numb_of_pts : integer := 6; {number of coordinate positions}
   delete_display : boolean; {This value is ignored in borrow mode.}
   pause  : time_$clock_t;
   status : status_$t; {error code}
 begin
      size.x_size := 700;
      size.y_size := 700;
      gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status);
      gpr_$multiline(x,y,numb_of_pts,status);
      {Keep figure displayed on the screen for five seconds.}
      pause.low32  := five_seconds;
      pause.high16 := 0;
      time_$wait( time_$relative, pause, status );
      gpr_$terminate(delete_display,status);
 end.
```

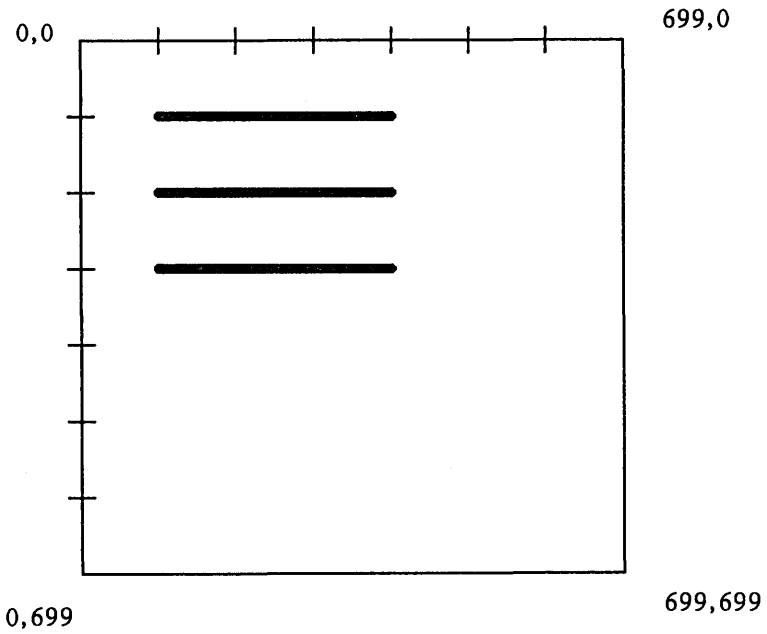


Figure 4-4. Disconnected Lines

4.3.4. A Program to Draw an Unfilled Circle

This program draws an unfilled circle centered at the coordinate position (300,300) with a radius of 200. See Figure 4-5.

The routine GPR_$CIRCLE requires that the x and y coordinates of the center point be passed in a two-element array. This call does not update the current position.

```
Program draw_circle;
 %nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %include '/sys/ins/time.ins.pas';
 %list;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;
 var
   size        : gpr_$offset_t;   {size of the initial bitmap}
   init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
   mode        : gpr_$display_mode_t := gpr_$borrow;
   hi_plane_id : gpr_$plane_t := 0;  {highest plane in bitmap}
   center : gpr_$position_t := [300,300];
   radius : integer := 200;
   delete_display : boolean; {This value is ignored in borrow mode.}
   status : status_$t; {error code}
   pause  : time_$clock_t;
 begin
      size.x_size := 700;
      size.y_size := 700;
      gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status);
      gpr_$circle(center,radius,status);
      {Keep figure displayed on the screen for five seconds.}
      pause.low32  := five_seconds;
      pause.high16 := 0;
      time_$wait( time_$relative, pause, status );
      gpr_$terminate(delete_display,status);
 end.
```

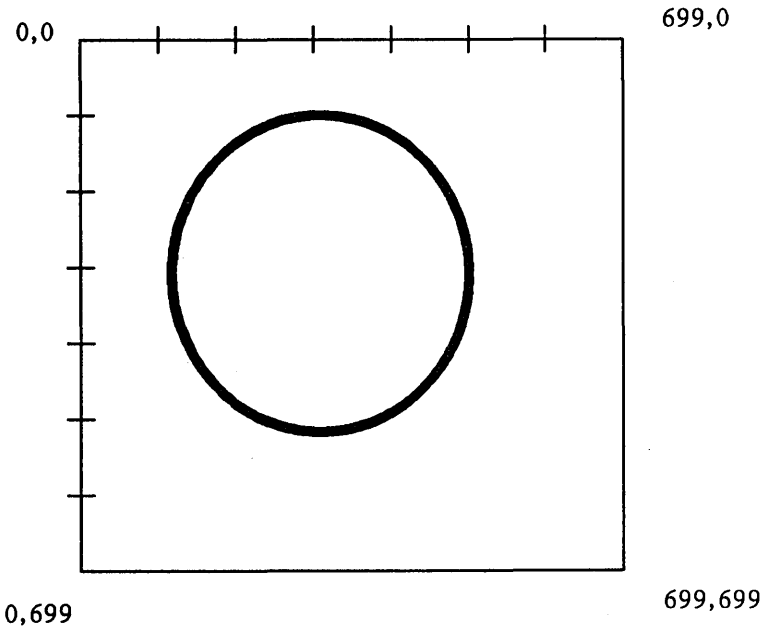


Figure 4-5. A Circle

4.4. GPR Fill Routines

The rectangle, triangle, trapezoid and multitrapezoid routines fill in a specified rectangle, triangle, trapezoid, or list of trapezoids. The rectangle routine fills a rectangle by writing the current fill value into the rectangle without regard to its previous contents or the raster operations in effect. (Raster operations are covered in section 6.7.)

The triangle, trapezoid, and multitrapezoid routines compute the current fill value the same way as in the rectangle routine.

The polygon routines open and define the boundaries of a polygon, and either close and fill the polygon immediately, or close the polygon and return its decomposition to the program for later drawing and filling. The routine GPR_$PGON_POLYLINE does not draw a polygon; the routine defines a series of line segments for decomposition for filling operations.

A polygon's boundary consists of one or more closed loops of edges. The polygon routine GPR_$START_PGON establishes the starting point for a new loop, closing off the old loop if necessary. The polygon routine GPR_$PGON_POLYLINE defines a series of edges in the current loop.

The polygon routines GPR_$CLOSE_FILL_PGON and GPR_$CLOSE_RETURN_PGON close a polygon by decomposing it. The graphics primitives define a trapezoid as a quadrilateral with two horizontally parallel sides. The polygon routines examine the polygon and break it into trapezoids that can be filled immediately or returned in an array to the program. At a later time, the program can reconstruct the polygon by filling the saved trapezoids with the multitrapezoid routine.

The polygon routines define the interior of a polygon to be all points from which a line can originate and cross the polygon boundary an odd number of times. The graphics primitives fill polygon interiors with the current fill value regardless of previous contents.

GPR_$CIRCLE_FILLED
: Draws and fills a circle with a specified radius around a specified center point. The current position is not updated.

GPR_$RECTANGLE
: Draws and fills a rectangle. The current position is not updated.

GPR_$TRIANGLE
: Draws and fills a triangle. The current position is not updated.

GPR_$TRAPEZOID
: Draws and fills a trapezoid. The current position is not updated.

GPR_$MULTITRAPEZOID
: Draws and fills one or more trapezoids. The current position is not updated.

GPR_$START_PGON
: Defines the starting position to create a loop of edges for a polygon boundary. The current position is not updated.

GPR_$PGON_POLYLINE
: Defines a series of line segments forming part of a polygon boundary. The current position is not updated.

GPR_$CLOSE_FILL_PGON
: Closes and fills the currently open polygon. The current position is not updated.

GPR_$CLOSE_RETURN_PGON
: Closes the currently open polygon and returns the list of trapezoids within its interior. The current position is not updated.

4.5. Fill Examples

Two programming examples are presented in this section to demonstrate how GPR fill operations are performed. The following GPR routines are presented in the examples:

- GPR_$TRIANGLE
- GPR_$START_PGON
- GPR_$PGON_POLYLINE
- GPR_CLOSE_FILL_PGON.

4.5.1. A Program to Draw and Fill a Triangle

This program draws and fills a triangle with verticies at (100,100), (400,100), and (400,400). See Figure 4-6.

The routine GPR_$TRIANGLE requires that the x and y coordinates of each vertex be passed in a two-element array. Three arrays, vertex_1, vertex_2, and vertex_3 are used. This call does not update the current position after drawing and filling the triangle.

```
Program filled_triangle;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;
const
   one_second = 250000;
   five_seconds = 5 * one_second;
var
  size        : gpr_$offset_t;   {size of the initial bitmap}
  init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
  mode        : gpr_$display_mode_t := gpr_$borrow;
  hi_plane_id : gpr_$plane_t := 0;  {highest plane in bitmap}
  vertex_1    : gpr_$position_t := [100,100];
  vertex_2    : gpr_$position_t := [400,100];
  vertex_3    : gpr_$position_t := [400,400];
  delete_display : boolean; {This value is ignored in borrow mode.}
  pause  : time_$clock_t;
  status : status_$t; {error code}
begin
     size.x_size := 700;
     size.y_size := 700;
     gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status);
     gpr_$triangle(vertex_1,vertex_2,vertex_3,status);
     {Keep figure displayed on the screen for five seconds.}
     pause.low32  := five_seconds;
     pause.high16 := 0;
     time_$wait( time_$relative, pause, status );
     gpr_$terminate(delete_display,status);
end.
```

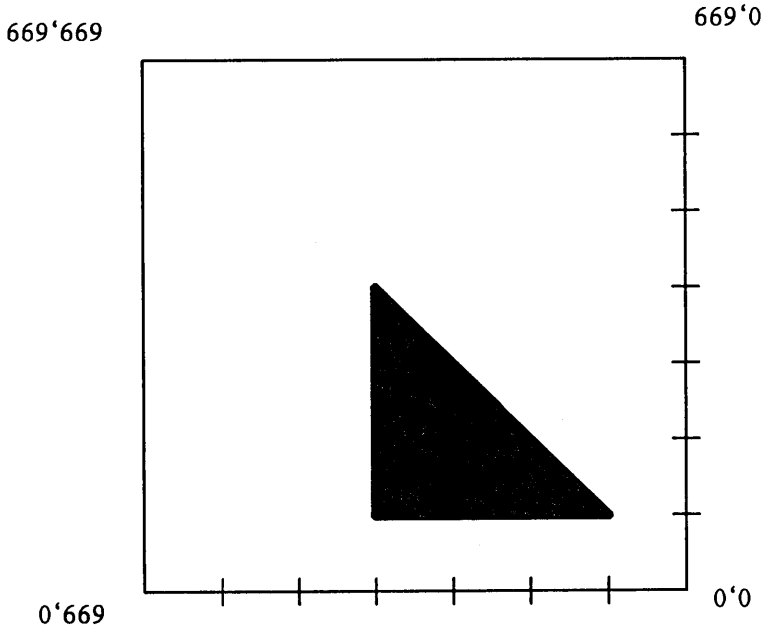


Figure 4-6. A Filled Triangle

4.5.2. A Program to Draw and Fill a Polygon

This program draws and fills a polygon with verticies at the points with coordinates (200,200), (300,300), (300,400), (100,400), and (100,300). See Figure 4-7.

The routine GPR_$START_PGON sets the starting position of the polygon at (200,200). The routine GPR_$PGON_POLYLINE defines four lines. The endpoints of the first line are (200,200), (300,300); the endpoints of the second line are (300,300), (300,400); the endpoints of the third line are (300,400), (100,400); and the endpoints of the fourth line are (100,400), (100,300). The routine GPR_$CLOSE_FILL_PGON closes the polygon by defining a line from the point (100,300) to the point (200,200) and then fills the polygon.

```
Program fill_pgon;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;
const
   one_second = 250000;
   five_seconds = 5 * one_second;
var
  size        : gpr_$offset_t;   {size of the initial bitmap}
  init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
  mode        : gpr_$display_mode_t := gpr_$borrow;
  hi_plane_id : gpr_$plane_t := 0;  {highest plane in bitmap}
  x : gpr_$coordinate_array_t := [300,300,100,100];
  y : gpr_$coordinate_array_t := [300,400,400,300];
  npositions : integer := 4;
  delete_display : boolean; {This value is ignored in borrow mode.}
  i      : integer;
  pause  : time_$clock_t;
  status : status_$t; {error code}
begin
     size.x_size := 700;
     size.y_size := 700;
     gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status);
     gpr_$start_pgon(200,200,status);
     gpr_$pgon_polyline(x,y,npositions,status);
     gpr_$close_fill_pgon(status);
     {Keep figure displayed on the screen for five seconds.}
     pause.low32  := five_seconds;
     pause.high16 := 0;
     time_$wait( time_$relative, pause, status );
     gpr_$terminate(delete_display,status);
end.
```

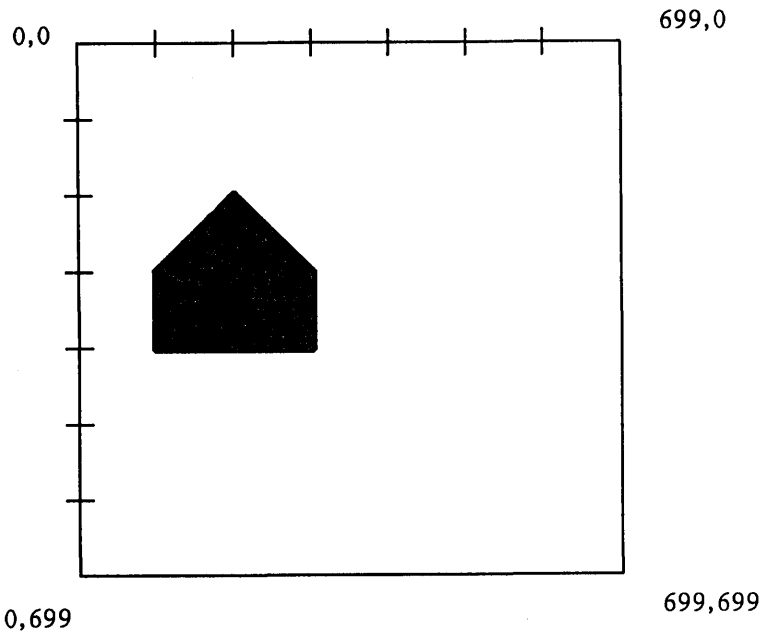


Figure 4-7. A Filled Polygon

4.6. A Program to Draw Two Diagonal Lines

The program presented in this section initializes GPR in Borrow mode with dimensions of 1024 x 800. The program draws the first line across the screen from the top left-hand corner of the bitmap to the bottom right-hand corner. After drawing the first line, the coordinates of the new current position are (1023,799). To draw a line from the top right-hand corner of the bitmap to the bottom left-hand corner, the current position is moved to the top right-hand corner of the bitmap using GPR_$MOVE.

If you are not using a node with a landscape display, you will have to modify the parameters used in GPR_$MOVE and GPR_$LINE to get the same results. For a portrait display the parameters for GPR_$LINE are (799,1023) to draw the first line and (0,1023) for the second line. The parameters for GPR_$MOVE are (799,0). On DN600 and DN660 color nodes the parameters for GPR_$LINE are (1023,1023) and (0,1023), respectively. The coordinates for GPR_$MOVE are (1023,0).

The bitmap dimensions used in this program represent a whole landscape display. When using other displays you must modify these dimensions. If you wish, you can initialize GPR with dimensions of 1024 x 1024 regardless of the display you are using. Fortunately, this does not create an error. GPR_$INIT will automatically allocate a bitmap with dimensions of 800 x 1024 for a portrait display, 1024 x 800 for a landscape display and, 1024 x 1024 for a DN600 or DN600 color display. GPR_$INIT, however, will not allocate a bitmap larger than the dimensions you provide. You can demonstrate this by initializing GPR with bitmap dimensions smaller than the size of the display you are using.

Figure 4-8 shows an "X" drawn across a landscape display.

```
PROGRAM draw_an_X;
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%INCLUDE '/sys/ins/time.ins.pas';
%LIST;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;

var
    status : status_$t;
    delete_display : boolean;
    disp_bm_size : gpr_$offset_t;
    init_bitmap  : gpr_$bitmap_desc_t;
    hi_plane_id  : gpr_$plane_t := 0;
    i      : integer;
    pause : time_$clock_t;
BEGIN

      {Declare the size of the bitmap you will be using.}
      disp_bm_size.x_size := 1024;
      disp_bm_size.y_size := 800;

      {Initialize GPR}
      gpr_$init(gpr_$borrow,1,disp_bm_size,hi_plane_id,
                init_bitmap,status);
      {Draw one line.}
```

```
        gpr_$line( 1023, 799, status);
        {Move the current position}
        gpr_$move(1023,0,status);
        {Draw the second line.}
        gpr_$line(0, 799,status);
        {Keep figure displayed on the screen for five seconds.}
        pause.low32  := five_seconds;
        pause.high16 := 0;
        time_$wait( time_$relative, pause, status );
        gpr_$terminate(delete_display,status);
END.
```

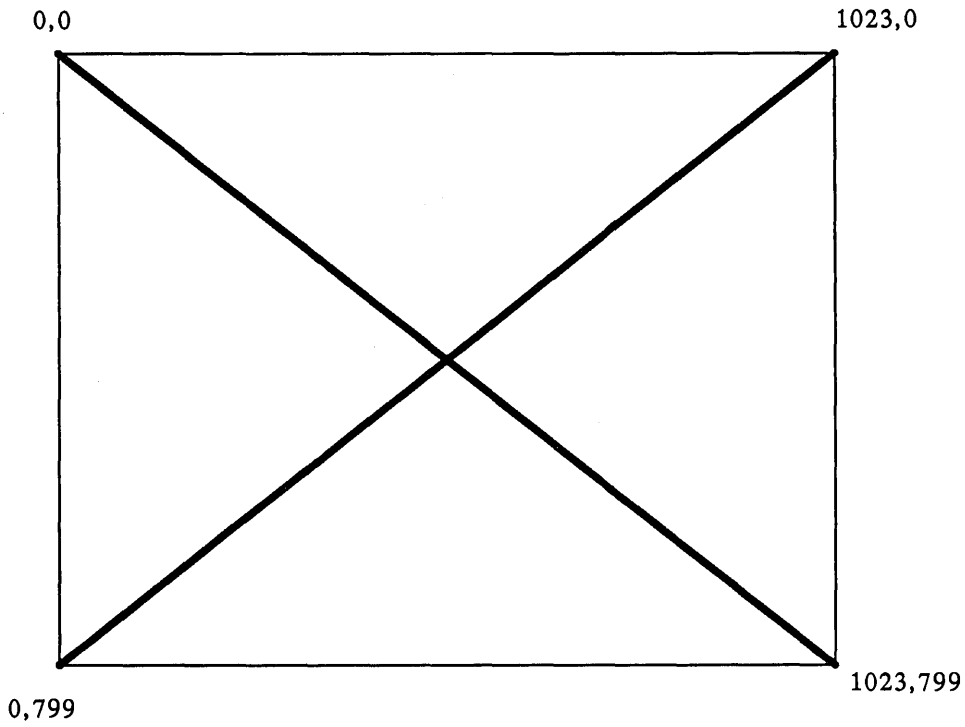


Figure 4-8. An "X" Across a Landscape Display

4.6.1. Extending the Line-Drawing Program

A program to draw an "X" across any size window when using direct mode is presented below. Notice that when using direct mode the display must be acquired using the routine GPR_$ACQUIRE_DISPLAY. The routine GPR_$RELEASE_DISPLAY releases the display.

You can see how this program operates in a frame by initializing GPR in frame mode.

```
PROGRAM draw_an_X;
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%INCLUDE '/sys/ins/time.ins.pas';
%LIST;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;

var
    status : status_$t;
    mode   : gpr_$display_mode_t := gpr_$direct;
    pause  : time_$clock_t;
    delete_display : boolean;
    disp_bm_size   : gpr_$offset_t;
    init_bitmap    : gpr_$bitmap_desc_t;
    hi_plane_id    : gpr_$plane_t := 0;
    num_of_planes  : gpr_$plane_t;
    i : integer;
    unobscured, scure: boolean;
    x,y : integer;
BEGIN
      {Declare the size of the bitmap you will be using.}
      disp_bm_size.x_size := 1024;
      disp_bm_size.y_size := 1024;
      gpr_$init(mode,1,disp_bm_size,hi_plane_id,init_bitmap,status);
      unobscured := gpr_$acquire_display(status); {Acquire the display.}
      {Find out the size of the bitmap.}
      gpr_$inq_bitmap_dimensions(init_bitmap,disp_bm_size,
                                 num_of_planes,status);
      x := disp_bm_size.x_size;
      y := disp_bm_size.y_size;
      gpr_$line( x-1, y-1, status);  {Draw one line.}
      gpr_$move(x-1,0,status);  {Move the current position}
      gpr_$line(0,y-1,status); {Draw the second line.}
      gpr_$release_display(status);
      {Keep figure displayed on the screen for five seconds.}
      pause.low32  := five_seconds;
      pause.high16 := 0;
      time_$wait( time_$relative, pause, status );
      gpr_$terminate(delete_display,status);  {Terminate GPR.}

END.
```

4.7. A Program to Draw a Simple Design

The program presented in this section draws the design in figure 4-9 on the screen.

This program initializes GPR in borrow mode with dimensions of 1024 x 800. The program draws the outside unfilled box first, then it draws the four filled squares and finishes by drawing the two connecting lines. There is no special reason or advantage to the order chosen.

Notice that GPR_$DRAW_BOX takes the coordinates of two opposite corners while GPR_$RECTANGLE requires a starting position (x and y coordinates) and a length and width.

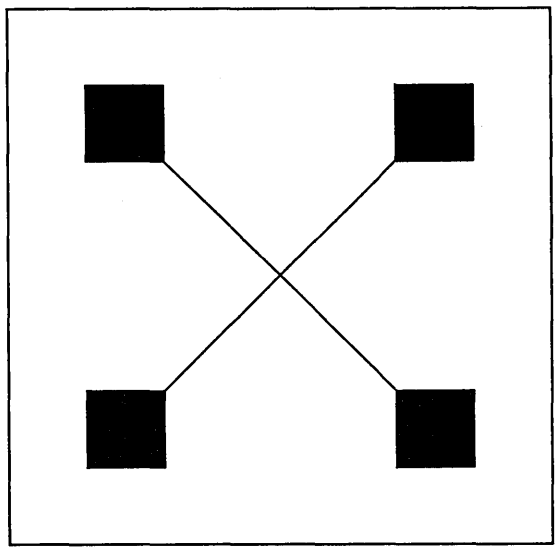


Figure 4-9. Four Filled Rectangles within a Box

GPR_$MOVE relocates the current position to (300,300) before GPR_$LINE draws the first line (the choice of which line to draw first is arbitrary). After the line is drawn, the current position changes to the destination of the line just drawn. To draw the second line, GPR_$MOVE relocates the current position again, this time to (300,500). With this complete, GPR_$LINE draws the second line.

```
program connect_four;
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%INCLUDE '/sys/ins/time.ins.pas';
%LIST;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;
 var
    init_bitmap : gpr_$bitmap_desc_t;
    st    : status_$t;
    mode : gpr_$display_mode_t := gpr_$borrow;
    x,y,x1,y1 :integer;
    rectangle : gpr_$window_t;
```

```
    disp_bm_size : gpr_$offset_t := [1024,800]; {size of initial bitmap}
    pause        : time_$clock_t;
    hi_plane_id  : gpr_$plane_t := 0;
BEGIN
      x := 200; x1 := 600; y := 200; y1 := 600; {dimensions of box}
      {starting position of 1st rectangle}
      rectangle.window_base.x_coord := 250;
      rectangle.window_base.y_coord := 250;
      rectangle.window_size.x_size := 50; {width of each rectangle}
      rectangle.window_size.y_size := 50; {height of each rectangle}
      gpr_$init(mode,1,disp_bm_size,hi_plane_id,init_bitmap,st );
      gpr_$set_auto_refresh(true,st);
      gpr_$draw_box(x,y,x1,y1,st); {Draw an unfilled box.}
      gpr_$rectangle(rectangle,st); {Draw a filled rectangle.}

      {Draw three more filled rectangles within the unfilled box.}
      rectangle.window_base.x_coord := 500;
      rectangle.window_base.y_coord := 250;
      gpr_$rectangle(rectangle,st);

      rectangle.window_base.x_coord := 250;
      rectangle.window_base.y_coord := 500;
      gpr_$rectangle(rectangle,st);

      rectangle.window_base.x_coord := 500;
      rectangle.window_base.y_coord := 500;
      gpr_$rectangle(rectangle,st);

      gpr_$move(300,300,st);  {Move the current position.}
      gpr_$line(500,500,st);  {Draw a line connecting two rectangles.}

      gpr_$move(300,500,st);
      gpr_$line(500,300,st);  {Draw a line connecting two rectangles.}

      {Keep figure displayed on the screen for five seconds.}
      pause.low32  := five_seconds;
      pause.high16 := 0;
      time_$wait( time_$relative, pause, st);

      gpr_$terminate( false, st );  {Terminate the graphics session.}
END.
```

4.7.1. Extending the Design Program

Try changing the Operation mode for this program to Direct mode (Remember, you must acquire the display in Direct mode.) Notice that only a portion of the design is visible unless you have a large window. If this is the case, enlarge the window and run the program again.

At this time, there is a situation worth mentioning. Open a few windows on your screen and run the program again in direct mode. With the design displayed in the window, pop one or more windows so that the window with the display gets fully or partially obscured. Pop the window with the design back up to the top. Notice that the window becomes blank. There are two remedies for this situation.

1. You can include the call GPR_$SET_AUTO_REFRESH in the application

program. This will signal the Display Manager to automatically redraw the contents of the window whenever the window grows or is popped. The Display Manager only redraws what was in the window before it was obscured or had grown. For example, if only a portion of a drawing is displayed in a window because the window was too small (your drawing got clipped), only that portion of the drawing will be redrawn if the window has grown. For this reason, GPR_$SET_AUTO_REFRESH is most useful to handle redrawing when windows get popped.

2. You can write your own refresh procedure. This technique allows your application program to call the actual procedures that created the drawing. This technique has the advantage that your whole drawing gets redrawn. A program that draws the design presented in this section, and uses a refresh procedure is given in Chapter 5.

4.8. Text Operations

Using the graphics package, a program can mix text characters and graphic images in a single bitmap in a Display Manager frame, an acquired window, the borrowed display, or main memory. The text routines are listed below. (Parameters are not included.)

GPR_$LOAD_FONT_FILE
: Loads a font from a file into the font storage area of display memory. A single program may load multiple fonts and then set them for use, one at a time.

GPR_$UNLOAD_FONT_FILE
: Unloads a font.

GPR_$SET_CHARACTER_WIDTH
: Sets the parameter WIDTH of the specified character in the specified font.

GPR_$INQ_CHARACTER_WIDTH
: Returns the width of the specified character in the specified font.

GPR_$SET_HORIZONTAL_SPACING
: Sets the parameter for the width of spacing between displayed characters for the specified font.

GPR_$INQ_HORIZONTAL_SPACING
: Returns the parameter for the width of spacing between displayed characters for the specified font.

GPR_$INQ_SPACE_SIZE
: Returns the width of the space to be displayed when a character requested is not in the specified font.

GPR_$REPLICATE_FONT
: Creates and loads a modifiable copy of a font.

GPR_$SET_SPACE_SIZE
: Specifies the width of the space to be displayed when a requested character is not in the specified font.

GPR_$SET_TEXT_FONT
Selects a loaded font for use in subsequent text operations.

GPR_$INQ_TEXT
Returns the descriptor of the currently set text font.

GPR_$SET_TEXT_PATH
Specifies the direction in which a line of text is written.

GPR_$INQ_TEXT_PATH
Returns the direction for writing a line of text.

GPR_$SET_TEXT_VALUE
Specifies the pixel value to use for writing text.

GPR_$SET_TEXT_BACKGROUND_VALUE
Specifies the pixel value to use for text background.

GPR_$INQ_TEXT_VALUES
Returns the text and text background pixel values.

GPR_$TEXT Writes text in the current bitmap, beginning at the current position and proceeding in the direction specified by the most recent use of GPR_$SET_TEXT_PATH.

GPR_$INQ_TEXT_EXTENT
Returns the width and height, in pixels, of the area a text string would span if it were written with GPR_$TEXT.

GPR_$INQ_TEXT_OFFSET
Returns the x and y offsets from the top left pixel of a string to be written by GPR_$TEXT to the origin of its first character. This routine also returns the x or y offset to the pixel that is the new current position after the GPR_$TEXT call. This is the y offset when the text path is vertical.

4.9. A Program Using Text

The program presented in this section demonstrates how to load a text font and how to write text into a bitmap.

The program begins by drawing an unfilled square in the bitmap. The top left-hand corner of the square is at the point with coordinates (100,100). The bottom right-hand corner is at (500,500).

The routine GPR_$LOAD_FONT_FILE loads a font into hidden-display memory. The list of fonts is in the directory /sys/dm/fonts.

The routine GPR_$SET_TEXT_FONT establishes the font to be used in all text operations. Notice that font_id is an output parameter in GPR_$LOAD_FONT_FILE and an input parameter in GPR_$SET_TEXT_FONT. As with drawing operations, text operations begin at the current position. To have text begin at the desired location, the current position is moved using GPR_$MOVE.

The routine GPR_$TEXT prints a specified string of text. The maximum length of a text string is 256 characters.

To print vertical text, the program uses the routine GPR_$SET_TEXT_PATH to establish the direction of text written into the bitmap as gpr_$up.

Figure 4-10 shows the output of this program.

```
Program text_on_square;
 %nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %include '/sys/ins/time.ins.pas';
 %list;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;
 var
   init_bitmap_size : gpr_$offset_t;   {size of the initial bitmap}
   init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
   mode        : gpr_$display_mode_t := gpr_$borrow;
   hi_plane_id : gpr_$plane_t := 0;  {highest plane in bitmap}
   delete_display : boolean; {This value is ignored in borrow mode.}
   status         : status_$t; {error code}
   font_id : integer; {identifier of a text font}
   i,j     : integer32;
   direction : gpr_$direction_t; {direction of text}
   pause     : time_$clock_t;
 begin
      init_bitmap_size.x_size := 700;
      init_bitmap_size.y_size := 700;
      gpr_$init(mode,1,init_bitmap_size,hi_plane_id,init_bitmap,status);
      gpr_$draw_box(100,100,500,500,status);

      gpr_$load_font_file('f7x13.b',SIZEOF('f7x13.b'),font_id,status);
      gpr_$set_text_font(font_id,status);

      gpr_$move(110,90,status);

      gpr_$text('This is the top of the rectangle.', 33,status);
      direction := gpr_$up;
      gpr_$set_text_path(direction,status);
      gpr_$move(90, 490,status);
      gpr_$text('This is the side of the rectangle.', 34,status);

      {Keep figure displayed on the screen for five seconds.}
      pause.low32  := five_seconds;
      pause.high16 := 0;
      time_$wait( time_$relative, pause, status);

      gpr_$terminate(delete_display,status);
 end.
```

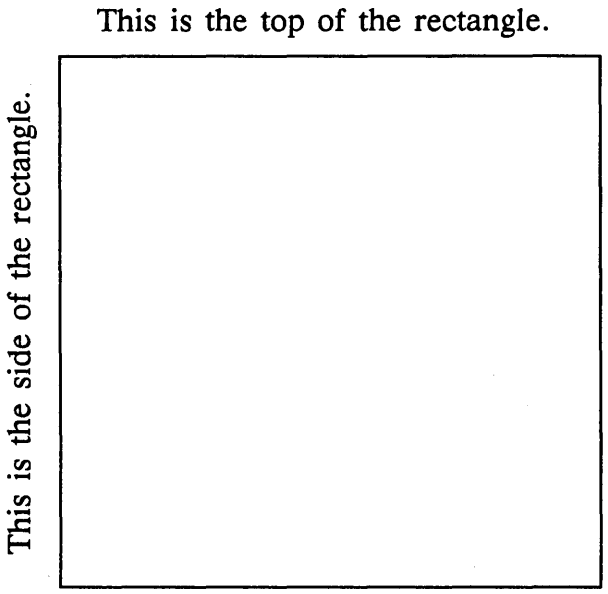


Figure 4-10. Text On A Square

Chapter 5
The Cursor and Input Events

This chapter describes cursor control and input operations. The input routines synchronize program execution around input events. These events include keystroke, mouse or puck buttons, locator and locator stop from mouse or touchpad, and window transition. Some of the information in this chapter refers to attribute blocks, which are discussed in Section 6.5. You may find it helpful to read about attribute blocks before reading this chapter.

5.1. Using Cursor Control

The complete set of cursor routines is available in Borrow-display and Direct mode. In Frame mode, the cursor is controlled by the Display Manager and is always displayed. Therefore, in Frame mode, you can change only the cursor's position. Cursor routines include the following:

GPR_$SET_CURSOR_ACTIVE
: Specifies whether to display the cursor. Initially, the cursor is disabled.

GPR_$SET_CURSOR_PATTERN
: Sets a bitmap pattern as the cursor pattern. This bitmap can be a maximum of 16 x 16 pixels. The initial cursor size varies, depending on the standard font the Display Manager uses.

GPR_$SET_CURSOR_POSITION
: Sets a position on the screen for display of the cursor. The initial cursor position is (0,0). Programs running in frame mode can call this routine.

GPR_$SET_CURSOR_ORIGIN
: Designates one of the cursor's pixels as the cursor origin. Thereafter, when the cursor is moved, the pixel designated as the cursor origin moves to the screen coordinate designated as the cursor position, as shown in Figure 5-1.

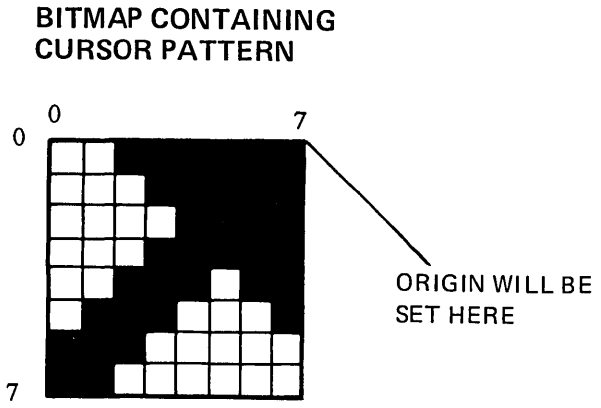
5.2. Implementation Restrictions On The Cursor

When the cursor is active, the cursor pattern is stored in display memory. Therefore, programs that operate in Borrow-display or Direct mode, have the potential to interfere with the cursor pattern and/or to cause the cursor to interfere with a bitmap pattern. To avoid this problem, disable the cursor before performing output procedures to any area of the display in which the cursor could be located.

5.3. Display Mode and Cursor Control

In Borrow-display and Direct mode, the program has complete control over the cursor. In Direct mode, the program-defined cursor pattern and origin are in effect only within the Direct-mode window. As the user moves the cursor between the direct window and other windows on the screen, the system automatically changes the cursor pattern.

If the program executes in frame mode, program control of the cursor is limited. The only cursor control routine that operates in frame mode is GPR_$SET_CURSOR_POSITION, and the program can move the cursor with this routine only if it lies within the frame when GPR_$SET_CURSOR_POSITION is called.




```
.
.
.
GPR_SET_CURSOR_ORIGIN(7,0,STATUS)
GPR_$SET_CURSOR_POSITION(400,400,STATUS)
.
.
.
```

Figure 5-1. Cursor Origin Example

5.4. Using Input Operations

The graphics primitives package includes a set of routines that enable graphics programs to accept input from various input devices. The input routines synchronize program execution around input events. Input routines function in all display modes except GPR_$NO_DISPLAY.

5.4.1. Event Types

An *event* occurs when input is generated in a frame, Direct-mode window, or borrowed display. The GPR package supports several classes of event, called *event types*. Programs use an input routine to select the types of events that should be reported; this operation is called *enabling an event type*. The event types are the following:

Keystroke — A keystroke event occurs when you type specified keyboard characters. Programs can select a subset of keyboard characters, called a keyset, to be recognized as keystroke events. Except in borrow-display mode, keys that do not belong to the keyset are processed normally by the Display Manager. In Borrow-display mode, these keys are ignored.

Button — A button event occurs when you press a button on the mouse or bitpad puck.

Locator — A locator event occurs when you move the mouse or use the touchpad or bitpad to move a locator around the display.

Locator stop — A locator stop event occurs when you stop moving the mouse or stop using the touchpad or bitpad.

Window transition — Except in borrow-display mode, the cursor may move into and out of the window in which GPR input is being performed. When the cursor leaves a window used for graphics display, the input routines report to the program an event of type GPR_$LEFT_WINDOW. When the cursor enters the window, the routines report an event type of GPR_$ENTERED_WINDOW.

Enabled input events are stored in attribute blocks (not with bitmaps) in much the same way as attributes. However, you cannot set and inquire about input events in the same way that you can attributes. You use GPR_$ENABLE_INPUT and GPR_$DISABLE_INPUT instead of GPR_$SET... and GPR_$INQ.... The effect of this difference is the following. *When a program changes attribute blocks for a bitmap during a graphics session, the input events you enabled are lost unless you enable those events for the new attribute block.*

5.4.2. Event Reporting

If an event type is enabled, the input routines report each event of the enabled type to the program with event data and a cursor position. This position is relative to the upper left corner of the window.

If the enabled event type is keystroke or button, the input routines return an ASCII character from the enabled keyset. When defining a keyset for a keystroke event, consult the system insert files /SYS/INS/KBD.INS.PAS, /SYS/INS/KBD.INS.FTN and /SYS/INS/KBD.INS.C. These files contain the definitions for the non-ASCII keyboard keys in the range 128 through 255.

The input routines report mouse button events as ASCII characters. Down transitions range from "a" to "d" (if the mouse has four buttons); up transitions range from "A" to "D". The three mouse keys start with (a/A) on the left side. As with keystroke events, button events can be selectively enabled by specifying a button keyset.

Locator events merely report the x and y coordinates of the locator input. If the program has not

enabled locator events, the GPR software handles any locator data itself by moving the arrow cursor around the window. At the next occurrence of an enabled event, the GPR software reports the locator final cursor position to the program as well as the enabled event.

As noted above, enabled input events are stored in attribute blocks (not with bitmaps) in much the same way as attributes. When a program allocates more than one attribute block, different sets of events are associated with each attribute block. The events enabled for a particular bitmap are the events stored in the attribute block for that bitmap. You must enable the desired events for each window.

GPR_$ENABLE_INPUT and GPR_$DISABLE_INPUT work on the attribute block of the following bitmap: the current bitmap if it is a screen bitmap; otherwise, the screen bitmap that was most current.

When you have more than one bitmap displayed, you can determine the source of input by:

1. Setting a distinct character as a window id with GPR_$SET_WINDOW_ID and making certain that you have enabled entered-window events for all windows. Then remember which window was the last entered. This window is the source of the input event.

2. Using GPR_$INQ_WINDOW_ID after each input event.

5.4.3. Input Routine

The graphics primitives provide the following routines to perform input operations:

GPR_$ENABLE_INPUT
: Enables events of a specific event type. If the event type is keystroke or button, the routine also enables a specific keyset to select which keys or buttons generate input events. Programs must call this routine once for each bitmap.

GPR_$DISABLE_INPUT
: Disables events for the event type previously enabled with GPR_$ENABLE_INPUT.

GPR_$EVENT_WAIT
: Suspends program execution until one of the events enabled by GPR_$ENABLE_INPUT occurs. If the event type is keystroke or button, this routine waits until a member of the specified keyset is input. The information returned includes the type of event that occurred, the character (if any) associated with the event, and the position at which the event occurred. The position will be relative to the upper left corner of the window, or, if the mode is borrow-display, the screen. Position information is not returned in frame mode.

GPR_$COND_EVENT_WAIT
: Performs the same function as GPR_$EVENT_WAIT except that if no event has occurred, the routine returns to the program immediately with an event type that indicates that no event has occurred (GPR_$NO_EVENT).

GPR_$GET_EC

Returns the event count associated with a graphic input event. Programs can use this routine with GPR_$COND_EVENT_WAIT to wait for a combination of system events as well as GPR input events. See the *Programming With General System Calls* for more information on event counts.

GPR_$SET_INPUT_SID

Establishes a selected stream as the standard input stream. The default standard input stream is STREAM_$STDIN. Programs can only use this call in frame mode. In borrow-display and direct modes, input comes directly from the keyboard.

GPR_$SET_WINDOW_ID

Establishes the character that identifies the current bitmap's window. This character is returned by GPR_$EVENT_WAIT and GPR_$COND_EVENT_WAIT when they return GPR_$ENTERED_WINDOW events. The character indicates which window was entered.

GPR_$INQ_WINDOW_ID

Returns the character that identifies the current bitmap's window.

5.5. A Program That Waits For An Event

This program is a modification of program connect_four that was presented in the previous chapter. This version waits for the user to type a character on the keyboard before it exits. Specifically, it waits for a character in the range "a".."d". In addition, this program uses a refresh procedure, which refreshes the drawing in the window if the window is grown or popped. Note that PROCEDURE draw, an external procedure, is used to draw the design initially and any time a refresh is required. *Refresh procedures must always be external.*

The routine GPR_$SET_REFRESH_ENTRY obtains the starting address of the refresh procedure.

The routine GPR_$ENABLE_INPUT defines what type of event is enabled and what keys are enabled. This program enables keyboard input and the set of keys "a".."d".

The routine GPR_$EVENT_WAIT causes the program to wait for one of the enabled events (the user pressing an "a", "b", "c" ,or "d") to occur before terminating.

```
PROGRAM connect_four;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%LIST;


VAR
```

```
    st              : status_$t;              {status code}

    mode            : gpr_$display_mode_t := gpr_$direct;  {gpr mode}

    disp_bm_size    : gpr_$offset_t := [1024, 800];    {size of initial bitmap}
    hi_plane_id     : gpr_$plane_t := 0;  {high plane number of initial bitmap}

    init_bitmap     : gpr_$bitmap_desc_t;  {gpr bitmap descriptor}
    unobscured      : boolean;  {whether window is unobscured on acquisition}

    ev_pos          : gpr_$position_t;      {input event position}
    ev_type         : gpr_$event_t;         {input event type}
    ev_char         : char;                 {input event character}
    keys            : gpr_$keyset_t;        {set of input characters}


PROCEDURE draw (IN unobs : boolean;  IN pos_change : boolean); EXTERN;


BEGIN

        {Initialize GPR.}
    gpr_$init (mode, 1, disp_bm_size, hi_plane_id, init_bitmap, st);

        {Do the graphics output.}
    unobscured := gpr_$acquire_display (st);        {Acquire the display.}
    draw (FALSE, FALSE);                            {Draw the picture.}
    {Establish the refresh procedure.}
    gpr_$set_refresh_entry (addr(draw), nil, st);

        {Wait for user input.}
    keys := ['a'..'d'];                 {Create a key set.}
    gpr_$enable_input (gpr_$keystroke, keys, st);{Enable input for the key set.}
    {Wait for input.}
    unobscured := gpr_$event_wait (ev_type, ev_char, ev_pos, st);


        {Terminate the graphics session.}
    gpr_$terminate(false, st);

END.




MODULE draw;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%LIST;


PROCEDURE draw (IN unobs : boolean;  IN pos_change : boolean);

VAR
```

```
    st              : status_$t;          {status code}
    x1, y1, x2, y2  : integer;            {box corner coordinates}
    rectangle       : gpr_$window_t;      {rectangle to be filled}

BEGIN

        {Set coordinate variables.}
    x1 := 200; x2 := 600;                     {Set dimensions of box.}
    y1 := 200; y2 := 600;
    {Set starting position of 1st rectangle}
    rectangle.window_base.x_coord := 250;
    rectangle.window_base.y_coord := 250;
    rectangle.window_size.x_size := 50;       {Set width of each rectangle.}
    rectangle.window_size.y_size := 50;       {Set height of each rectangle.}

        {Draw outer box and first rectangle.}
    gpr_$draw_box (x1, y1, x2, y2, st);       {Draw an unfilled box.}
    gpr_$rectangle (rectangle, st);           {Draw a filled rectangle.}

        {Draw three more filled rectangles within the unfilled box.}
    rectangle.window_base.x_coord := 500;
    rectangle.window_base.y_coord := 250;
    gpr_$rectangle (rectangle, st);

    rectangle.window_base.x_coord := 250;
    rectangle.window_base.y_coord := 500;
    gpr_$rectangle (rectangle, st);

    rectangle.window_base.x_coord := 500;
    rectangle.window_base.y_coord := 500;
    gpr_$rectangle (rectangle, st);

        {Draw diagonals.}
    gpr_$move (300, 300, st);   {Move the current position.}
    gpr_$line (500, 500, st);   {Draw a line connecting two rectangles.}

    gpr_$move (300, 500, st);
    gpr_$line (500, 300, st);   {Draw a line connecting two rectangles.}

END;
```

Chapter 6
Initial Bitmaps and Attributes

This chapter describes the various types of bitmaps and the attribute blocks associated with them. This chapter also makes refrences to bit block transfers, which are discussed in the next chapter. It may be helpful to read about bit block transfers before reading this chapter.

6.1. Bitmap Structure

As discussed in Chapter 2, DOMAIN displays are raster scan devices. This type of display requires the use of a bitmap to store the intensity values for each pixel in the raster. Monochromatic displays require only one bit of information to be stored for each pixed in the raster. Therefore, bitmaps for monochromatic displays are only one plane deep. Color displays require several bits of information to be stored for each pixel and consequently color bitmaps are composed of several planes. Bitmaps for color displays are discussed in Chapter 8.

6.2. Bitmap Locations

A bitmap may reside in display memory, main memory, hidden display memory, or external storage. The only bitmaps that are visible on the screen are those in display memory. To see the contents of any other bitmap, you must copy it to display memory using a bit-block transfer.

When you initialize GPR using GPR_$INIT, you are allocated an initial bitmap. You determine the location of the initial bitmap when you select the operation mode. The dimensions of the initial bitmap are determined by two things: size, an input parameter in GPR_$INIT; and the type of display you are using.

Initializing GPR in borrow, direct, or frame mode allocates an initial bitmap in display memory. Any graphics operations performed in this bitmap are immediately visible on the screen. If you initialize GPR in no-display mode, the initial bitmap is allocated in main memory, and any graphics operations performed in this bitmap are not visible on the screen.

6.3. Initial Bitmap Size

The size of an initial bitmap is determined by the operation mode, the dimensions you provide, and, in borrow and direct mode, the type of display you are using. The operation mode and dimensions are input parameters in GPR_$INIT.

6.3.1. Initial Bitmap in Borrow Mode

In borrow mode the initial bitmap is allocated in display memory. If you provide bitmap dimensions smaller than the display memory of the node you are using, the size of the bitmap will match the dimensions you provide, and the origin of the bitmap will match the origin of the screen. If, however, you provide dimensions larger than the size of the display memory, the size of the initial bitmap is reduced to match the size of the display memory on your node.

6.3.2. Initial Bitmaps in Frame Mode

In frame mode the initial bitmap is in display memory and you can assign bitmap dimensions up to 4096 x 4096. If you provide dimensions larger than these dimensions, you will be allowed, by default, the maximum size of 4096 x 4096. When you provide dimensions that are smaller than the window, the bitmap is located in the top left-hand corner of the window. (See Figure 6-1 for the relationship between frame, bitmap, and window.)

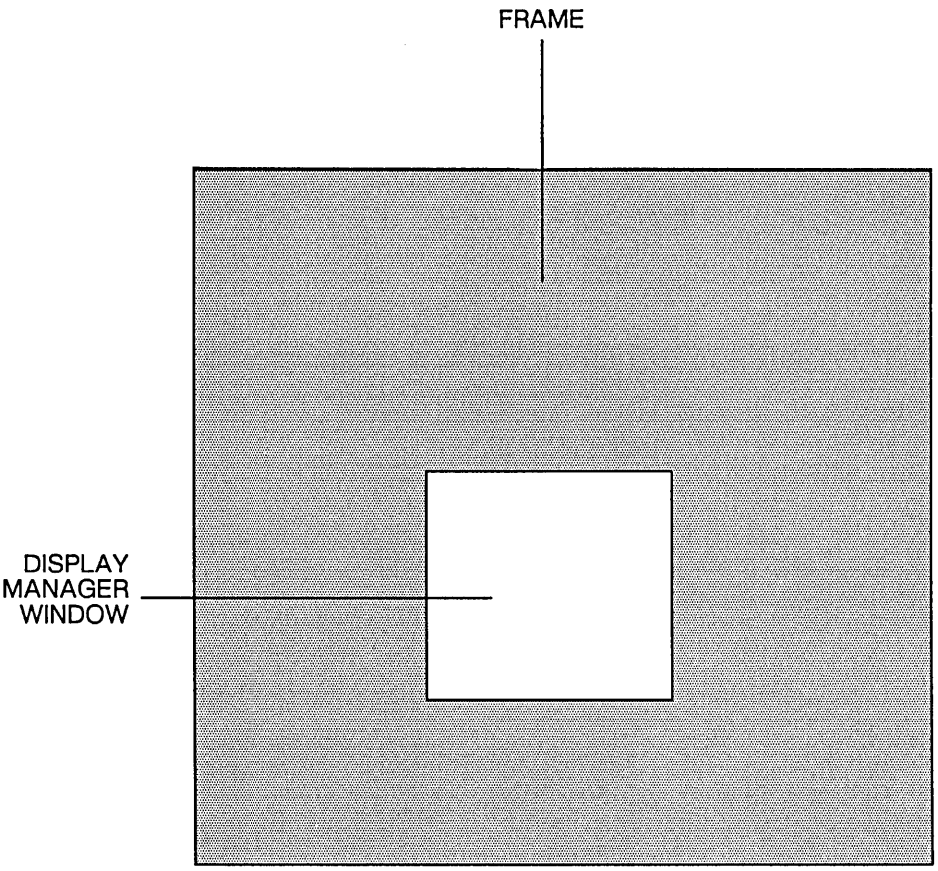


Figure 6-1. Frame Display

6.3.3. Initial Bitmap in Direct Mode

In direct mode the initial bitmap is also in display memory. The size of the bitmap is the size of the display window regardless of the dimensions you provide in GPR_$INIT, unless you provide dimensions smaller than the dimensions of the window. In this case, the bitmap is located in the top left-hand corner of the window.

Bitmaps allocated to the size of a window can be troublesome if the window is smaller than the bitmap you need. If you are using a window that is displayed on the screen before you initialize GPR, you can adjust the size of the window before initializing GPR. Alternatively, you can use PAD_$CREATE_WINDOW to create a transcript pad within a window of a user-defined size, and then initialize GPR using the new pad's stream id as the unit.

6.3.4. Initial Bitmap in No-Display Mode

In no-display mode, the initial bitmap is allocated in main memory, not display memory. Main-memory bitmaps can contain up to eight planes regardless of the display, and can have dimensions up to 8192 x 8192. The contents of main memory bitmaps are not visible on the screen.

6.4. The Current Bitmap

When you initialize GPR, the initial bitmap is also the current bitmap. This bitmap can be in display memory or main memory. All graphics output operations performed take place on the current bitmap. The initial bitmap remains the current bitmap until another is designated to be current using GPR_$SET_BITMAP. Only one bitmap can be current at a time.

6.5. Bitmap Attributes

Each bitmap is associated with a set of attributes identified in an attribute block. These attributes specify the characteristics that operations performed on that bitmap will have. For example, with attributes you can specify that only a certain section of the bitmap be manipulated in any subsequent operations (clipping attribute), that lines be drawn with dashed lines (line style attribute), or that text written on the bitmap be displayed in a specific font (font id attribute). You can change any of the attributes in an attribute block.

6.5.1. The Current Attribute Block

The current bitmap is associated with the current attribute block. When you initialize GPR, the initial bitmap is allocated an attribute block with default settings. This attribute block is the current attribute block and remains so until you change it. If the attribute settings in this block are acceptable, you do not need to concern yourself with the attribute block. If, however, you want to change some of the attributes, you can change them as follows:

- Change them in the current attribute block.
- Allocate a new attribute block, make it current, change the necessary attributes on the new attribute block.

6.5.2. Creating Attribute Blocks

It is possible and often convenient to allocate additional attribute blocks using GPR_$ALLOCATE_ATTRIBUTE_BLOCK. This call establishes a new attribute block with default settings. The form of the call is the following:

```
GPR_$ALLOCATE_ATTRIBUTE_BLOCK(attrib_block_desc, status)
```

Output parameter

attrib_block_desc
: The descriptor of the attribute block. This value is needed to make the attribute block current.

6.5.3. Making an Attribute Block the Current Attribute Block

To make an attribute block the current one that will be associated with the current bitmap, use gpr_$set_attribute_block. The form of the call is the following:

```
GPR_$SET_ATTRIBUTE_BLOCK(attrib_block_desc,status)
```

Input parameter

attrib_block_desc
: The describtor of the attribute block you want to make current. This parameter is an output parameter in GPR_$ALLOCATE_ATTRIBUTE_BLOCK.

6.6. Other Bitmaps

In addition to bitmaps in visible display memory and main memory, there are two other types of bitmaps: external and hidden-display-memory (HDM). The contents of these bitmaps are not visible on the screen. To view them, you must perform a bit-block transfer to display memory.

6.6.1. External Bitmaps

External bitmaps allow you to allocate space on disk in order to store a bitmap for later use. External bitmaps can be treated like any others. Their content, however, is not visible. In this respect, they are similiar to main-memory bitmaps. See Chapter 7 for a sample program that uses external bitmaps.

6.6.2. Hidden-Display-Memory Bitmaps

In either borrow or direct mode, you can allocate a bitmap in HDM using GPR_$ALLOCATE_HDM_BITMAP. The advantage of HDM bitmaps is their location: they are part of display memory, but their contents are not visible. This means that images can be stored in HDM and transferred to visible display memory more quickly than from main memory. The drawback of HDM bitmaps is their size. The largest bitmap can be 224 x 224 pixels.

On DN6XX and DN550 nodes in borrow mode, you can effectively use all of hidden-display-memory by using GPR_$SET_BITMAP_DIMENSIONS and increasing the bitmap size to 1024 x 2048. To use the hidden portion, vary the y coordinate by 1024, and to see the contents of hidden-display memory use either:

1. A bit-block transfer, where the current bitmap is both the source and destination bitmap, and only the x and y offset changes.

2. GPR_$SELECT_COLOR_FRAME to display:

   - Frame 0 : normally visible display
   - Frame 1 : normally hidden display

6.7. Listing of Bitmap Attributes and Bitmap Attribute Default Values

Bitmap attributes, their descriptions, and default values are listed below.

Clipping Window The clipping window attribute specifies a rectangular section of the bitmap, outside which no pixels can be modified. (See Figure 6-2.) After a program calls the routine GPR_$SET_CLIP_WINDOW to specify the dimensions of a clipping window, it may call GPR_$SET_CLIPPING_ACTIVE to enable the new clipping window. Otherwise, the default clipping window remains active.

Default Same size as bitmap. If the program reassigns the attribute block from one bitmap to a smaller bitmap, the clipping window is automatically reduced to the new bitmap size.

NOTE In borrow and frame mode, clipping is disabled by default. In direct mode, it is enabled, and the clip window is set to the size of the window.

Enabling and Disabling clipping has two effects:

1. With clipping enabled, you are restricted to the area of the bitmap which is within the clipping window.

2. With clipping disabled, you are allowed access to the entire bitmap, but some GPR routines, such as GPR_$TRIANGLE, will return an errror status if any of the specified coordinate values are lie outside bitmap limits. Other routines, such as GPR_$LINE, will perform as if clipping were enabled but the clip window covered the entire bitmap.

Coordinate Origin

The coordinate origin specifies a pair of offset values to add to all coordinate positions. These values are subsequently used to calculate offsets for all drawing, text, bit block transfers and move operations on the current bitmap. For example, the coordinate origin affects calls to the routines GPR_$MOVE, GPR_$LINE, and GPR_$PIXEL_BLT.

Default (0,0)

Draw Value The draw pixel value specifies the value to which pixels will be set when drawing lines.

Default 1

Fill Value The fill pixel value specifies the value to which pixels will be set when filling areas.

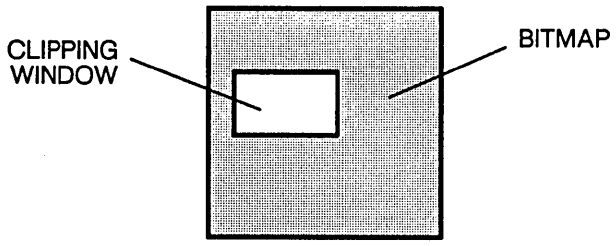


Figure 6-2. Clipping Window On A Bitmap

Default 1

Fill Pattern The fill pattern value specifies the pattern used to fill the current bitmap.

Default Solid.

Text Value The text pixel value specifies the value to which pixels will be set to write text.

Default 1, for borrowed displays, direct mode displays, memory bitmaps, and display manager frames on monochromatic displays; 0, for Display Manager frames on color displays.

Text Background Value
The text background pixel specifies the value to which pixels will be set for text background.

Default -2 (same as bitmap background, which is 0 for borrowed displays, direct mode displays, and memory bitmaps, and the same as the window background for display manager frames).

Text Font The text font attribute specifies the font in which to display text characters in the bitmap.

Default No default. Program must load and set font.

Line Style The line style attribute specifies the style in which to display line segments in the bitmap. Line style can be either solid or dashed; if dashed, the style scale factor determines the length of the dash.

Default Solid line.

Plane Mask The plane mask specifies which planes of a bitmap can be modified by any graphics operation and which planes are protected from modification.

Default — All planes can be modified.

Raster Operation — A raster operation specifies how pixel values are determined in each plane of a destination bitmap for BLT, drawing and text operations. There are sixteen different raster operations that form the set of rules for combining pixel values. Assigning a raster operation code to a bitmap or to a plane of a bitmap alters no values: it specifies how pixel values are determined when BLTs and drawing operations are performed.

For BLTs, the raster operation compares each pixel value within the boundary of the BLT in the source bitmap with each appropriate pixel value in the destination bitmap. The ultimate value of a particular pixel in the destination bitmap is then determined by combining these values using the current raster operation.

For drawing and text operations there is no source bitmap. Destination pixel values are determined as follows. For each pixel included in the drawing or text, the draw value is compared with the value of all pixels affected by the drawing or text operation with the current raster operation.

Default — Op = 3, set all destination bit values to source bit values.

Table 6-1. Raster Operations and Their Functions

Op Code	Logical Function
0	Assign zero to all new destination values.
1	Assign source AND destination to new destination.
2	Assign source AND complement of destination to new destination.
3	Assign all source values to new destination. (Default)
4	Assign complement of source AND destination to new destination.
5	Assign all destination values to new destination.
6	Assign source EXCLUSIVE OR destination to new destination.
7	Assign source OR destination to new destination.
8	Assign complement of source AND complement of destination to new destination.
9	Assign source EQUIVALENCE destination to new destination.
10	Assign complement of destination to new destination.
11	Assign source OR complement of destination to new destination.
12	Assign complement of source to new destination.
13	Assign complement of source OR destination to new destination.
14	Assign complement of source OR complement of destination to new destination.
15	Assign one to all new destination values.

Table 6-2. Raster Operations: Truth Table

Source Bit Value	Destination Bit Value	Resultant Bit Values For The Following Op Codes: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0	0	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0	1	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
1	0	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1	1	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

6.8. Changing Attributes

To change an individual attribute, a program must call one of the attribute-setting routines. These routines change the attributes on the *current* attribute block. To change the attributes on an attribute block which is not current, you must make it current using GPR_$SET_ATTRIBUTE_BLOCK before calling a routine to change an attribute.

The following guidelines may be helpful for using attribute blocks and changing attributes.

If you only have one bitmap, use multiple attribute blocks (if necessary) and use GPR_$SET_ATTRIBUTE_BLOCK to switch between them. If you are only changing one or two attributes, just change the default attribute block as needed.

If you have *multiple bitmaps*, use one attribute block per bitmap. Use GPR_$SET_BITMAP to get the current bitmap and the current attribute block. Then modify the current attribute block as necessary.

The routines for setting attributes are listed below:

GPR_$SET_CLIP_WINDOW
: Changes the clipping window for the current bitmap.

GPR_$SET_CLIPPING_ACTIVE
: Enables/disables a clipping window for the current bitmap.

GPR_$SET_COORDINATE_ORIGIN
: Establishes x- and y-offsets to add to all x and y coordinates used as input for these operations: moving the current position, drawing and text operations, and block transfers.

GPR_$SET_DRAW_VALUE
: Specifies the color/intensity to use to draw lines.

GPR_$SET_FILL_BACKGROUND_VALUE
: Specifies the color/intensity value used for drawing the background of tile fills.

GPR_$SET_FILL_PATTERN
Specifies the fill pattern to use for the current bitmap.

GPR_$SET_FILL_VALUE
Specifies the color/intensity to use to fill rectangles.

GPR_$SET_LINESTYLE
Specifies the line style as solid or dashed.

GPR_$SET_LINE_PATTERN
Establishes the pattern used in drawing lines.

GPR_$SET_PLANE_MASK
Establishes a plane mask that specifies which planes to use for subsequent write operations.

GPR_$SET_RASTER_OP
Specifies a new raster operation for BLTs and lines.

GPR_$SET_TEXT_BACKGROUND_VALUE
Specifies the color/intensity to use for text background.

GPR_$SET_TEXT_FONT
Establishes a new font for subsequent text operations.

GPR_$SET_TEXT_VALUE
Specifies the color/intensity to use for writing text.

6.8.1. Retrieving Attributes

Before you change an attribute, you may want to know the value it currently has. The following routines return attribute values as output parameters:

GPR_$INQ_CONSTRAINTS
Returns the clipping window and plane mask used for the current bitmap.

GPR_$INQ_COORDINATE_ORIGIN
Returns the x and y offsets added to all x and y coordinates used as input to move, line drawing, and BLT operations on the current bitmap.

GPR_$INQ_DRAW_VALUE
Returns the color/intensity value used for drawing lines.

GPR_$INQ_FILL_BACKGROUND_VALUE
Returns the color/intensity value used for drawing the background of tile fills.

GPR_$INQ_FILL_PATTERN
Returns the fill pattern in use for the current bitmap.

GPR_$INQ_FILL_VALUE
Returns the color/intensity value used for filling rectangles.

GPR_$INQ_LINE_PATTERN
Returns the pattern used in drawing lines.

GPR_$INQ_LINESTYLE
Returns information about the current line style.

GPR_$INQ_RASTER_OPS
Returns the raster operations in use for the current bitmap.

GPR_$INQ_TEXT
Returns the text font and text path used for the current bitmap.

GPR_$INQ_TEXT_OFFSET
Returns the x and y offsets from the top left pixel of a string to the origin of the string's first character. This routine also returns the pixel that is the new current position after the text is written with GPR_$TEXT.

GPR_$INQ_TEXT_VALUES
Returns the current values of color/intensity for text and text background in the current bitmap.

6.9. A Program Using Clipping

This program modifies the program in Section 4.6 that draws an "X" across the screen. This version draws the "X" with dashed instead of solid lines. In addition, this program establishes a clipping window with a length and width of 100 pixels. The location of the clipping window is the center of the Display-manager window

The routine, GPR_$SET_CLIP_WINDOW requires that you define the coordinate position of the top left-hand corner of the clipping window, and the length and width of the window.

Two methods are available for changing attributes for a particular bitmap:

1. You can change the attributes on the current bitmap's current attribute block. This may be the most convenient method when you are not changing the same attributes several times within the same program.

2. You can create a new attribute block, associate it with the current bitmap, and change the necessary attributes. This approach initially requires more work, but will save time if you frequently use a particular set of attributes.

This program changes the necessary attributes on the current attribute block since the changes are made only once.

```
PROGRAM clip_an_X;
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%LIST;

var
    st : status_$t;
```

```
    delete_display : boolean;
    disp_bm_size : gpr_$offset_t;
    init_bitmap : linteger;
    i,x,y : integer;
    bm_size : gpr_$offset_t;
    num_of_planes : gpr_$plane_t;
    unobscured : boolean;
    second_attr_block : integer32;
    style : gpr_$linestyle_t;
    scale : integer;
    window : gpr_$window_t;
    mid_x, mid_y : integer;
    mode : gpr_$display_mode_t := gpr_$direct;
    ev_pos : gpr_$position_t;
    ev_type : gpr_$event_t;
    ev_char : char;
    keys : gpr_$keyset_t;  {set of characters}

BEGIN    {Main program}
     {Declare the size of the bitmap you will be using.}
  disp_bm_size.x_size := 1024;
  disp_bm_size.y_size := 1024;
     {Initialize GPR}
  gpr_$init(mode,1, disp_bm_size, 0, init_bitmap, st );
  unobscured := gpr_$acquire_display(st);
  gpr_$inq_bitmap_dimensions(init_bitmap,bm_size,num_of_planes,st);
  x := bm_size.x_size;
  y := bm_size.y_size;

     {Change the attribute for line style in the attribute block associated}
     {with the initial bitmap. Make it dotted.}
   style := gpr_$dotted;
   scale := 5;
   gpr_$set_linestyle(style,scale,st);

   mid_x := (x div 2); { Find the midpoint of one of}
   mid_y := (y div 2); { the lines.}

      {Set the origin of the clipping window.}
   with window.window_base do
      begin
        x_coord := mid_x - 50;
        y_coord := mid_y - 50
      end;
   with window.window_size do {Set the width and height of the }
                              {clipping window.}
      begin
        x_size :=  100;
        y_size :=  100
      end;
      gpr_$set_clip_window(window,st);  {Set clipping active.}

     {Draw the lines. This time, only pixels within the clipping window}
     {will be visible. }
  gpr_$line( x, y, st );  {Draw one line.}
  gpr_$move(x,0,st);  {Move the current position}
  gpr_$line(0,y,st); {Draw the second line.}
  keys := ['a'..'d']; {Create a key set.}
  gpr_$enable_input(gpr_$keystroke, keys, st);
```

```
  unobscured := gpr_$event_wait(ev_type, ev_char, ev_pos, st);
  gpr_$release_display(st);  {Release the display.}
  gpr_$terminate(delete_display,st)  {Terminate GPR}

END.  {Main program}
```

6.10. A Program To Demonstrate Rubberbanding

This program is interactive and demonstrates how to perform rubberbanding. It allows the user to define where a line starts by pressing:

- <F1>
- <F2>
- The left-most mouse button
- The middle mouse button.

It allows the user to rubberband (stretch) a line by moving the cursor either with the mouse or the touch pad. The end of the line is defined when the user presses:

- <F1>
- <F2>
- <F3>
- The left-most mouse button
- The middle mouse button
- The right-most mouse button

The program ends when the user presses either <F3> or the right-most mouse button after a line is drawn.

```
PROGRAM rubberband;

%nolist ;
%include '/sys/ins/base.ins.pas' ;
%include '/sys/ins/gpr.ins.pas' ;
%include '/sys/ins/error.ins.pas' ;
%include '/sys/ins/kbd.ins.pas' ;
%list ;

CONST
  black = 0 ;
  white = 1 ;
VAR
  offset : gpr_$offset_t ;
```

```
  pos : gpr_$position_t ;
  i : integer ;
  b_desc : gpr_$bitmap_desc_t ;
  status : status_$t ;
  size: gpr_$offset_t;
  mouse_buttons : gpr_$keyset_t := ['a', 'b', 'c'];
  pfks : gpr_$keyset_t := [kbd_$f1, kbd_$f2, kbd_$f3];
  null_buttons : gpr_$keyset_t := [ ] ;
  first : boolean ;
  et : gpr_$event_t ;
  ed : char ;
  last, anchor : gpr_$position_t ;
  wait : boolean ;
  rect : gpr_$window_t;
BEGIN
  offset.x_size := 800 ;
  offset.y_size := 800 ;
  gpr_$init (gpr_$borrow, 1, offset, 0, b_desc, status ) ;
  rect.window_base.x_coord := 200;
  rect.window_base.y_coord := 200;
  rect.window_size.x_size := 200;
  rect.window_size.y_size := 200;
  gpr_$rectangle(rect, status);
  {Enable the three mouse buttons.}
  gpr_$enable_input ( gpr_$buttons, mouse_buttons, status ) ;
  {Enable the three function keys.}
  gpr_$enable_input ( gpr_$keystroke, pfks, status );
  REPEAT
     first := true ;
     { Set 'exclusive or' raster op. }
     gpr_$set_raster_op ( 0, 6, status ) ;
     { Wait for the inital mouse key to begin. }
     gpr_$set_cursor_active ( true, status ) ;
     wait := gpr_$event_wait ( et, ed, pos, status ) ;
     gpr_$set_cursor_active ( false, status ) ;
     if ((ed = 'c') or (ed = kbd_$f3)) then exit;
     anchor.x_coord := pos.x_coord ;
     anchor.y_coord := pos.y_coord ;
     gpr_$move ( anchor.x_coord, anchor.y_coord, status ) ;
     gpr_$enable_input ( gpr_$locator, null_buttons, status ) ;

     { Rubberband to the locator position until mouse key. }
     REPEAT
         wait := gpr_$event_wait ( et, ed, pos, status ) ;
         IF et = gpr_$locator
          THEN
           begin
            IF not first THEN
              begin
               gpr_$move ( anchor.x_coord, anchor.y_coord, status ) ;
               gpr_$line ( last.x_coord, last.y_coord, status ) ;
              end
            ELSE
               first := false ;

            gpr_$set_draw_value(white,status);
            gpr_$move ( anchor.x_coord, anchor.y_coord, status ) ;
            gpr_$line ( pos.x_coord, pos.y_coord, status ) ;
            last.x_coord := pos.x_coord ;
```

```
            last.y_coord := pos.y_coord ;
          end ; { if locator }

      UNTIL ((et = gpr_$buttons) or (et = gpr_$keystroke));

      { Now really draw the line with normal a raster_op. }
      gpr_$set_raster_op ( 0, 3, status);
      gpr_$move(anchor.x_coord, anchor.y_coord, status);
      gpr_$line ( last.x_coord, last.y_coord, status ) ;
      gpr_$disable_input ( gpr_$locator, status ) ;
  UNTIL false;

  gpr_$terminate ( false, status ) ;

END.
```

Chapter 7
Bitmaps and Bit Block Transfers

This chapter discusses bitmaps outside display memory. It demonstrates how to use bit-block transfers to copy information from one bitmap to another or from one location to another location in the same bitmap.

7.1. Bitmaps In Main-memory, Hidden-display Memory and On Disk

For some graphics applications, it is necessary or convenient to establish bitmaps in locations other than display memory. These bitmaps are used just like bitmaps in display memory except that nothing appears on the screen. Therefore, these bitmaps can be used as scratch areas or as areas to save images. To display images from any of these bitmaps, you must use a bit-block-transfer operation (BLT) to transfer information to a bitmap in visual display memory.

7.1.1. Allocating Bitmaps In Main Memory

Use GPR_$ALLOCATE_BITMAP to allocate a bitmap in main memory. The form of the call is the following:

```
GPR_$ALLOCATE_BITMAP(size,hi_plane_id,attrib_block_desc,bitmap_desc,status)
```

Input Parameters

size — The dimension of the memory bitmap. Main Memory bitmaps can have dimensions up to 8192 x 8192.

hi_plane_id — The number of the highest plane in the bitmap. Bitmap planes are numbered from 0 - 7.

attrib_block_desc — The descriptor of the attribute block that the new main-memory bitmap will use. This can be the current attribute block, or you can can designate an already existing block which is not current. You can create a new attribute block if necessary and use its descriptor.

Output Parameters

bitmap_desc — The descriptor of the new main-memory bitmap.

7.1.2. Making Main-memory Bitmaps Current

You have a bitmap allocated in memory, but it is not the current bitmap. You can make it the current by using the call:

```
GPR_$SET_BITMAP(bitmap_desc,status)
```

When you use this call, the attribute block associated with the bitmap becomes the current attribute block (see attrib_block_desc in section 7.1.1). This call makes a bitmap current, not visible.

7.2. Hidden-display-memory Bitmaps

In borrow-mode and direct-mode you can allocate additional bitmaps in hidden display memory using GPR_$ALLOCATE_HDM_BITMAP. The parameters for this call are identical to the parameters in GPR_$ALLOCATE_BITMAP.

The following restrictions apply to hidden-display-memory bitmaps:

- The maximum size allowed for a HDM bitmap is 224 x 224.
- In direct mode, if your program releases the display and another process acquires the display (this could be the Display Manager or another program) before the original program reacquires the display, the contents of hidden-display memory may be written over.

7.3. External Bitmaps

To save a graphic image for use at a latter time, you must store it on disk using GPR_$OPEN_BITMAP_FILE. The form of the call is the following:

```
GPR_$OPEN_BITMAP_FILE(access,filename,name_size,version,size
                      groups,group_headers,attribs,bitmap,created,status)
```

Input Parameters

access — Specifies how you are going to use the file. There are four possible values:

gpr_$create	Allocates a new file on disk for storage of a graphic image.
gpr_$update	Allows you to modify a previously created file, or create a new one.
gpr_$write	Allows you to write to an existing file.
gpr_$readonly	Permits you to read a previously created file.

filename — The pathname of the bitmap file.

Input or Output Parameters

The following parameters can be input or output parameters depending on the value of access and whether or not the file exists. See Table 7-1.

version — The number on the header of the external bitmap file.

size	Contains the dimensions of the bitmap.
groups	The number of groups in external bitmaps. Currently, groups are not used; this value must be 1.
group_headers	The descriptors of the external bitmap group headers.
attribs	The descriptor of the attribute block to be used by this bitmap.

Output Parameters

bitmap	The descriptor of the bitmap on disk.
created	A boolean value which specifies whether the bitmap file was created.

Table 7-1. GPR_$OPEN_BITMAP_FILE Access Table

	GPR_$CREATE	GPR_$UPDATE (file exists: no)	GPR_$UPDATE (file exists: yes)	GPR_$WRITE	GPR_$READONLY
version, size, groups, group-headers	IN	IN	OUT	OUT	OUT

7.4. Using Blts With External Bitmaps and Hidden-display Memory

If you have a bitmap in main memory, hidden-display memory, or on disk, and you want to make all or part of it visible, use a bit-block transfer to a display memory bitmap. A BLT copies a rectangle of pixels from one bitmap to another, or from one place in a bitmap to another place in the same bitmap.

Bit-block transfers (sometimes called transfers) can be made from one bitmap to another or within the same bitmap. When a BLT is performed from one bitmap to another, the bitmap that contains the information being transferred is the source bitmap, and the bitmap that receives the information is the destination bitmap. When a BLT is done within the same bitmap, this bitmap serves as both the source and destination bitmap.

The three BLT subroutines are listed below. With each call, there is a description of its parameters. Identical parameter descriptions are not repeated for each call.

```
GPR_$PIXEL_BLT(source_bitmap_desc,source_window,dest_origin,status)
```

This routine copies a rectangle from all planes of the source bitmap to the corresponding planes of destination bitmap.

Input Parameters

source_bitmap_desc
: The bitmap descriptor of the source bitmap. The current bitmap is the destination.

source_window
: The coordinates of the rectangle to be moved. This is the same format as window in GPR_$SET_CLIP_WINDOW.

dest_origin
: The coordinates of the upper left corner of the rectangle in the destination bitmap.

```
GPR_$BIT_BLT(source_bitmap_desc,source_window,source_plane,
             dest_origin,dest_plane,status)
```

This routine copies a rectangle from one plane of the source bitmap to one plane of the destination bitmap. You specify the planes.

Input Parameters

source_plane
: The plane id of the plane from which the rectangle is to be moved.

dest_plane
: The plane id of the plane to which the rectangle is to be moved.

```
GPR_$ADDITIVE_BLT(source_bitmap_desc,source_window,source_plane,
                  dest_origin,status)
```

This routine copies a rectangle from one plane of the source bitmap (you specify the plane) to all planes of the destination bitmap.

Input Parameters

source_plane
: The plane id of the plane from which the rectangle is to be moved.

7.4.1. Using a Plane Mask With a BLT

A program can mask planes of a bitmap to establish the following:

- Destination planes of a pixel BLT operation
- Destination planes of an additive BLT operation.

For plane masking procedures, see the routine GPR_$SET_PLANE_MASK.

7.4.2. Using Raster Operations With a BLT

When a program invokes a BLT with the default raster operation, the BLT moves the rectangle and retains all bit values. When the program uses a BLT with any other raster operation, the BLT combines two rectangles and assigns the resultant bit values according to the raster operation of the destination bitmap.

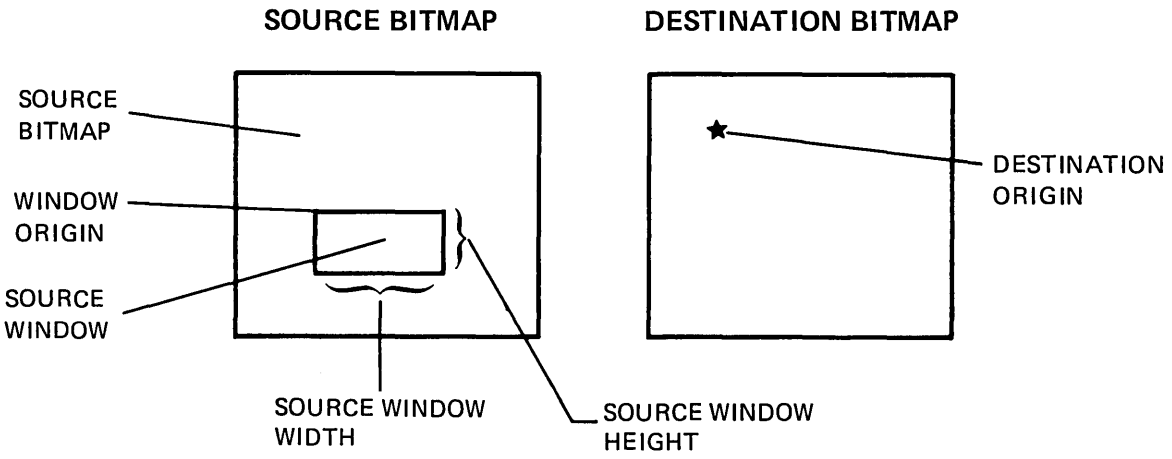


Figure 7-1. Information Required for Graphics BLT

7.4.3. Example of a BLT Operation

In a BLT operation, bits are transferred only on the rectangular area in which the source bitmap, source window, and destination clipping window intersect (see Figure 7-2). Nothing is transferred outside the bounds of the bitmap. For example, if the clipping window of the current bitmap (the destination bitmap) excludes part of the destination rectangle that would otherwise receive pixels, the size of the actual rectangle moved will be smaller than the source window. Similarly, if the source window overflows the boundaries of the source bitmap, the size of the actual rectangle moved will be smaller than the source window.

7.5. Example of A Blt With A Raster Operation

Figure 7-3 shows a source bitmap in main memory, a destination bitmap in display memory, and the bitmap created by using a BLT with raster operation 1, the logical "AND" function. The figure shows 0 bits as black, and 1 bits as white.

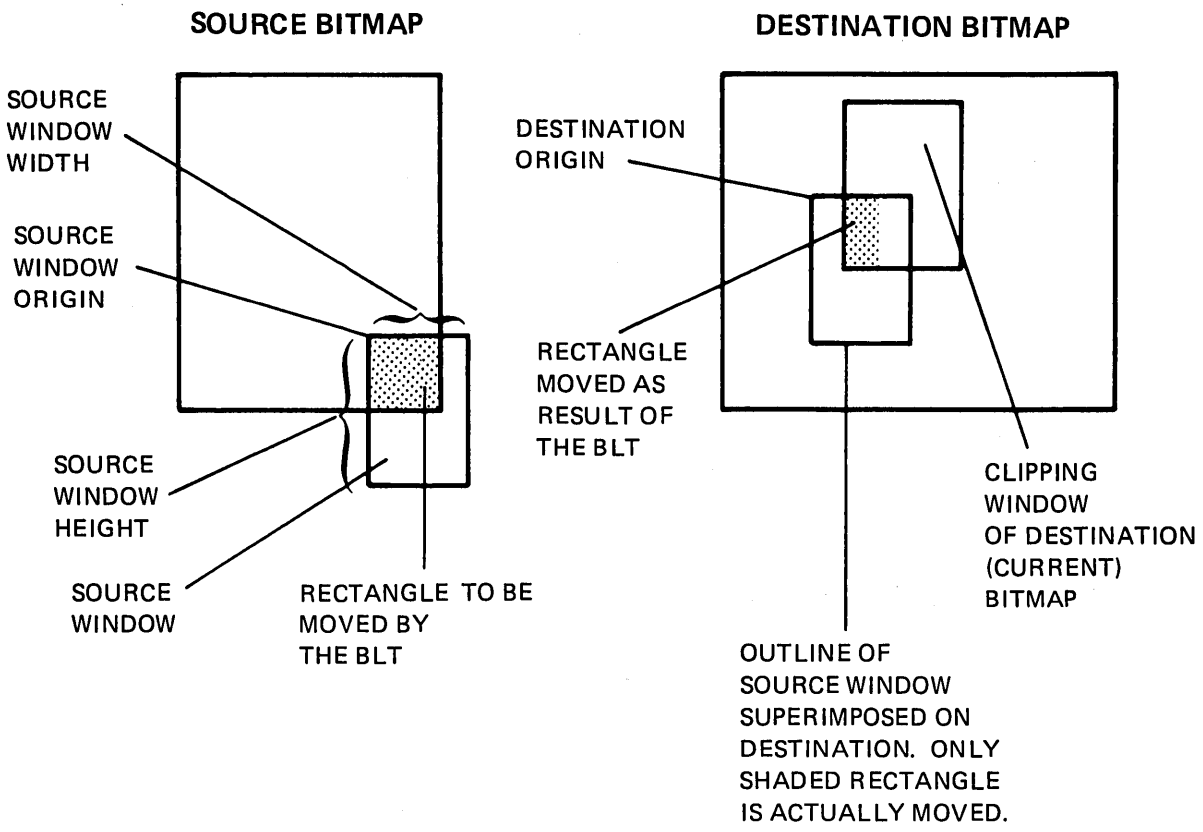


Figure 7-2. BLT Example: Intersection of Source Bitmap, Source Window, Destination Clipping Window

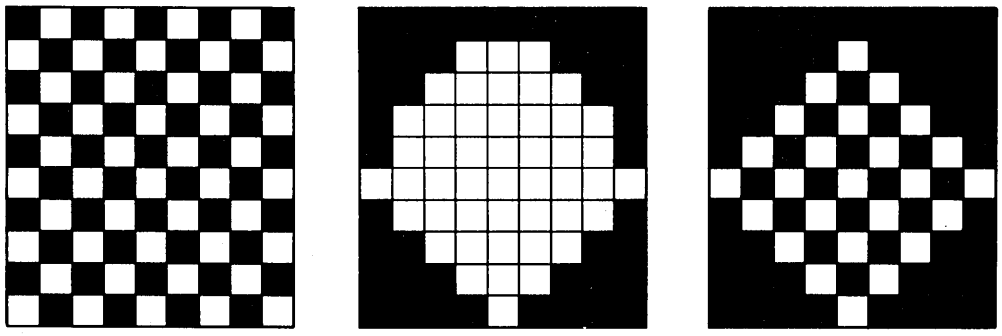


Figure 7-3. Example of BLT with Raster Op Code = 1 (Logical "AND")

7.6. A Program To Draw A Checker Board

The program presented in this section draws the checker board in Figure 7-4.

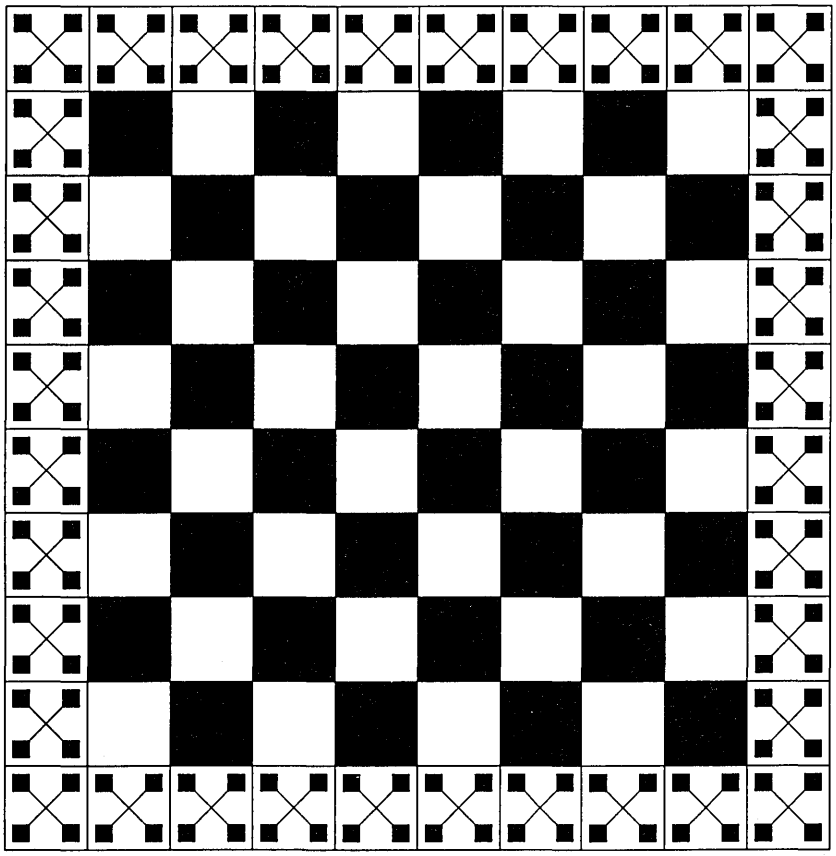


Figure 7-4. Checker Board with Border

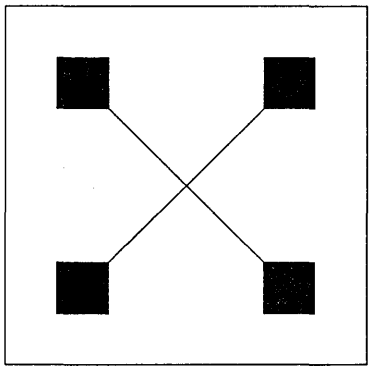


Figure 7-5. Border Design

The problem of drawing this design can be approached in several ways. In order to demonstrate BLT operations and the various types of bitmaps, the program uses the following approach:

1. Draws and stores the checker board in a bitmap file.

2. Draws and stores the design in Figure 7-5 in hidden-display memory.

3. BLTs the design in Figure 7-5 into a display bitmap and create the border.

4. BLTs the checker board from the file on disk into the border in display memory.

Each of the steps above is performed in a separate procedure to facilitate explanations. The program Checker_with_Border is listed first. Following this, each procedure is listed in a separate subsection. All variables are global and are listed in the main program.

To create a checker board on an external bitmap, the program creates an external file using GPR_$OPEN_BITMAP_FILE. After the program sets the external bitmap current, the checker board is created in procedure CHECK_ON_DISK.

Once the checker board is drawn in the external bitmap, the program draws one square of the border design in hidden display memory. This is performed in the procedure DRAW_DESIGN.

With one square of the design drawn in hidden display memory, the remainder of the border design is drawn in visible display by transferring the design several times. This is done in procedure BLT_BORDER. Notice that the current bitmap is both the source and the destination.

The program finishes by transferring the checker board stored on disk into the border. This is done in the procedure BLT_CHECKER_TO_BORDER.

```
program checker_with_border;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%LIST;

 var
    init_bitmap :   gpr_$bitmap_desc_t; {bitmap descriptor}
    hdm_bitmap  : gpr_$bitmap_desc_t;    {bitmap descriptor}
    i,j     :   integer;
    source :   gpr_$window_t;
    dest_pos :   gpr_$position_t;
    ev_type  :   gpr_$event_t;
    event_data,ev_char :   char;
    corner_1, corner_2 :   gpr_$position_t;
    st    :   status_$t;
    keys :   gpr_$keyset_t;  {set of characters}
    mode :   gpr_$display_mode_t;
    x,x1,y,y1     :   integer;
    disp_bm_size :   gpr_$offset_t; {size of initial bitmap}
    filename      :   array[1..256] of char;
    name_size :   integer;
```

```
  version    :   gpr_$version_t;
  size       :   gpr_$offset_t;
  groups     :   integer;
  group_headers :   gpr_$bmf_group_header_array_t;
  header     :   gpr_$bmf_group_header_array_t;
  disk_bitmap :   gpr_$bitmap_desc_t;
  created :   boolean;
  attribs :   gpr_$attribute_desc_t;
  hdm_attr_block :   gpr_$attribute_desc_t;
  hi_plane :   gpr_$plane_t := 0;
  hdm_attr_blk_desc :   gpr_$attribute_desc_t;


BEGIN

 disp_bm_size.x_size := 1024;   {width of bitmap for display}
 disp_bm_size.y_size := 800;   {height of bitmap for display}

 {initialize graphics primitives}
 gpr_$init( gpr_$BORROW, 1, disp_bm_size, hi_plane, init_bitmap, st );

 name_size := 7;  {number of characters in the pathname}
 groups := 1;
 header[0].n_sects := hi_plane + 1;
 header[0].pixel_size := 1;
 header[0].allocated_size := 0;
 header[0].bytes_per_line := 0;
 header[0].bytes_per_sect := 0;

 {Declare the size of the external bitmap.}
 size.x_size := 500;
 size.y_size := 500;

 gpr_$allocate_attribute_block(attribs,st);
 gpr_$open_bitmap_file(gpr_$create,'checker',name_size,version,size,
                       groups,header,attribs,disk_bitmap,created,st);

 gpr_$set_bitmap(disk_bitmap,st); {Set the external bitmap current.}

 CHECK_ON_DISK; {Procedure to draw a checker board in an external bitmap}

 gpr_$allocate_attribute_block(hdm_attr_blk_desc,st);
 size.x_size := 224;   {width of bitmap for hdm}
 size.y_size := 224;   {height of bitmap for hdm}
 gpr_$allocate_hdm_bitmap(size,0,hdm_attr_blk_desc,hdm_bitmap,st);

 gpr_$set_bitmap(hdm_bitmap,st); {Make the HDM bitmap current.}

 gpr_$clear(0,st); {Clear the HDM bitmap.}

 DRAW_DESIGN; {Procedure to draw one rectangle of the border design.}

 gpr_$set_bitmap(init_bitmap,st); {Make display bitmap current.}

 BLT_BORDER;  {Procedure to create the border on the screen.}

 BLT_CHECKER_TO_BORDER; {Procedure to BLT the checker board from}
                        {the external bitmap into the border which}
                        {is displayed on the screen.}
```

```
  {Typing a lower-case a - d will end the program.}
  keys := ['a'..'d'];
  GPR_$ENABLE_INPUT(GPR_$KEYSTROKE, KEYS, ST);
  UNOBSCURED := GPR_$EVENT_WAIT(EV_TYPE, EV_CHAR, dest_POS, ST)
  gpr_$terminate( false, st )

end.
```

7.6.1. Procedure check_on_disk

This procedure creates a checker board with dimensions of 400 x 400. An unfilled box is drawn as a border. Following this, two rows of the board are drawn and then these two rows are transferred three times to complete the entire board. Notice that the external bitmap is used as both the source and the destination of the BLT.

```
Procedure check_on_disk;

 BEGIN

  x := 0; x1 := 400; y := 0; y1 := 400; {dimensions of box}
  gpr_$draw_box(x,y,x1,y1,st);

  x := 0; y := 0;
  {Define the dimensions of the rectangle to be drawn.}
  source.window_base.x_coord := x; {x coord. of rectangle}
  source.window_base.y_coord := y; {y coord. of rectangle}
  source.window_size.x_size := 50; {rectangle width}
  source.window_size.y_size := 50; {rectangle height}

  {This loop will draw two rows of rectangles. There are}
  {eight rectangles per row but only four of them need be}
  {drawn since four are filled and four are empty.}

  for j := 1 to 2 do {Draw two rows of rectangles.}
   begin
     for i := 1 to 4 do {Draw four rectangles 50  pixels apart.}
       begin
         source.window_base.x_coord := x;
         gpr_$rectangle(source,st);
         x := x + 100;
       end;
      y := y + 50; {Go to next row.}
      x := 50; {Move over one position.}
      source.window_base.y_coord := y;
   end;

   {BLT the two rows of rectangles three times to get}
   {eight rows. This is a BLT with the current bitmap}
   {serving as both the source and the destination.}

   {Define the area for the BLT.}
     source.window_base.y_coord := 0;
     source.window_base.x_coord := 0;
```

```
    source.window_size.x_size := 400;
    source.window_size.y_size := 100;

  dest_pos.x_coord := 0; {Define x coordinate for the destination.}
                         {of the BLT.}
  y := 100;
  for i := 1 to 3 do  {BLT figure 3 times.}
   begin
    dest_pos.y_coord := y;  {Define y coordinate for the destination.}
                            {of the BLT.}
    gpr_$pixel_blt(disk_bitmap,source,dest_pos,st);
    y := y + 100; {Increment y value of destination.}
   end;


END;
```

7.6.2. Procedure draw_design

This procedure draws one square of the border design in hidden display. It is similiar to program Connect_Four in Chapter 4.

```
Procedure draw_design;

begin
  x := 0; x1 := 49; y := 100; y1 := 149; {coordinates of box}
  gpr_$draw_box(x,y,x1,y1,st);

  {Draw filled rectangles.}
  rectangle.window_base.x_coord := 10;
  rectangle.window_base.y_coord := 110;
  rectangle.window_size.x_size := 10;
  rectangle.window_size.y_size := 10;
  gpr_$rectangle(rectangle,st);

  rectangle.window_base.x_coord := 30;
  rectangle.window_base.y_coord := 110;
  gpr_$rectangle(rectangle,st);

  rectangle.window_base.x_coord := 10;
  rectangle.window_base.y_coord := 130;
  gpr_$rectangle(rectangle,st);

  rectangle.window_base.x_coord := 30;
  rectangle.window_base.y_coord := 130;
  gpr_$rectangle(rectangle,st);

  {Draw connecting lines.}
  gpr_$move(20,120,st);
  gpr_$line(30,130,st);

  gpr_$move(20,130,st);
  gpr_$line(30,120,st);
 end;
```

7.6.3. Procedure blt_border

This procedure transfers the design stored in hidden display to visible display memory. First the design is transferred ten times vertically and then nine times horizontally. This creates one corner of the border. Two more transfers are done in the procedure. These transfers take advantage of the work already done and BLT a whole row and a whole column of squares. The display bitmap is used as both the source and the destination bitmap.

```
Procedure blt_border;

BEGIN

   {Define the area for the BLT.}
   source.window_base.x_coord := 0;
   source.window_base.y_coord := 100;
   source.window_size.x_size := 50;
   source.window_size.y_size := 50;

   dest_pos.x_coord := 290; {Define x coordinate for the destination.}
                       {of the BLT.}
   y := 290;
   for i := 1 to 10 do  {BLT figure vertically 10 times.}
    begin
     dest_pos.y_coord := y;  {Define y coordinate for the destination.}
                          {of the BLT.}
     gpr_$pixel_blt(hdm_bitmap,source,dest_pos,st);
     y := y + 50; {Increment y value of destination.}
    end;

   dest_pos.y_coord := 740; {Define x coordinate for the destination.}
                       {of the BLT.}
   x := 340;
   for i := 1 to 9 do  {BLT figure 9 times horizontally.}
    begin
     dest_pos.x_coord := x;  {Define y coordinate for the destination.}
                          {of the BLT.}
     gpr_$pixel_blt(hdm_bitmap,source,dest_pos,st);
     x := x + 50; {Increment x value of destination.}
    end;

   {Define the area for the BLT to the top row.}
     source.window_base.x_coord := 340;
     source.window_base.y_coord := 740;
     source.window_size.x_size := 450;
     source.window_size.y_size := 50;

    {Define destination coordinates for BLT to the right-hand column.}
     dest_pos.x_coord := 340;
     dest_pos.y_coord := 290;
```

```
    gpr_$pixel_blt(init_bitmap,source,dest_pos,st);
          •

  {Define the area for the BLT.}
    source.window_base.y_coord := 340;
    source.window_base.x_coord := 290;
    source.window_size.x_size := 50;
    source.window_size.y_size := 400;

    dest_pos.x_coord := 740;
    dest_pos.y_coord := 340;
    gpr_$pixel_blt(init_bitmap,source,dest_pos,st);

END;
```

7.6.4. Procedure blt_checker_to_border

This procedure BLTs the checker board stored on disk into the border in display memory. The source bitmap is the disk bitmap (disk_bitmap) and the destination bitmap is the display bitmap (init_bitmap).

```
Procedure blt_checker_to_border;

Begin

   {Define the area for the BLT.}
   source.window_base.y_coord := 0;
   source.window_base.x_coord := 0;
   source.window_size.x_size := 400;
   source.window_size.y_size := 400;

   {Define the origin for the destination of the BLT.}
   dest_pos.x_coord := 340; dest_pos.y_coord :=340;
   gpr_$pixel_blt(disk_bitmap,source,dest_pos,st);

End;
```

Chapter 8
Color Graphics

This chapter describes color configurations, formats, color maps, and the operation modes for color graphics. The information presented in this chapter builds upon the basic information presented in Chapter 2.

8.1. Display Configurations

Similiar to monochrome displays, color displays are bit-mapped, raster-scan devices. Two hardware configurations are available for this device.

- Two-board configuration, which has four planes.
- Three-board configuration, which has eight planes.

Within each of these configurations two formats are available.

- Interactive format
- Imaging format.

The formats are user-defined and specify the number of colors that can be displayed simultaneously (see Figures 8-1 and 8-2). Imaging formats restrict GPR operations by decreasing the number of calls that can be used, and with 24-bit imaging, screen resolution is reduced.

Table 8-1. Two-Board Configuration for Color Display

Format	Pixel Dimensions		
	Visible Display	Hidden Display	Number of Colors
DN6xx 4-bit interactive (Default)	1024 x 1024	1024 x 1024	16
8-bit imaging	1024 x 1024	none	256
DN550 4-bit interactive (Default)	1024 x 800	1024 x 1024 plus 1024 x 224	16
8-bit imaging	1024 x 800	1024 x 224	256

Table 8-2. Three-Board Configuration for Color Display

Format	Pixel Dimensions		
	Visible Display	Hidden Display	Number of Colors
DN6xx 8-bit interactive (Default)	1024 x 1024	1024 x 1024	256
24-bit imaging	512 x 512	512 x 512	16.7 million
DN550 8-bit interactive (Default)	1024 x 800	1024 x 1024 plus 1024 x 224	256
24-bit imaging	512 x 400	512 x 512 plus 512 x 112	16.7 million

8.1.1. Two-Board Configuration

The interactive 4-bit pixel format is the default for a two-board configuration. This means that four bits are used to assign a pixel value (color map index) to each pixel. This format allows sixteen different colors to appear on the screen at one time. On a DN6xx graphics processor, the pixels are arranged 1024 x 1024 in visible display memory, and 1024 x 1024 in hidden-display. On a DN550 graphics processor, there are two sections of display memory. Each section has the dimensions of 1024 x 1024. Each section is divided into two subsections: one subsection has the dimensions 1024 x 800 pixels and is viewable, the other subsection has the dimensions 1024 x 224 and is hidden. You can map only one section to the display at a time. Interactive formats support all GPR operations.

Optionally, software can change a two-board configuration to an 8-bit imaging format, with eight bits used to assign a pixel value (color map index) to each pixel. This format allows 256 colors to appear on the screen at one time. The pixels are arranged 1024 x 1024 in the display memory. Using a DN6xx graphics processor, you can view the entire display memory. Using a DN550 graphics processor, you can view only 1024 x 800 of the display memory. Hidden-display memory is not available with imaging formats on a two-board configuration. Imaging formats support only limited GPR operations.

8.1.2. Three-Board Configuration

The interactive 8-bit pixel format is the default for a three-board configuration. This means that eight bits are used to assign a pixel value (color map index) to each pixel. This format allows 256 different colors to appear on the screen at one time. On a DN6xx graphics processor, the pixels are arranged 1024 x 1024 in visible display memory, and 1024 x 1024 in hidden-display. On a

DN550 graphics processor, there are two sections of display memory. Each section has the dimensions of 1024 x 1024. Each section is divided into two subsections: one subsection has the dimensions 1024 x 800 pixels and is viewable; the other subsection has the dimensions 1024 x 224 and is hidden. You can map only one section to the display at a time.

Interactive formats support all GPR operations. The 8-bit interactive format is compatible with the 4-bit interactive format. For example, the Display Manager uses four planes, but runs on a configuration using eight planes. In general, the operations performed in 4-bit format can also be performed in 8-bit format.

Optionally, software can change a three-board configuration to a 24-bit imaging format. This means that 24 bits are used to assign a pixel value (color map index) to each pixel, making it possible to use over 16 million different colors. On a DN6xx graphics processor, the pixels are arranged with 512 x 512 in visible display and 512 x 512 in hidden display. On a DN550 graphics processor, the pixels are arranged 512 x 400 in visible display, 512 x 400 in hidden display. Imaging formats support only limited GPR operations.

8.2. Displaying Colors On The Screen

The color of every pixel in a raster is determined by interpreting its pixel value. A pixel value is a number that is used as an index into a color map where color values are stored. For drawing operations, the pixel value is called the draw value, and for fill operations, it is called the fill value. For text operations, it is the text value.

For monochrome displays, a pixel value can be represented by one bit, and a color map has only two entries. A pixel value of 0 can represent white, and 1 can represent black or vice-versa For color displays, pixel values must be represented by several bits, and the color map must have several entries.

Colors are displayed on the screen as follows. Each pixel value is used as an index into the color map to determine the correct color value. The intensity level for each primary color is then calculated from the color value and sent to the appropriate color gun which illuminates the pixel with the proper amount of light. Figure 8-1 illustrates the relationship between pixel values and the color map.

8.2.1. The Color Map: A Set of Color Values

A color map is a set of indexed color values. Each color value is a 4-byte integer that uses eight bits to represent the intensity of each of the three primary colors of this graphics package (red, green and blue). The remaining eight bits are ignored. Figure 8-2 shows the format of a color value.

The eight bits used to represent the intensity of each primary color provide 256 levels. Intensity level 0 means none of that color, while intensity level 255 represents full intensity. Combinations of the 256 levels of each of the three primary colors provide over 16 million colors.

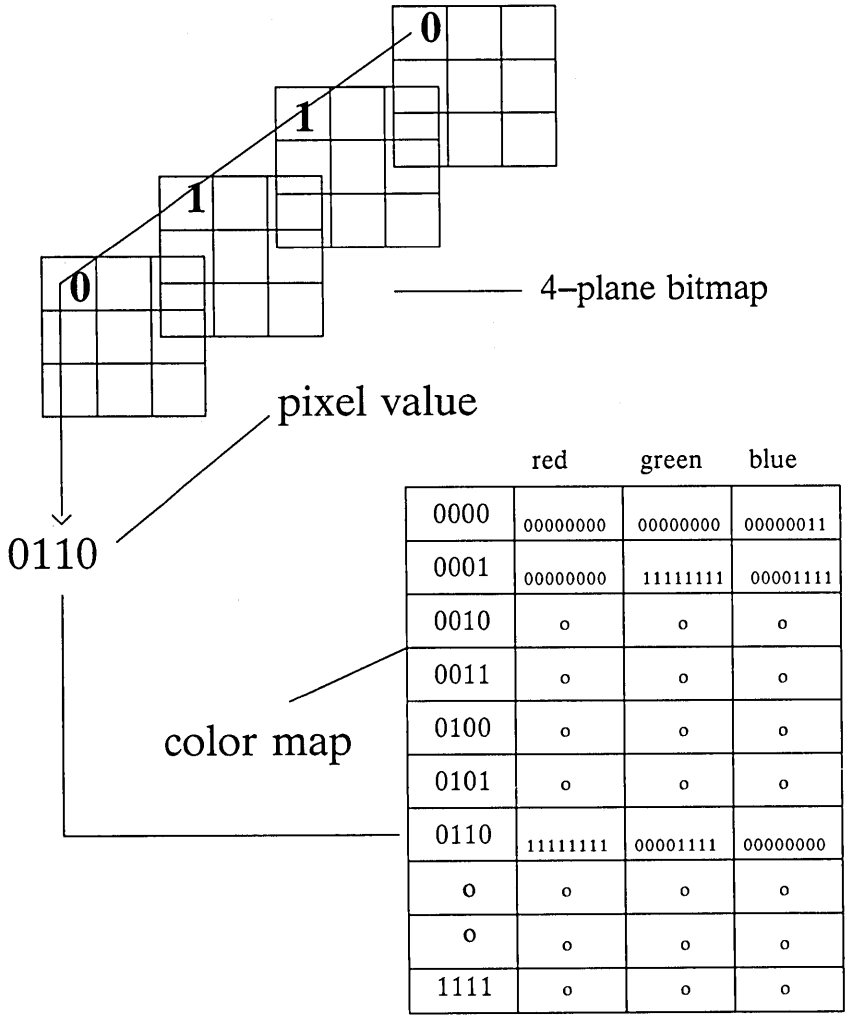


Figure 8-1. Four Plane Color System

Bit position	31 24	23 16	15 8	7 0
	Ignored	Red Component	Green Component	Blue Component

Figure 8-2. Color Value Structure

8.2.2. The Size of a Color Map

The size of a color map is determined by the machine configuration and the color format being used. These two taken together specify how many bits represent each pixel. For example, with a 2-board configuration using interactive format, each pixel is represented by four bits. Four bits allow 16 different combinations, thus the color map can contain at most 16 colors. You can fill the 16 places in the color map with any colors you wish, but you cannot have more than 16 colors.

8.2.3. Color Map for Color Displays: 4-Bit and 8-Bit Formats

For a color display in the 4-bit pixel format, the color map has 16 entries, with index values 0-15. For a color display in the 8-bit pixel format, the color map has 256 entries, with index values 0-255. In both formats, all entries are set to default values at the initialization of the graphics primitives package. After initialization, either of these maps can be changed to contain any colors from the over 16 million available colors.

The color map can be thought of as a one-dimensional array of 4-byte integers. The pixel values can be thought of as the indices to that array. To set the color of a pixel, you assign it the value of the index in the array that contains the color value you want.

8.2.4. Color Map for Color Displays: 24-Bit Imaging

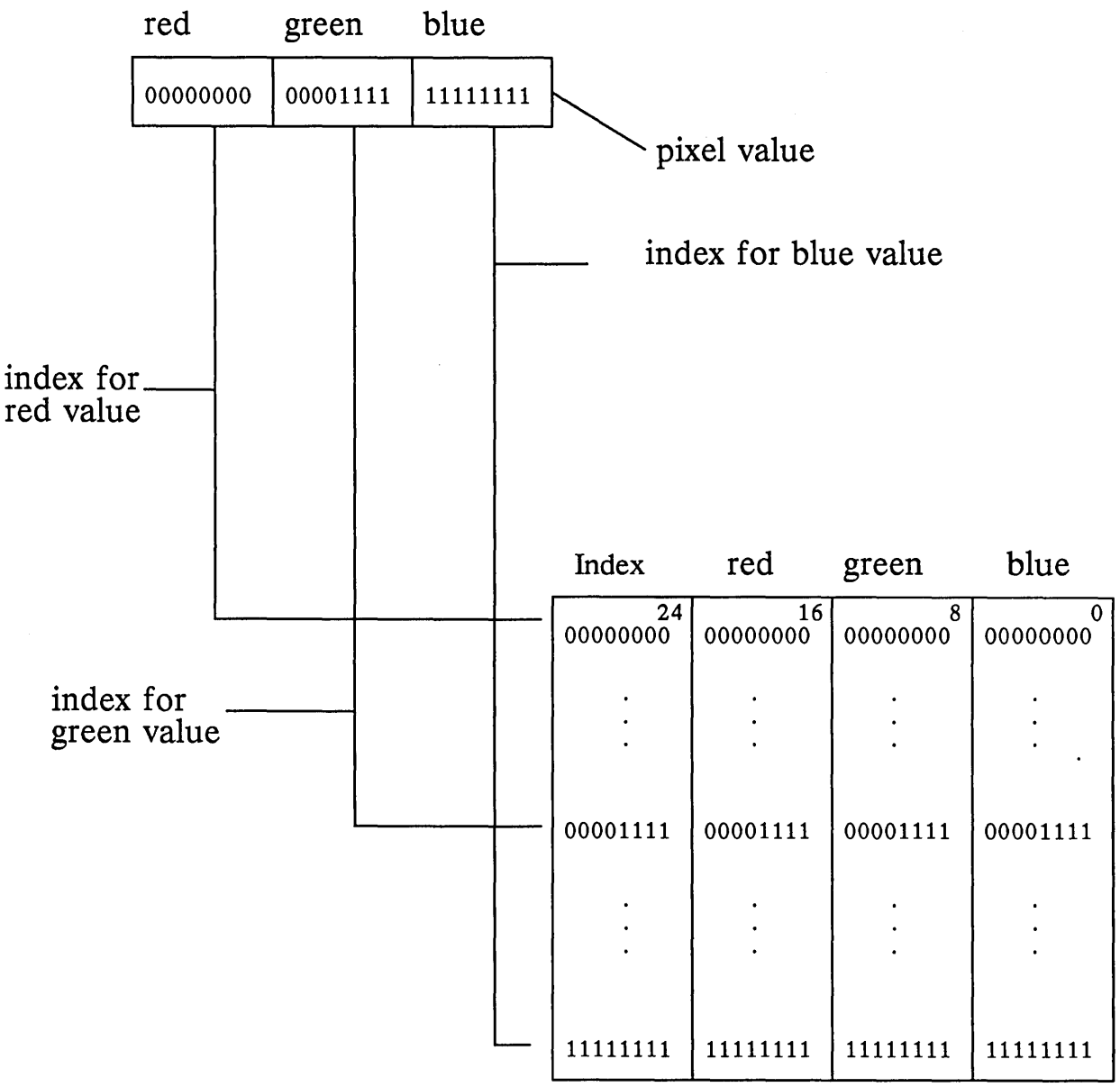
Color maps for 24-bit imaging are slightly different. The color map can be thought of as a 256 x 3 matrix. The color value is still represented by a 4-byte integer, (however, only three bytes are used), but each byte can be indexed separately. The rows of the matrix represent intensity levels and the columns represent the primary colors.

In this format, the 3-byte pixel value is divided into three 8-bit fields. The value in each field is associated with a particular column of the matrix. In other words, eight bits of the pixel value are an index into the red column, eight bits of the pixel value are an index into the green column, and eight bits of the pixel value are an index into the blue column. This allows you to choose the intensity of each primary color. Figure 8-3 displays how the color map is used in 24-bit imaging.

With 24-bit imaging, the color map is the same size as it is for 8-bit interactive format or 8-bit imaging format. You get access to more colors because you are using eight bits from the pixel value as an index to each primary color (a column in the matrix).

8.3. Establishing A Color Map

At initialization of the graphics primitives package, a default color map is established. The default color maps for monochromatic and color displays are displayed in Tables 8-3 and 8-4. In frame mode or direct mode, a program cannot modify color map entries 0 and 7-15. These colors are used by the display manager for window backgrounds and borders. All other entries in the map can be modified.



This pixel value specifies no red, half green, and full blue.

Figure 8-3. From Pixel to Color Map in 24-bit Imaging

8.3.1. Using a Color Map

After a color map is established, a program can use it to specify the color/intensity to use for displaying lines, text, text background, fill operations, and the full screen, as follows. The program assigns a pixel value (color map index) to the draw value attribute, the text value attribute, the text background value attribute, the fill value attribute, and/or uses the index to clear the screen. See the description of the following routines in the DOMAIN System Call Refrence (Volume I):

```
GPR_$SET_DRAW_VALUE
GPR_$INQ_DRAW_VALUE
GPR_$SET_TEXT_VALUE
GPR_$SET_TEXT_BACKGROUND_VALUE
GPR_$INQ_TEXT_VALUES
GPR_$SET_FILL_VALUE
GPR_$INQ_FILL_VALUE
GPR_$CLEAR
```

Table 8-3. Default Color Map for Monochromatic Displays

Color Table Index	Color Value	Resultant Visible Color/Intensity
0	0	black
1	16#FFFFFF	white

To establish a particular color value, you must specify the amount of each primary color. This can be done in FORTRAN and Pascal as shown in the following subsections.

8.3.2. FORTRAN Example to Establish a Color Value

The FORTRAN function presented in this section returns a 4-byte integer that contains intensity values for each primary color. The parameters red, green, and blue must be assigned values in the range 0 - 255.

```
Integer*4 Function color_entry(red,green,blue)
  integer*2 red,green,blue

     color_entry = (65536 * red) + (256 * green) + blue
     return

end
```

Table 8-4. Default Color Map for Color Displays

Color Table Index	Color Value				Resultant Visible Color/Intensity
	R	G	B		
0	0	0	0	(GPR_$BLACK)	black
1	255	0	0	(GPR_$RED)	red
2	0	255	0	(GPR_$GREEN)	green
3	0	0	255	(GPR_$BLUE)	blue
4	0	255	255	(GPR_$CYAN)	cyan
5	255	255	0	(GPR_$YELLOW)	yellow
6	255	0	255	(GPR_$MAGENTA)	magenta
7	255	255	255	(GPR_$WHITE)	white
8-15	contain colors used by the Display Manager to display windows.				
16-255	0	0	0	(GPR_$BLACK)	black

8.3.3. Pascal Example to Establish a Color Value

The Pascal function presented in this section returns
a 4-byte integer
that contains intensity values for each primary color.
The parameters red, green, and blue must be assigned values in the
range 0 - 255.

```
Function color_entry(IN red : integer;
                     IN green: integer;
                     IN blue : integer) : integer32;

begin
     color_entry := (65536 * red) + (256 * green) + blue;
end;
```

8.3.4. Modifying a Color Table

There is one system color table. To modify it, use GPR_$SET_COLOR_MAP. The format of the call is the following:

```
GPR_$SET_COLOR_MAP(start_index, n_entries, values, status)
```

This call allows you to set several consecutive color values in the color map with just one call. Start_index is the index of the first entry to be modified. N_entries is the number of entries to be modified; values is an array that contains the new color values that you are placing in the color map.

If you want to modify the color map, but the entries you want to modify are not consecutive, you have to use GPR_$SET_COLOR_MAP for each entry, or each group of consecutive entries.

8.3.5. Changing Pixel Values

To change the color of a pixel, line, text, etc., you can use either of two procedures.

1. You can change the color value that is stored in the location that corresponds to the pixel value index. When you do this, any other pixels with the same pixel value index will also change color because they look to that location for a color value.

2. You can change the draw value, text value, etc., if another location in the color map stores the color value you need.

8.3.6. Color Map for Monochromatic Displays

For a monochromatic display, the color map has only two entries. The default color map assigns the color value 0 to color map index 0, and the color value 1 to color map index 1 (see Table 8-3). If a program uses the default color map and sets a particular bitmap pixel to 1, (GPR_$WHITE), the corresponding pixel on the screen appears bright. If it selects 0, (GPR_$BLACK), the corresponding pixel appears dark. On a monochromatic node, there are no other choices -- you cannot get grey-scale output.

8.3.7. Saving/Restoring Pixel Values

In interactive formats, a program can read the pixel values of each pixel in a bitmap or section of a bitmap and store the values in a pixel array. Imaging formats do not permit read operations. In both interactive and imaging formats, a program can write the pixel values from a pixel array into a bitmap. See the routines GPR_$READ_PIXELS and GPR_$WRITE_PIXELS.

8.4. Using Color Display Formats

Interactive display formats fully support all GPR output operations -- bit-block transfer, area filling, line drawing, text manipulation. Imaging display formats support only limited display operations -- displaying (not reading) pixel data and changing the color map. Other functions return error messages.

Imaging display formats make it possible to display images with more bits per pixel than are available with interactive formats. Additionally, in 24-bit pixel format, operations to select a frame (GPR_$SELECT_COLOR_FRAME) are allowed. These operations are used to look at either half of display memory.

8.4.1. Using Imaging Display Formats

Switching the display between an interactive format and an imaging format causes the hardware to reconfigure the refresh buffer memory and to rearrange the bitmap. This means that an intelligible image in one format becomes unintelligible in another.

The imaging formats are supported only in borrow-display mode. To change from an interactive to an imaging format, you must be in borrow-display mode.

8.4.2. Routines for Imaging Display Formats

Use the following routines and procedures for imaging display formats. For a detailed description of these calls, see the GPR calls in the reference library.

1. Establish borrow-display mode:

   GPR_$INIT

   You may or may not first want to perform some graphics operations in interactive format.

2. Set the display to 24-bit pixel format:

   GPR_$SET_IMAGING_FORMAT

   Use the format argument to switch to 8- bit or 24-bit imaging format.

3. To inquire about the format, use:

   GPR_$INQ_IMAGING_FORMAT

4. To establish new values for the color map, use:

   GPR_$SET_COLOR_MAP

5. To write pixel data to the display, use:

   GPR_$WRITE_PIXELS

6. To return to interactive format, use the following call with the

interactive argument:

```
GPR_$SET_IMAGING_FORMAT
```

7. To terminate the session and return the display to the Display Manager, use:

```
GPR_$TERMINATE
```

8.5. Color Zoom Operations

The DOMAIN color displays have a hardware zoom feature, to make an image larger. This feature only works on color displays and only in borrow-display mode. The zooming is done by pixel replication.

You specify a separate zoom factor for the x and y directions. One pixel in display memory is then shown on the screen in x by y pixels (see Figure 8-4).

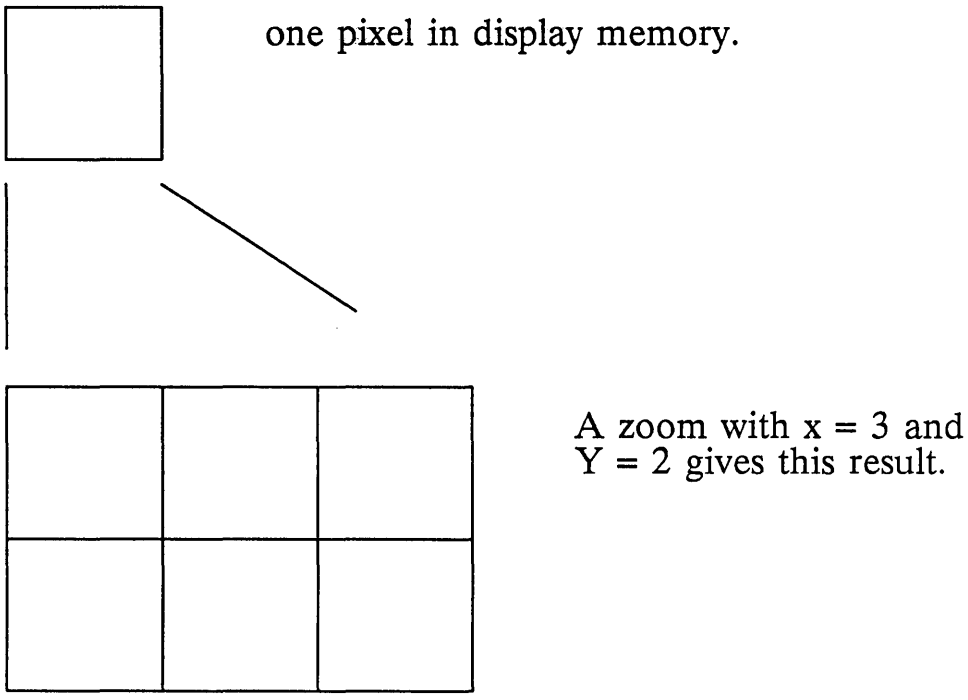


Figure 8-4. Color Zoom

If desired, you can keep the aspect ratio equal by making x and y equal. The zoom always starts at the upper left corner of the screen.

8.6. Color Examples

Three programming examples are presented in this section to demonstrate how to load a color map and how the following GPR routines work:

- GPR_$SET_COLOR_MAP
- GPR_$SET_DRAW_VALUE
- GPR_$SET_FILL_VALUE
- GPR_$SET_TEXT_VALUE.

8.6.1. A Program to Draw a Rectangle and Text in Color

This program draws an unfilled rectangle with text. It is identical to the program in Section 4.9 except that routines to set the draw value and text value have been added. This program uses the default color map.

```
Program color_rec_text;
 %nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %include '/sys/ins/time.ins.pas';
 %list;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;

 var
   init_bitmap_size : gpr_$offset_t;   {size of the initial bitmap}
   init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
   mode        : gpr_$display_mode_t := gpr_$borrow;
   hi_plane_id : gpr_$plane_t := 4;  {highest plane in bitmap}
   delete_display : boolean; {This value is ignored in borrow mode.}
   status  : status_$t; {error code}
   font_id : integer; {identifier of a text font}
   i,j     : integer32;
   direction : gpr_$direction_t; {direction of text}
   pause     : time_$clock_t;


 BEGIN
      init_bitmap_size.x_size := 700;
      init_bitmap_size.y_size := 700;
      gpr_$init(mode,1,init_bitmap_size,hi_plane_id,init_bitmap,status);

      gpr_$set_draw_value(4,status); {blue box}
      gpr_$draw_box(100,100,500,500,status);

      gpr_$load_font_file('f7x13.b',SIZEOF('F7X13.B'),font_id,status);
      gpr_$set_text_font(font_id,status);

      gpr_$move(110,90,status);

      gpr_$set_text_value(3,status); {green text}
      gpr_$text('This is the top of the rectangle.', 33,status);
      direction := gpr_$up;
      gpr_$set_text_path(direction,status);
      gpr_$move(90, 490,status);


      gpr_$set_text_value(6,status); {magenta text}
      gpr_$text('This is the side of the rectangle.', 34,status);

      {Keep figure displayed on the screen for five seconds.}
      pause.low32  := five_seconds;
      pause.high16 := 0;
      time_$wait( time_$relative, pause, status );
```

```
    gpr_$terminate(delete_display,status);
end.
```

8.6.2. A Program to Draw a Design in Color

This program draws the design in Figure 4-9. It is identical to the program presented in Section 4.7 except that routines to change the default color map and routines to change the draw and fill values have been added.

```
program color4;
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';       {required insert file}
%INCLUDE '/sys/ins/gpr.ins.pas';        {required insert file}
%INCLUDE '/sys/ins/time.ins.pas';
%LIST;

 const
    one_second = 250000;
    five_seconds = 5 * one_second;


 var
    init_bitmap : gpr_$bitmap_desc_t;
    unobscured  : boolean;
    st    : status_$t;
    mode : gpr_$display_mode_t := gpr_$borrow;
    x,y,x1,y1 :integer;
    rectangle : gpr_$window_t;
    disp_bm_size : gpr_$offset_t := [1024,800]; {size of initial bitmap}
    hi_plane_id  : gpr_$plane_t := 4;
    color_value  :array [0..7] of gpr_$pixel_value_t;
    pause : time_$clock_t;

 function color_entry(IN red : integer;
                      IN green : integer;
                      IN blue : integer) : integer32;
 begin
    color_entry :=lshft(red,16) ! lshft(green,8) ! lshft(blue,0);
 end;

 BEGIN

    color_value[0] := color_entry (0,0,0); {color--black}
    color_value[1] := color_entry (255,125,0); {color--orange}
    color_value[2] := color_entry (255,0,0); {color--red}
    color_value[3] := color_entry (0,255,0); {color--green}
    color_value[4] := color_entry (0,0,255); {color--blue}
    color_value[5] := color_entry (255,255,0); {color--yellow}
    color_value[6] := color_entry (255,0,255); {color--magenta}
    color_value[7] := color_entry (255,255,255); {color--white}

    gpr_$init(mode,1,disp_bm_size,hi_plane_id,init_bitmap,st );
    gpr_$set_color_map(0,8,color_value,st); {modifies color table}
```

```
    x := 200; x1 := 600; y := 200; y1 := 600; {dimensions of box}
    rectangle.window_base.x_coord := 250; {starting position of 1st rectangle}
    rectangle.window_base.y_coord := 250;
    rectangle.window_size.x_size := 50; {width of each rectangle}
    rectangle.window_size.y_size := 50; {height of each rectangle}

    gpr_$set_auto_refresh(true,st);

    gpr_$set_draw_value(4,st);  {blue box}
    gpr_$draw_box(x,y,x1,y1,st); {Draw an unfilled box.}

    gpr_$set_fill_value(5,st);   {yellowbox}
    gpr_$rectangle(rectangle,st); {Draw a filled rectangle.}

    {Draw three more filled rectangles within the unfilled box.}
    rectangle.window_base.x_coord := 500;
    rectangle.window_base.y_coord := 250;

    gpr_$set_fill_value(1,st); {orange box}
    gpr_$rectangle(rectangle,st);

    rectangle.window_base.x_coord := 250;
    rectangle.window_base.y_coord := 500;

    gpr_$set_fill_value(7,st);  {white box}
    gpr_$rectangle(rectangle,st);

    rectangle.window_base.x_coord := 500;
    rectangle.window_base.y_coord := 500;

    gpr_$set_fill_value(2,st);
    gpr_$rectangle(rectangle,st);

    gpr_$move(300,300,st);  {Move the current position.}

    gpr_$set_draw_value(7,st);  {line}
    gpr_$line(500,500,st);  {Draw a line connecting two rectangles.}


    gpr_$move(300,500,st);
    gpr_$set_draw_value(6,st);  {line}
    gpr_$line(500,300,st);  {Draw a line connecting two rectangles.}

    {Keep figure displayed on the screen for five seconds.}
    pause.low32  := five_seconds;
    pause.high16 := 0;
    time_$wait( time_$relative, pause, st);

    gpr_$terminate( false, st );  {Terminate the graphics session.}
END.
```

8.6.3. A Program to Draw Concentric Circles in Color

This program draws seven concentric circles. The outer most circle is drawn first in dark blue. Each addition circle is drawn in a lighter shade of blue except the last one which is drawn in white. The various shades of blue are achieved by loading the color map with the desired shades of blue. The darkest shade of blue has no red, no green, and the maximum amout of blue. Lighter shades of blue have increasing amouts of red and green with the maximum amount of blue.

```
Program color_circles;
 %nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %include '/sys/ins/time.ins.pas';
 %list;
 const
    one_second = 250000;
    five_seconds = 5 * one_second;

 var
   size        : gpr_$offset_t;   {size of the initial bitmap}
   init_bitmap : gpr_$bitmap_desc_t;  {descriptor of initial bitmap}
    ev_pos  : gpr_$position_t;
    ev_type : gpr_$event_t;
    event_data,ev_char : char;
    unobscured         : boolean;
    st   : status_$t;
    keys : gpr_$keyset_t;  {set of characters}

    mode        : gpr_$display_mode_t := gpr_$borrow;
    hi_plane_id : gpr_$plane_t := 3;  {highest plane in bitmap}
    center : gpr_$position_t := [300,300];
    radius : integer; { := 200;}
    delete_display : boolean; {This value is ignored in borrow mode.}
    status : status_$t; {error code}
    cv : integer;
    pause : time_$clock_t;
    color_value :array [0..7] of gpr_$pixel_value_t;


 function color_entry(IN red : integer;
                      IN green : integer;
                      IN blue : integer) : integer32;
 begin
    color_entry :=lshft(red,16) ! lshft(green,8) ! lshft(blue,0);
 end;

 BEGIN
    size.x_size := 700;
    size.y_size := 700;
    gpr_$init(mode,1,size,hi_plane_id,init_bitmap,status);


    color_value[0] := color_entry (0,0,0); {color--black}
    color_value[1] := color_entry (0,0,255); {color--dark blue}
```

```
    color_value[2] := color_entry (50,50,255);
    color_value[3] := color_entry (75,75,255);
    color_value[4] := color_entry (100,100,255);
    color_value[5] := color_entry (150,150,255);
    color_value[6] := color_entry (200,200,255); {color--light blue}
    color_value[7] := color_entry (255,225,225); {color--white}

    gpr_$set_color_map(0,8,color_value,st); {modifies color table}
    radius := 300;

{Draw concentric circles - each a lighter color of blue.}
{ The last circle is white.}
    for cv := 1 to 7 do
        begin
      gpr_$set_draw_value(cv,status);
      gpr_$circle(center,radius,status);

      gpr_$set_fill_value(cv,status);
      gpr_$circle_filled(center,radius,status);
      radius := radius - 50;
    end;  {for}
     {Keep figure displayed on the screen for five seconds.}
     pause.low32  := five_seconds;
     pause.high16 := 0;
     time_$wait( time_$relative, pause, status );

    gpr_$terminate( false, st );  {Terminate the graphics session.}

END
```

Chapter 9
Graphics Map Files

9.1. A Graphics Map File

A graphics map file, or GMF, is an image of the graphic information in a bitmap. Each bit in the GMF represents the state of one visible point on the display. On DOMAIN color nodes, where more than one plane is used to represent visible information, a GMF stores the state of only one plane, typically plane 0.

Once you have stored image data in a GMF, you can restore it to the display or produce a printed copy of the image. The GMF contains information that helps the GMF manager or application program interpret the contents of the GMF. For instance, the GMF may indicate the following: the physical density of the original image, and the dimensions of the display area stored in the GMF. A GMF can contain the contents of an entire plane or any specified rectangular portion of the plane (subplane).

In Software Releases 6.0 and earlier, GMFs were called graphics metafiles. The calls to the GMF manager begin with the letters GMF. To store image data in a GMF, you typically use GMF_$OPEN to create or open the GMF, then use GMF_$COPY_PLANE to specify the information to be copied into the GMF, then close the GMF with GMF_$CLOSE.

The GMF_$COPY_PLANE routine copies a plane of display memory. To make a GMF of any rectangular area on the display, regardless of its position, use the more general call GMF_$COPY_SUBPLANE.

The GMF_$RESTORE_PLANE call returns to the screen any image data that is stored in a specified GMF. To use this call, the node must be in borrow-display mode. The call changes a rectangular portion of the display, with the size determined by the size of the GMF you specify.

In place of graphic map files, it is *strongly recommended* that you use external bitmap files. For a description of the routine for creating such files, see GPR_$OPEN_BITMAP_FILE in Section 7.2 of this manual.

9.2. Insert Files

To use GMF calls in an application program, the following insert file must be included in your program:

FORTRAN	Pascal	C
/SYS/INS/BASE.INS.FTN	SYS/INS/BASE.INS.PAS	SYS/INS/BASE.INS.C
/SYS/INS/GMF.INS.FTN	SYS/INS/GMF.INS.PAS	SYS/INS/GMF.INS.C

The GMF manager does not define any new data structures.

9.3. Error Messages

Here are the possible error messages generated by the GMF calls described in this chapter:

```
GMF_$BAD_BPI     -- Bits/inch parameter is negative
GMF_$BAD_X_DIM   -- X dimension parameter is not positive
GMF_$BAD_Y_DIM   -- Y dimension parameter is not positive
GMF_$BAD_WPL     -- Words/line parameter is too small for the
                    X dimension you specified
GMF_$BAD_POS     -- Opening position parameter is illegal
GMF_$NOT_GMF     -- The file you wanted to open is not a GMF
```

9.4. Programming Example

This example in Pascal shows how to combine the calls described in this chapter with GPR calls (see Chapter 11) to form a typical GMF operation. This example restores a previously saved GMF.

```
    {Initialize the graphics primitive package in borrow-display mode.}

gpr_$init (gpr_$borrow, 1, scsize, 0, disp_desc, sts);

   { Get the starting pointer. }

gpr_$inq_bitmap_pointer (disp_desc, ptr, width, sts);

   { Open the file. }

gmf_$open ('//pepsi/adm/gmf/turbine.pad',27,gmf_$read,id,sts);

   { Restore the screen. }

gmf_$restore_plane (id,scsize.x_size,scsize.y_size,wpl,ptr,bpi,sts);

    { Close the file. }

 gmf_$close (id,status);
```

9.4.1. Comments on Programming Example

The call to GPR_$INIT puts the screen in borrow-display mode. The next call obtains "ptr," the pointer to the start of the screen bitmap. The call to GMF_$OPEN opens a GMF with the specified name, returning the identification by which you subsequently refer to the GMF. The next call restores the screen from this GMF. (Alternatively, you can use a call here to copy a plane or subplane to the GMF.) The final call closes the GMF.

Appendix A
Glossary

Attribute Specification of the manner in which a primitive graphic operation is to be performed, (for example line type or text value). Each bitmap has a set of attributes.

Bitmap A three-dimensional array of bits having width, height, and depth. When a bitmap is displayed, it is treated as a two-dimensional array of sets of bits. The color of each displayed pixel is determined by using the set of bits in the corresponding pixel of the frame-buffer bitmap as an index into the color table.

Bit plane A one-bit-deep layer of a bitmap. On a monochromatic display, displayed bitmaps contain one plane. On a color display, displayed bitmaps may contain more planes, depending on the hardware configuration and the number of bits per pixel.

Borrow display mode A mode for use of the DOMAIN display whereby a program borrows the entire screen from the Display Manager and performs graphics operations by directly calling the display driver.

Button A logical input device used to provide a choice from a small set of alternatives. Two physical devices of this type are function keys on a keyboard and selection buttons on a mouse.

Clipping window A rectangular section of a bitmap outside of which graphics operations do not modify pixels.

Color map See Color Table.

Color table A set of color table entries, each of which can store one color value. Each color value contains red, blue, and green components. Each entry is accessed by a color table index.

Color table entry One location in a color table. Each entry stores one color value that can be accessed by a corresponding color table index.

Color table index An index to a particular color table entry.

Color value The numeric encoding of a visible color. A color value is stored in a color table entry. Each color value is divided into three fields: the first stores the value of the red component of the color, the second stores the value of the green component of the color, and the third stores the value of the blue component. Each component value is specified as an integer in the range of zero to 255, where zero is the absence of the primary color and 255 is the full intensity color.

Core graphics system A package of graphics functional capabilities designed for building higher-level

interactive computer graphics applications programs. Unlike graphics primitives, the Core graphics system allows temporary storage of picture data during execution, with limited segmentation of the pictures. In addition, the Core system uses device-independent coordinates.

Current bitmap — The bitmap on which a program is currently operating.

Current position — In graphics primitives, the starting point of any line drawing and text operations. The current position is initially set at the coordinate position at the top left corner of the bitmap (0,0).

Direct mode — A mode for use of the DOMAIN display whereby the program performs graphics operations in a window borrowed from the Display Manager. Direct mode allows graphics programs to coexist with other activities on the screen, with less Display Manager overhead than frame mode.

Event — An input primitive which is associated with an interrupt from a device such as a keyboard, button, mouse, or touchpad.

Font — One set of alphanumeric and special characters. The font in which text is to be displayed may be specified as an attribute.

Frame — A two-dimensional data structure that holds a picture in a Display Manager pad. This structure is looked at through a Display Manager window. The structure can be larger (or smaller) than the window, and it can be scrolled.

Frame buffer — The digital memory in a raster display unit used to store a bitmap.

Frame mode — A mode for use of the DOMAIN display whereby a program performs graphics operations on a Display Manager pad. In this mode, the user has access to other processes through windows on the display, and can scroll the frame under the Display Manager window. In this mode, unlike direct mode, the Display Manager refreshes the window when appropriate.

Imaging display format
An 8-bit or 24-bit color display format which allows display of an extended color range, but supports only limited graphics primitives operations. In an 8-bit imaging format, eight bits are used to assign a pixel value (color map index) to each pixel. In a 24-bit imaging format, 24 bits are used to assign a pixel value to each pixel. An 8-bit imaging format allows 256 colors to appear on the screen at one time. A 24-bit imaging format extends the possible color range to 16 million different colors, with 512 x 512 pixels visible at one time.

Initial bitmap — The first bitmap created in a graphics session.

Input device — A device such as a function key, touchpad, or mouse that enables a user to provide input to a program.

Input device number
The identifier of one input device in an input device class.

Interactive display format
: A 4-bit or 8-bit color display format which supports all graphics primitives operations. In a 4-bit interactive format, four bits are used to assign a pixel value (color map index) to each pixel. In an 8-bit format, eight bits are used to assign a pixel value to each pixel. A 4-bit format allows sixteen different colors to appear on the screen at one time. An 8-bit format allows 256 colors to appear on the screen at one time.

Keyboard
: A logical input device used to provide character or text string input. One physical device of this type is the alphanumeric keyboard.

Line style
: An attribute that specifies the style of lines and polylines (for example, solid or dotted).

Locator
: A logical input device used to specify one position in coordinate space (for example, a touchpad, data tablet, or mouse).

Logical input device
: An abstraction of an input device that provides a particular type of input data. This abstraction corresponds to a group of physical input devices that provide this type of input data.

No-display mode
: A mode for use of the DOMAIN system whereby a program creates a bitmap in nondisplayed memory and performs graphic operations there, bypassing the display.

Picture element
: A single element of a two-dimensional displayed image or of a two-dimensional location within a bitmap. It is commonly called a pixel.

Pixel
: See Picture Element.

Pixel value
: The set of bits at a two-dimensional location within a bitmap. A pixel value is used as an index to the color map.

Plane
: See Bit Plane.

Primitive
: The least divisible graphic operation that changes a bitmap (for example, lines, polylines, and text).

Primitive attribute
: See Attribute.

RGB color model
: A model used to specify color values. It defines red, green, and blue as primary colors. All other colors are combinations of the primaries, including the three secondary colors (cyan, magenta, and yellow).

Scan line
: A row of pixels; one horizontal line of a bitmap.

Window
: A rectangular area of the visible screen. Parts of the area may be obscured by other windows.

Appendix B
Keyboard Charts

The following two charts and figures give the 8-bit ASCII values generated for two DOMAIN keyboards: 880 and low-profile. These charts include characters used in keystroke events. The columns represent the four highest order bits of an 8-bit value. The rows represent the four lowest order bits of an 8-bit value. For a more complete description of conventions for naming keys, see the *DOMAIN System Command Reference*.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	^SP	^P	SP	0	@	P	`	p		R1		R1U	F1	F1S	F1U	F1C
1	^A	^Q	!	1	A	Q	a	q	L1	R2	L1U	R2U	F2	F2S	F2U	F2C
2	^B	^R	"	2	B	R	b	r	L2	R3	L2U	R3U	F3	F3S	F3U	F3C
3	^C	^S	#	3	C	S	c	s	L3	R4	L3U	R4U	F4	F4S	F4U	F4C
4	^D	^T	$	4	D	T	d	t	L4	R5	L4U	R5U	F5	F5S	F5U	F5C
5	^E	^U	%	5	E	U	e	u	L5	BS	L5U	R2S	F6	F6S	F6U	F6C
6	^F	^V	&	6	F	V	f	v	L6	CR	L6U	R3S	F7	F7S	F7U	F7C
7	^G	^W	'	7	G	W	g	w	L7	TAB	L7U	R4S	F8	F8S	F8U	F8C
8	^H	^X	(	8	H	X	h	x	L8	STAB	L8U	R5S	R1S	L8S	L1A	L1AU
9	^I	^Y	)	9	I	Y	i	y	L9	CTAB	L9U		L1S	L9S	L2A	L2AU
A	^J	^Z	*	:	J	Z	j	z	LA		LAU		L2S	LAS	L3A	L3AU
B	^K	ESC	+	;	K	[	k	{	LB		LBU		L3S	LBS	R6	R6U
C	^L	^\	,	<	L	\	l	|	LC		LCU		L4S	LCS	L1AS	
D	^M	^]	-	=	M	]	m	}	LD		LDU		L5S	LDS	L2AS	
E	^N	^~	.	>	N	^	n		LE		LEU		L6S	LES	L3AS	
F	^O	^?	/	?	O		o	DEL	LF		LFU		L7S	LFS	R6S	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure B-1. Low-Profile Keyboard Chart - Translated User Mode

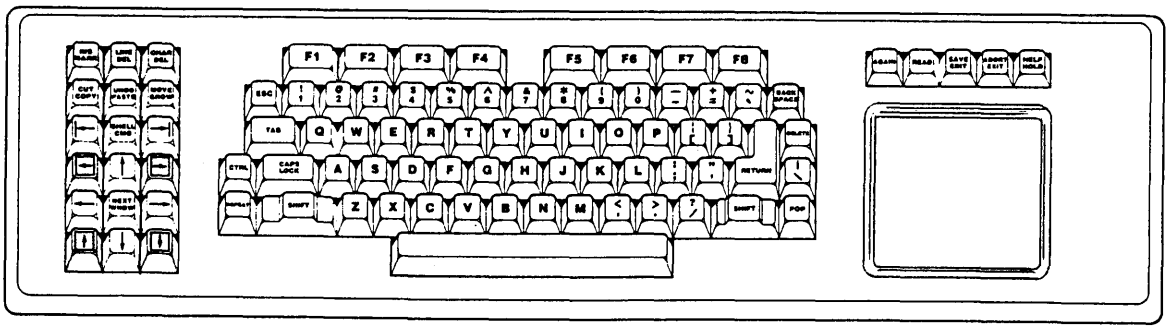


Figure B-2. Low-Profile Keyboard

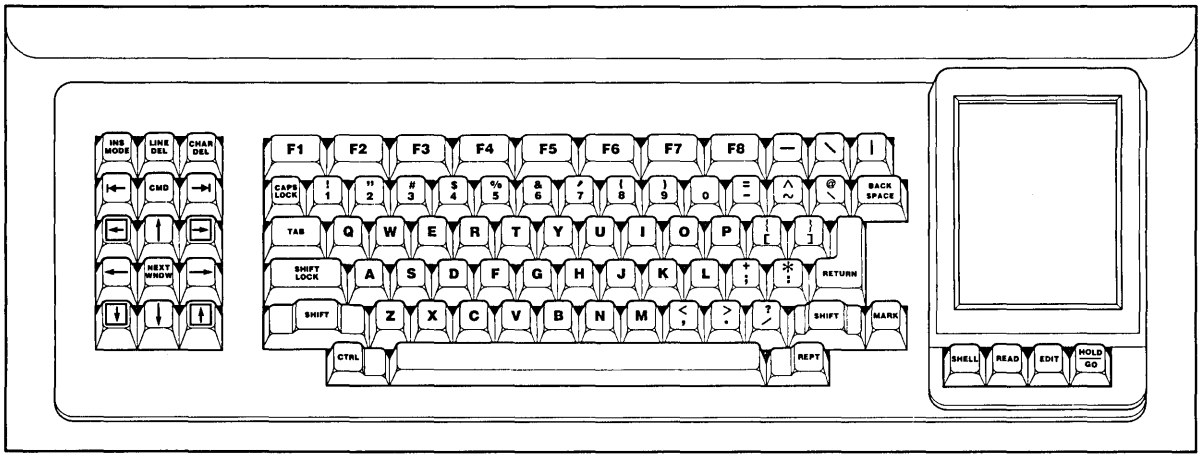


Figure B-3. 880 Keyboard

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	^`	^P	SP	0	@	P	`	p		R1		R1U	F1	F1S	F1U	F1C
1	^A	^Q	!	1	A	Q	a	q	L1	R2	L1U	R2U	F2	F2S	F2U	F2C
2	^B	^R	"	2	B	R	b	r	L2	R3	L2U	R3U	F3	F3S	F3U	F3C
3	^C	^S	#	3	C	S	c	s	L3	R4	L3U	R4U	F4	F4S	F4U	F4C
4	^D	^T	$	4	D	T	d	t	L4	R5	L4U	R5U	F5	F5S	F5U	F5C
5	^E	^U	%	5	E	U	e	u	L5	BS	L5U		F6	F6S	F6U	F6C
6	^F	^V	&	6	F	V	f	v	L6	CR	L6U		F7	F7S	F7U	F7C
7	^G	^W	'	7	G	W	g	w	L7	TAB	L7U		F8	F8S	F8U	F8C
8	^H	^X	(	8	H	X	h	x	L8	STAB	L8U		N0	N8	N0U	N8U
9	^I	^Y	)	9	I	Y	i	y	L9	CTAB	L9U		N1	N9	N1U	N9U
A	^J	^Z	*	:	J	Z	j	z	LA		LAU		N2	N.	N2U	N.U
B	^K	^[	+	;	K	[	k	{	LB		LBU		N3	N=	N3U	N=U
C	^L	^\	,	<	L	\	l	|	LC		LCU		N4	N+	N4U	N+U
D	^M	^]	-	=	M	]	m	}	LD		LDU		N5	N-	N5U	N-U
E	^N	^~	.	>	N	^	n	~	LE		LEU		N6	N*	N6U	N*U
F	^O	^/	/	?	O		o	^	LF		LFU		N7	N/	N7U	N/U
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure B-4. 880 Keyboard Chart - Translated User Mode

Appendix E

Decomposition and Rendering Techniques

This appendix describes the various decomposition and rendering techniques available in GPR.

E.1. The Need for New Decomposition Techniques

New features affecting filled primitives provide flexibility for filling operations, raster operations on filled primitives, and performance improvements during filled-polygon rasterization. These features affect both the decomposition and rasterization of filled primitives drawn with the following routines:

- GPR_$TRIANGLE
- GPR_$MULTITRIANGLE
- GPR_$TRAPEZOID
- GPR_$MULTITRAPEZOID
- GPR_$CLOSE_FILL_PGON

GPR_$PGON_DECOMP_TECHNIQUE implements the new features by allowing you to choose a decomposition and rasterization technique. The available choices are:

- GPR_$FAST_TRAPS. This value indicates that filled polygons are decomposed into trapezoids. The decomposed polygons are rendered as a group of trapezoids.
- GPR_$PRECISE_TRAPS. This value indicates that filled polygons are decomposed into trapezoids. The rendering algorithm that is used is slower but more accurate for self-intersecting polygons than the algorithm used for GPR_$FAST_TRAPS. The decomposed polygons are rendered as a group of trapezoids.
- GPR_$NON_OVERLAPPING_TRIS. This value indicates that filled polygons are decomposed into triangles. The decomposed polygons are rendered as a group of triangles.
- GPR_$RENDER_EXACT. This value indicates that polygons are to be decomposed into individual pixels. The decomposed polygons are rendered pixel by pixel. When possible, adjacent pixels are grouped together and the group of pixels is rendered.

E.1.1. Decomposition Versus Rasterization

Polygon decomposition is the process of breaking a complex polygon down into simpler elements. Currently, GPR decomposes a complex polygon into groups of triangles, trapezoids, or individual pixels (possibly rectangles). In GPR, decomposition takes place in software on all devices. Figure E–1 displays a six–sided polygon decomposed into both triangles and trapezoids.

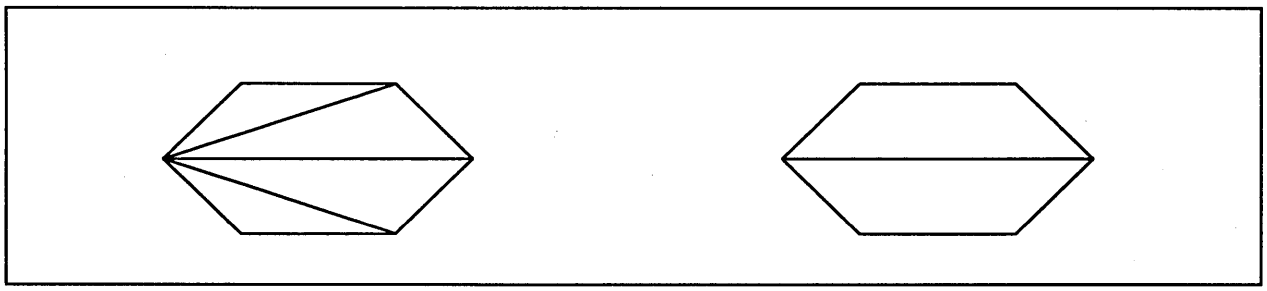


Figure E–1. A Polygon Decomposed into Triangles and Trapezoids

Rendering or rasterization is the process of representing a graphic object after it has been decomposed. A list of primitive objects that the polygon was decomposed into is passed to the rendering algorithm and this algorithm decides which pixels belong to each primitive object. For example, the triangle technique passes a list of triangles to the rendering algorithm, and the rendering algorithm decides which pixels belong to each triangle. Rendering takes place in software or microcode depending on the device. Our more sophisticated hardware devices (DN5XXs and DN6XXs) provide rendering algorithms in microcode.

E.1.2. Comparing the Techniques

Prior to Software Release 9.2 the trapezoid techniques were the only available decomposition and rendering techniques. At Software Release 9.2, the triangle technique was introduced to complement the capabilities of the DN570s and DN580s and to provide the basis for implementing new filling techniques and raster operations on filled primitives. At Software Release 9.5 a new technique, GPR_$RENDER_EXACT, is being introduced. This technique is provided to decompose and render polygons that are not accurately rendered by the triangle technique.

The major difference among the available techniques is in the pixels that are rendered after decomposition. The trapezoid and triangle techniques render slightly different pixels. In most cases, the triangle and render–exact techniques render identical pixels; however, differences may occur with self–intersecting polygons. For these polygons, the render–exact technique provides a more accurate rendering. In addition, minor differences exist in rendering speed. This issue is discussed in Section E.3. The differences between the various techniques are described in the following sections.

Triangle versus Trapezoid Decomposition

The following two problems exist with polygons decomposed into trapezoids:

1) Adjacent trapezoids overlap.

2) Adjacent polygons decomposed into trapezoids overlap.

The trapezoid technique includes the following pixels as a part of a trapezoid:

- Every pixel whose center lies within the boundary of the trapezoid
- Every pixel that is touched by the boundary line of the trapezoid.

Figure E–2 shows the pixels that are rendered for a trapezoid.

NOTE: The polygons in the following figures are drawn with heavy borders to emphasize the polygon in the figure. The filled polygons drawn with GPR routines are borderless.

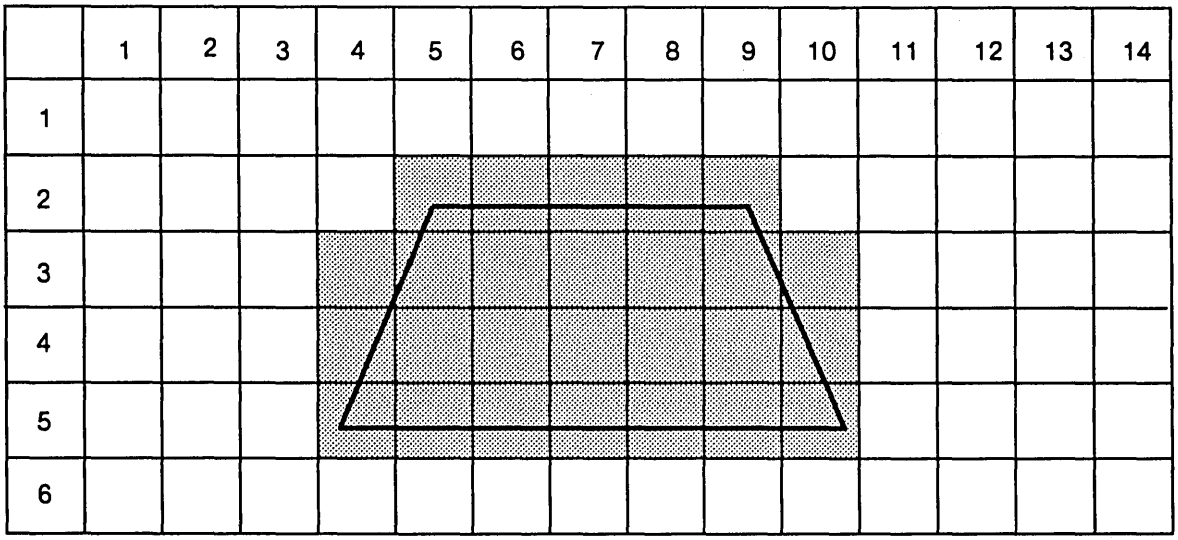


Figure E–2. The Pixels Rendered for a Trapezoid with the Trapezoid Technique

Figure E–3 shows the pixels that are rendered for a six–sided polygon decomposed and rendered using the trapezoid technique. Notice that some of the pixels have been rendered twice. This feature produces undesirable results when some raster operations are applied to this polygon. For example, if a raster operation of six (XOR) were applied, the polygon would appear as displayed in Figure E–4. Similar results exist if adjacent polygons are decomposed into trapezoids. For example, Figure E–5 shows two adjacent polygons decomposed into trapezoids. Notice that all the pixels between adjacent trapezoids as well as the pixels between the adjacent six–sided polygons are rendered more than once. Figure E–6 shows how the two adjacent six–sided polygons would appear if an XOR raster operation were applied. The raster operations work correctly; however, the problem is the overlapping of adjacent trapezoids.

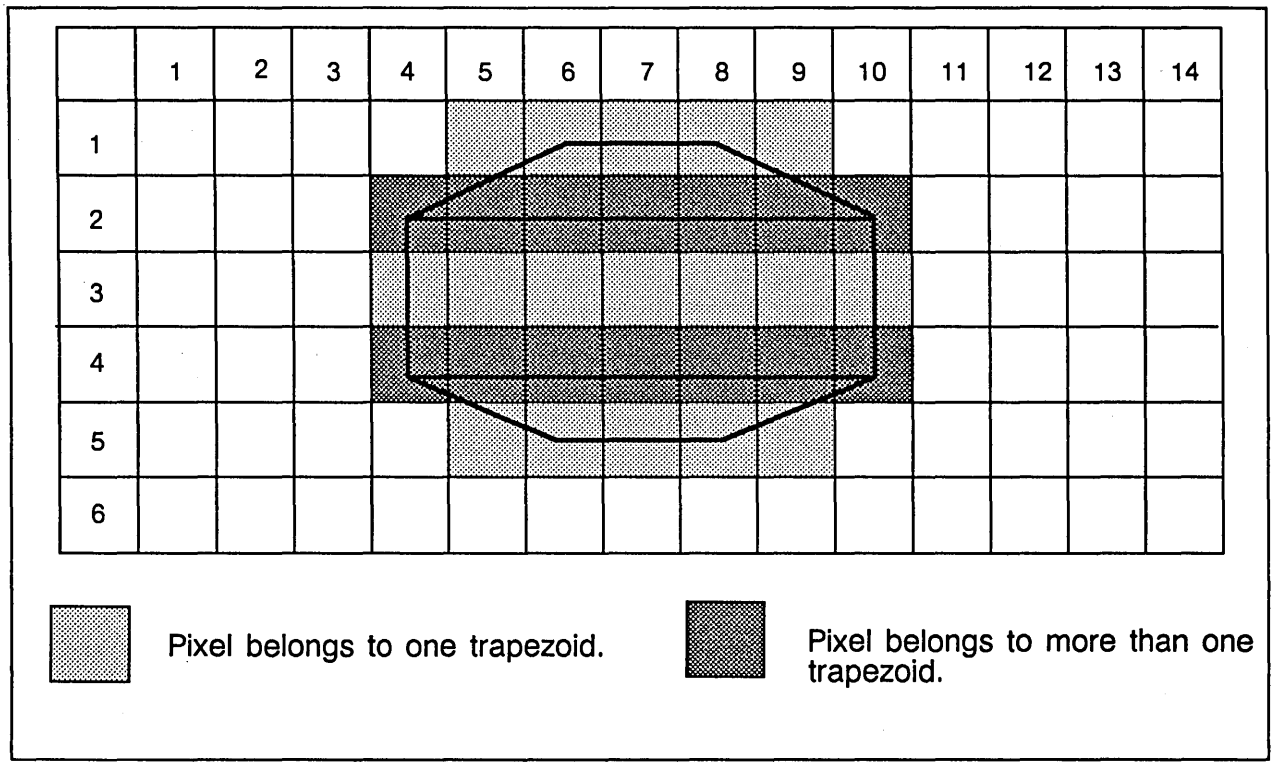


Figure E–3. Interior Pixels of a Six–sided Polygon Decomposed Into Trapezoids

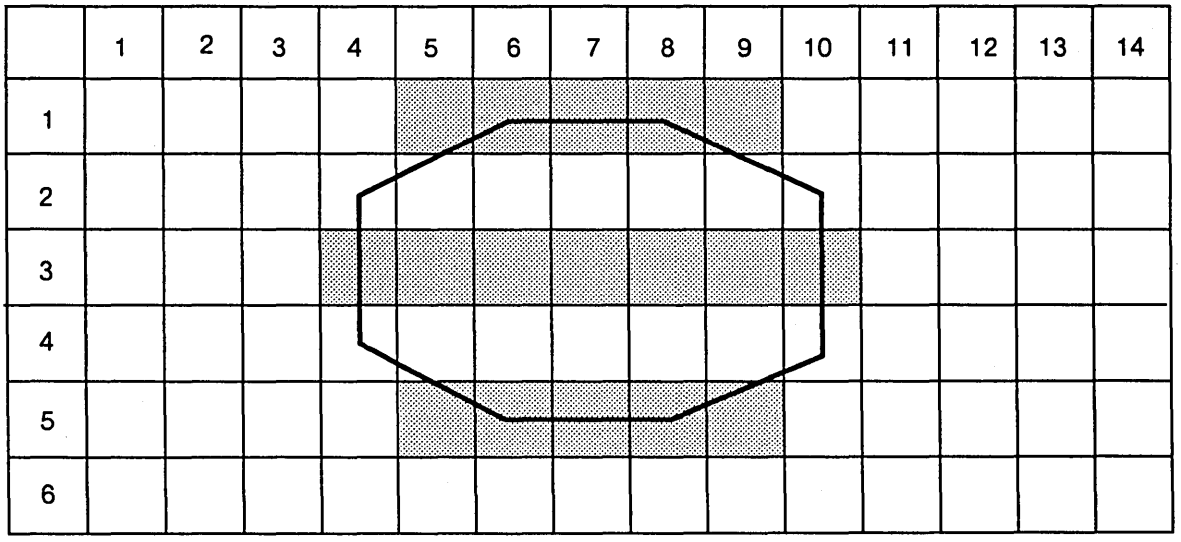


Figure E–4. Six–sided Polygon Decomposed and Rendered with the Trapezoid Technique. The Raster Operation was Set to XOR.

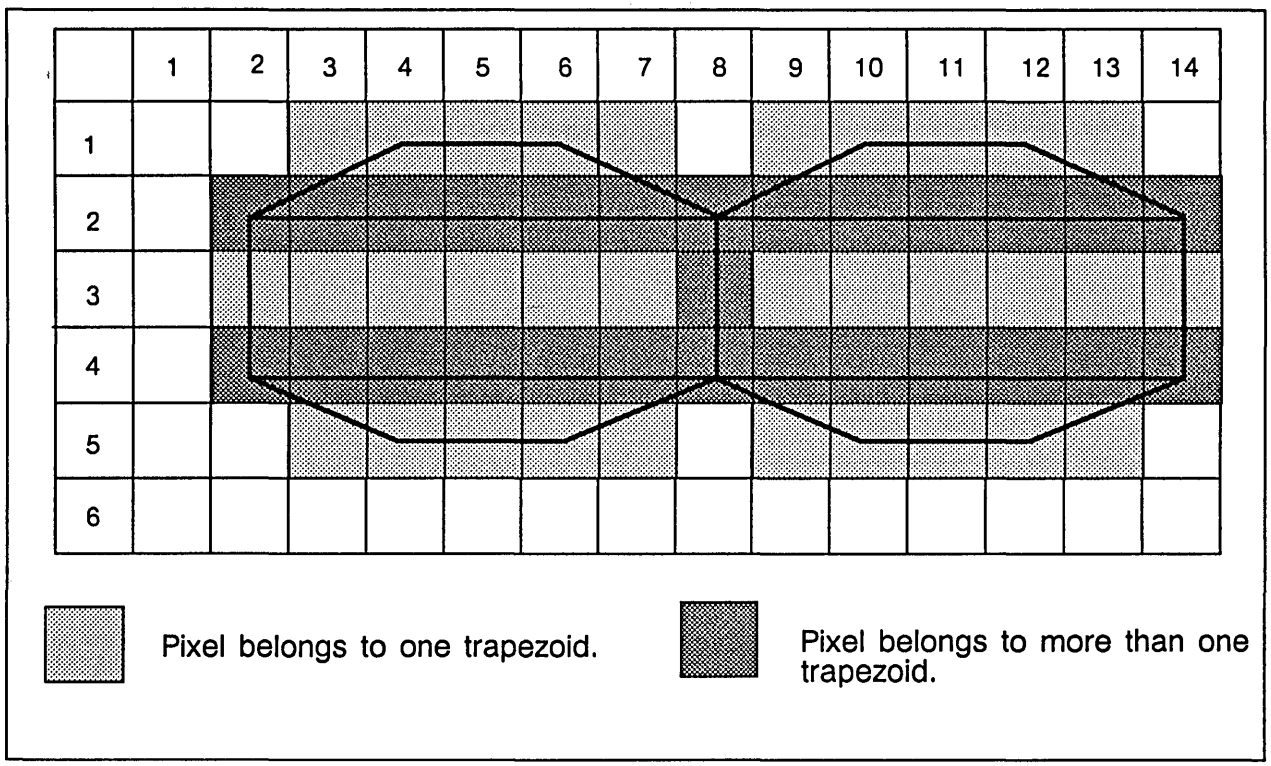


Figure E–5. Two Adjacent Six–sided Polygons Decomposed into Trapezoids

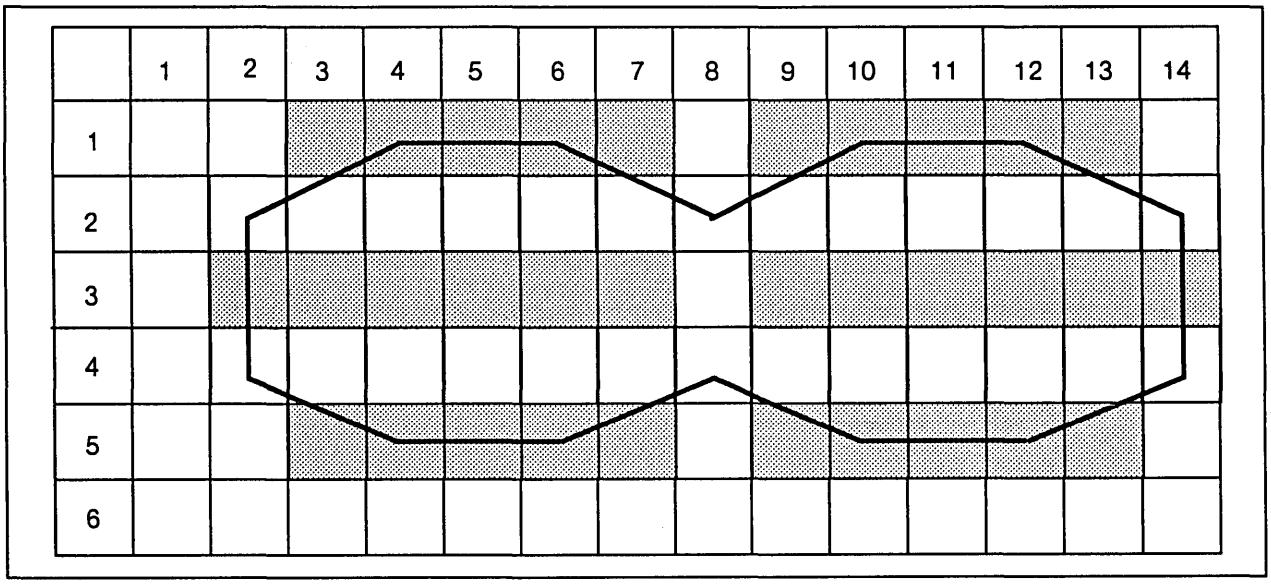


Figure E–6. Two Adjacent Six–sided Polygons Decomposed and Rendered with the Trapezoid Technique. The Raster Operation was Set to XOR.

The triangle rendering algorithm avoids the problems associated with the trapezoid technique by establishing guidelines for pixel selection; the guidelines prevent adjacent polygons and triangles from overlapping. Figure E–7 shows the pixels that would be rendered for a pair of triangles with the triangle technique.

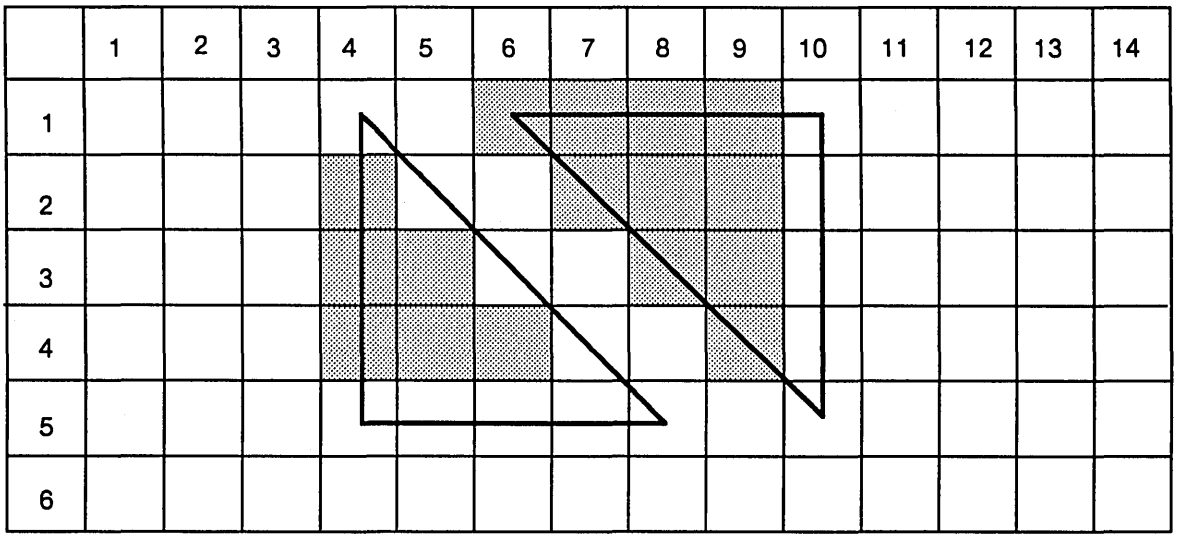


Figure E–7. Interior Pixels of Two Triangles Decomposed into Triangles.

The triangle technique includes the following pixels as a part of a triangle:

- Any pixel whose center lies within the boundary of the triangle
- Any pixel whose center lies on a boundary line is included if the following condition is true: the interior of the polygon lies directly to the right or below the pixel.

The triangle technique excludes any pixels whose center lies on a right–hand boundary line.

> **NOTE:** If a pixel forms the vertex for two boundary lines, the pixel is an interior pixel only if it passes the above criteria for both boundary lines. For example, pixel (1,4) in Figure E–2 is not an interior pixel because it is on a right–hand boundary line.

These rules prevent the triangles that compose a complex polygon from overlapping. In addition, they prevent adjacent polygons decomposed into triangles from overlapping. For this reason, any raster operation can be applied to polygons decomposed and rendered using the triangle technique with satisfactory results.

A six–sided polygon decomposed into triangles is displayed in Figure E–8 (interior pixels are shaded).

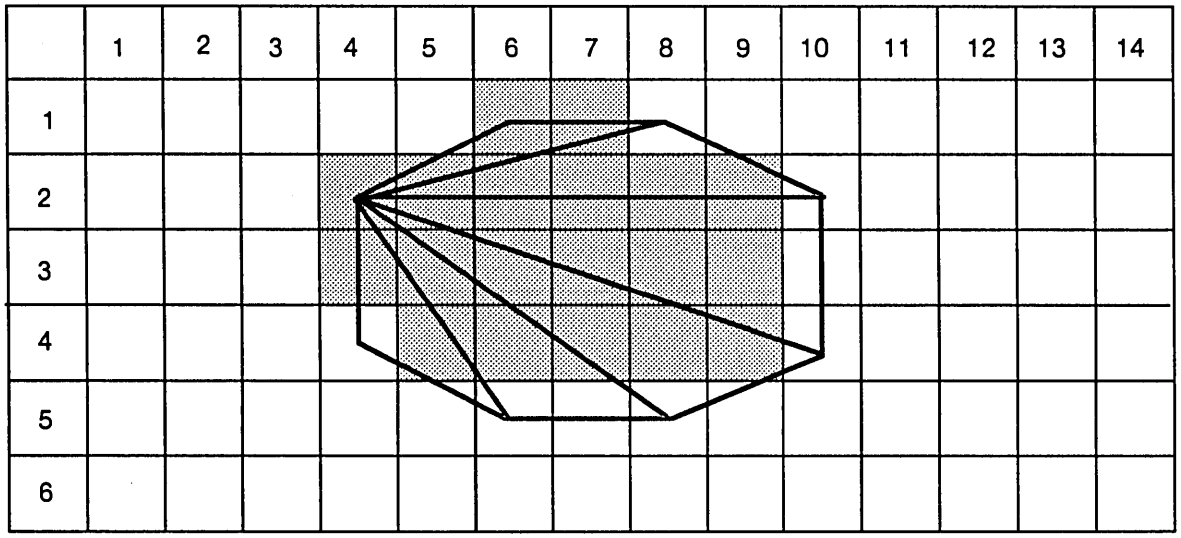


Figure E–8. Six–sided Polygon Decomposed into Triangles.

Triangle versus Render–Exact Techniques

The triangle technique is similar to the trapezoid technique for the following reasons: both techniques decompose a complex polygon into simpler primitives and then render those primitives. The final rendered polygon from either technique is actually a group of several simple polygons. The render–exact technique does not decompose a complex polygon into simpler polygons: it examines each pixel, and if the pixel should be included in the polygon it is rendered. The render–exact technique uses the same criteria for determining which pixels belong to a polygon as does the triangle technique to determine which pixels belong to a triangle. For this reason, most polygons rendered with the triangle technique will be similar to the same polygons rendered with the render–exact technique. The only exceptions are self–intersecting polygons where one of the coordinates of the intersection is a noninteger. For these polygons, the render–exact technique will render a more accurate polygon.

Consider the polygon in Figure E–9. Two edges of the polygon intersect between four pixels. If the triangle technique is used, the decomposition algorithm will move the intersection so that it passes through the center of a pixel. This process actually creates a new polygon which will be different from the original. Figure E–10 displays the polygon in Figure E–9 if the triangle technique is used (rendered pixels are shaded). Figure E–11 displays the polygon in Figure E–9 if the render–exact technique is used (rendered pixels are shaded). For this polygon, render–exact provides a more accurate rendering.

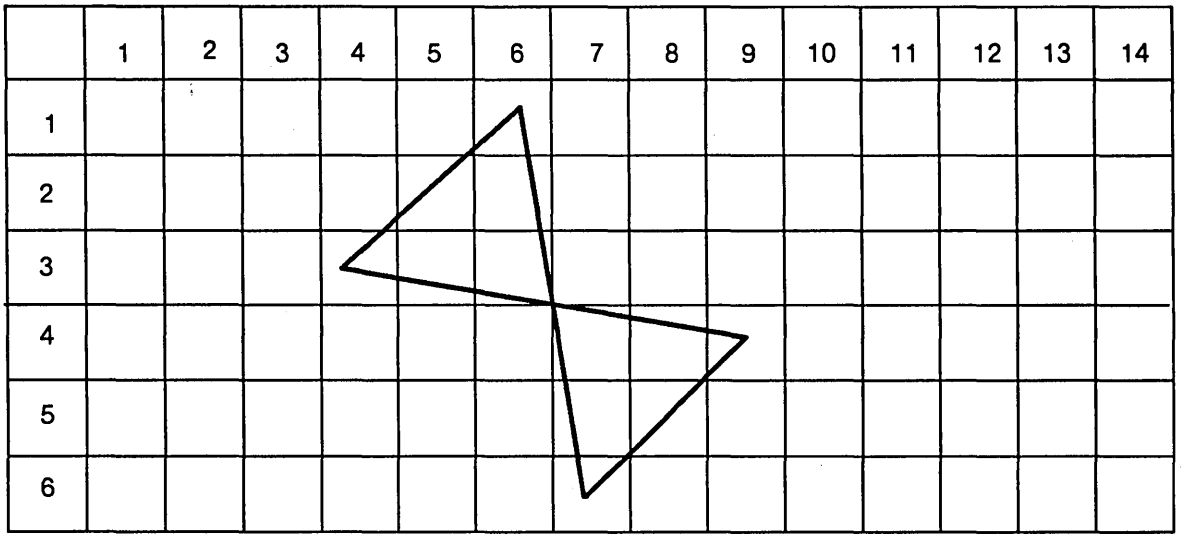


Figure E–9. A Sample Self–Intersecting Polygon

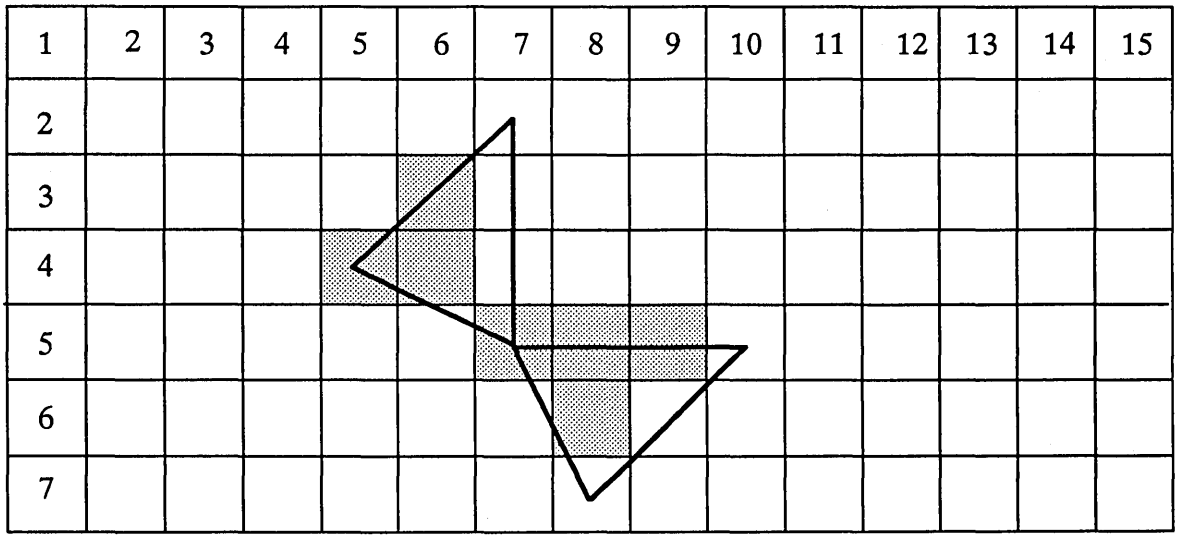


Figure E–10. The Pixels Rendered for the Polygon in Figure E–9 with the Triangle Technique.

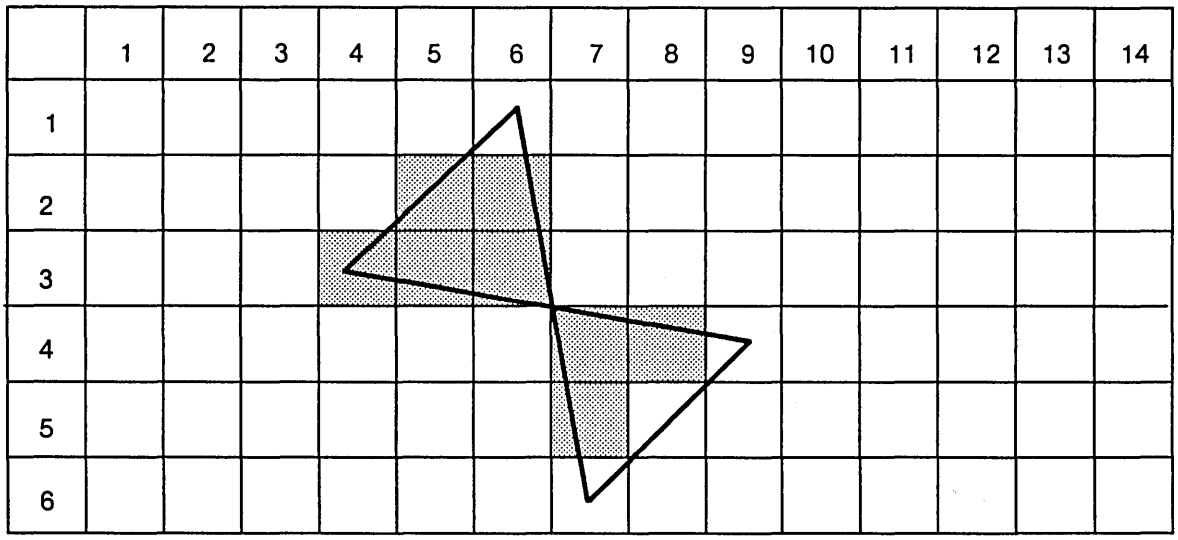


Figure E–11. The Pixels Rendered for the Polygon in Figure E–9 with the Render–Exact Technique.

E.2. Filling Polygons

The decomposition and rendering technique determines the available filling criteria. The following sections describe how the different techniques allow you to fill a polygon. The programs in sections E.7.2 demonstrate how to use various filling criteria with the triangle technique.

E.2.1. Filling Polygons with the Triangle and Render–Exact Techniques

The triangle and render–exact techniques offer the most flexibility for filling polygons because these algorithms calculate winding numbers during decomposition. Winding numbers allow the following filling techniques to be used: parity filling, nonzero filling, and specific winding number filling.

Calculate winding numbers as follows:

1. Trace around the polygon from some point and keep track of the direction of the line.

2. Draw imaginary horizontal lines through the polygon. If the imaginary line passes through a line and the direction of that line is upwards, the winding number for the region to the right of the polygon line is incremented by 1. If the horizontal line passes through a polygon line and the direction of that line is downwards, the winding number for the region to the right of the polygon line is decreased by 1. Initially the winding number is zero.

Figure E–12 illustrates the winding numbers for all regions of the polygon. A parity fill is used to fill the regions with odd winding numbers; a nonzero fill is used to fill all the regions with nonzero winding numbers; and a specific winding number fill is used to fill all the regions with a specific winding number (zero is not allowed as a specific winding number).

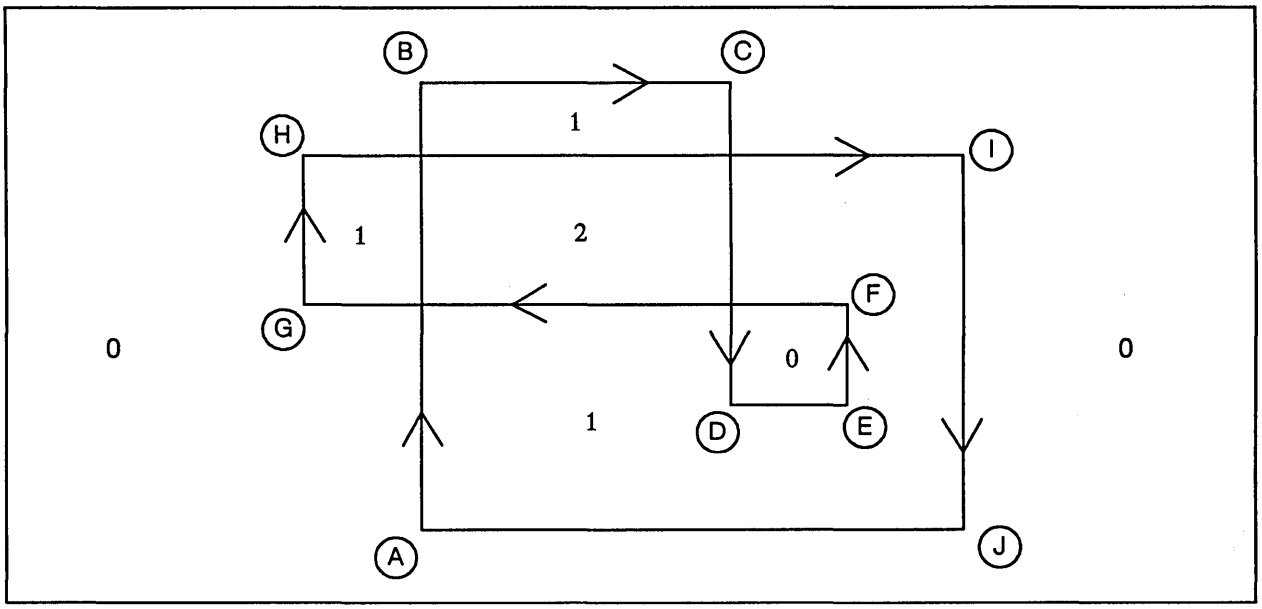


Figure E–12. The Winding Numbers of a Complex Polygon

E.2.2. Filling Polygons with the Trapezoid Technique

The trapezoid technique uses parity filling numbers to determine which areas of a polygon to fill. This prevents some classes of polygons from being completely filled and does not provide any flexibility. Calculate parity filling numbers as follows:

1. Set the filling number outside the polygon to zero.

2. Draw imaginary horizontal lines through the polygon. When an imaginary line passes through a line of the polygon, increase the parity filling number from zero to one. When the line passes through another line of the polygon, decrease the parity filling number from one to zero. Continue this process until the imaginary line is outside the polygon. The parity filling number outside the polygon must always be zero.

Figure E–13 illustrates the parity filling numbers calculated for all regions of the polygon. Only the areas with a filling number of 1 can be filled.

NOTE: The same polygon decomposed and rendered with the triangle or the render–exact technique is guaranteed to be similar to the same polygon decomposed and rendered with the trapezoid technique only in the following situation: when the polygon decomposed with either the triangle or render–exact technique is filled using gpr_$parity as the filling criterion.

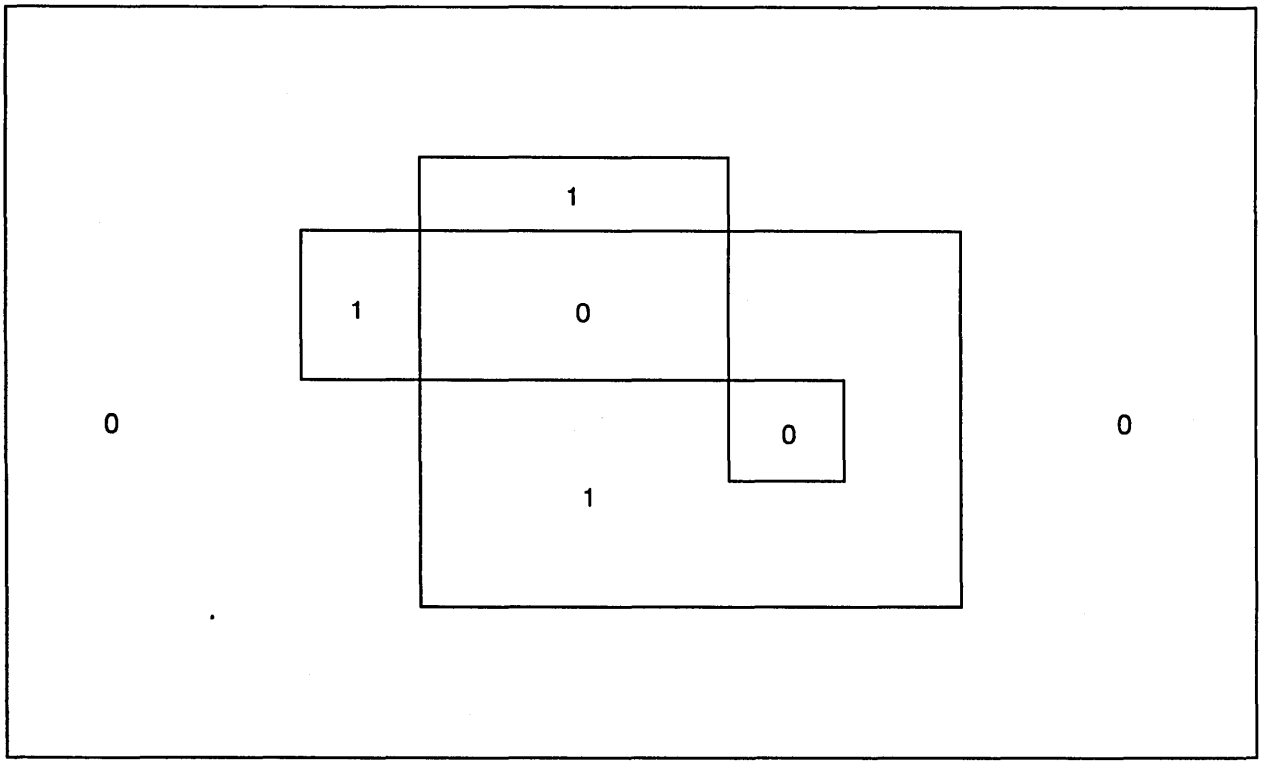


Figure E–13. The Parity Filling Numbers of a Complex Polygon

E.3. Polygons From Start to Fill

To create a filled polygon, perform the following steps:

1. Make certain that the decomposition technique is appropriate. Use GPR_$INQ_PGON_DECOMP_TECHNIQUE to inquire the current decomposition technique. If the current technique is not adequate, change it with GPR_$PGON_DECOMP_TECHNIQUE. See Section E.6 for additional information.

2. Set the filling criterion with GPR_$SET_TRIANGLE_FILL_CRITERIA unless the trapezoid technique is used.

3. Define the starting location of a polygon with GPR_$START_PGON.

4. Define the remaining points of a polygon's boundary with GPR_$PGON_POLYLINE.

   (Steps 3 and 4 can be repeated. See example program in Section E.4.3.)

5. Close the polygon with one of the following routines: GPR_$CLOSE_RETURN_PGON_TRI, GPR_$CLOSE_RETURN_PGON, or GPR_$CLOSE_FILL_PGON.

   GPR_$CLOSE_RETURN_PGON_TRI returns a list of triangles that can be rendered at any time with GPR_$MULTITRIANGLE. (The decomposition technique must be set to GPR_$NON_OVERLAPPING_TRIS.) GPR_$CLOSE_RETURN_PGON_TRI does not render a polygon.

   GPR_$CLOSE_RETURN_PGON returns a list of trapezoids that can be rendered at any time with GPR_$MULTITRAPEZOID. (The decomposition technique must be set to GPR_$FAST_TRAPS or GPR_$PRECISE_TRAPS.) GPR_$CLOSE_RETURN_PGON does not render a polygon.

GPR_$CLOSE_FILL_PGON renders the decomposed polygon immediately; it does not store any list of triangles or trapezoids. Any decomposition technique works with GPR_$CLOSE_FILL_PGON. You must, however, use this procedure to render polygons decomposed using the render-exact technique.

NOTE: An error occurs if you attempt to close a polygon using gpr_$close_return_pgon_tri and the decomposition technique is not gpr_$non_overlapping_tris. An error also occurs if you attempt to close a polygon using gpr_$close_return_pgon and the decomposition technique is not gpr_$fast_traps or gpr_$precise_traps. This means that existing applications that use gpr_$close_return_pgon will have run-time errors on DN570/580s and DN3000s if they use the default decomposition technique.

E.4. The Default Decomposition Techniques

All display devices in the DOMAIN product line use a default decomposition technique to take full advantage of the hardware during rasterization. Depending on your application, the default may or may not be adequate. For example, the trapezoid technique may not be adequate if you are using raster operations on filled polygons.

NOTE: The decomposition technique can affect the application's portability. For example, if you attempt to use a specific winding number fill and the decomposition technique is set to the trapezoid technique, your application will not run to completion.

Table E-1 lists the default decomposition technique used on existing models. The render-exact technique is not the default on any existing hardware devices.

E.5. Performance Considerations

When choosing a decomposition technique, keep in mind the following:

On DN570s and DN580s, triangle technique provides the best performance because the algorithm to render triangles is in microcode. On DN550/560s and DN600/660s, this technique runs considerably slower since the algorithm to render triangles is in software. All other machines, DN100s, DN3XX/4XXs and DN3000s, have rendering algorithms in software regardless of the technique used. If, however, you are filling complex polygons and you need the flexibility that filling with winding numbers provides, or you plan to use raster operations on filled polygons, you must use either the triangle or render-exact technique.

If performance is the only issue, decompose polygons into triangles on DN570/580s and decompose polygons into trapezoids on DN550/560/600/660s. Be aware, however, that the polygons rendered with the triangle technique contain fewer pixels than the same polygon rendered with the trapezoid technique. The difference is minor, but you must be aware that it exists.

The render-exact technique currently provides the best performance for rectilinear axis aligned polygons. Your application may be able to take advantage of this.

Table E-2 shows where the rendering algorithms are located on our current devices.

For static polygons that are frequently displayed, it is efficient to decompose the polygon into a list of trapezoids or triangles . In this way, you avoid the overhead of repeatedly decomposing the same polygon. For polygons that change frequently or polygons that are rendered only once, it is more efficient to use GPR_$CLOSE_FILL_PGON.

Table E–1. The Default Decomposition Techniques Used

Model	Trapezoid Decomposition	Triangle Decomposition
DN300/330/320	X	
DN400/420/460	X	
DN550/560	X	
DN570		X
DN580		X
DN600/660	X	
DN3000		X

E.6. Limitations

You cannot change the decomposition technique from one of the trapezoid techniques to the triangle or render–exact technique when a polygon definition is in progress. Likewise, you cannot change the decomposition technique from the triangle or render–exact technique to one of the trapezoid techniques if a polygon definition is in progress. For example, if you are currently in a polygon operation and the decomposition technique is set to one of the trapezoid techniques, you cannot change the decomposition technique to gpr_$non_overlapping_tris. You can, however, change the decomposition technique either before beginning a polygon operation or upon completion of a polygon operation.

The following is true in borrow mode on DN550/560s and DN6XXs with extended bitmap dimensions (GPR_$SET_BITMAP_DIMENSIONS). Drawing operations cannot span frame 0 and frame 1 if you are using triangle decomposition.

Table E-2. Where Rasterization Occurs

Machine	Decomposition Technique Used		
	TRAPEZOIDS	TRIANGLES	RENDER_EXACT
DN300/330/320	software	software	software
DN400/420/460	software	software	software
DN550/560	microcode	software	software
DN570	software	microcode	microcode
DN580	software	microcode	microcode
DN600/660	microcode	software	software
DN3000	software	software	software

E.7. Sample Programs

There are two example programs presented in this section. Each program is translated into Pascal, FORTRAN, and C.

E.7.1. Programs to Set the Decomposition Technique

These sample programs are intended as shells for future enhancements: the actual graphics application has been omitted. The purpose of these programs is to set the decomposition technique to gpr_$non_overlapping_tris if it is not already set to that value. The program begins by checking the default display type with GPR_$INQ_DISP_CHARACTERISTICS. If the display type is gpr_$ctl_color_2, gpr_$ctl_color_3, gpr_$ctl_color_4, or gpr_$mono_4, the default decomposition technique is gpr_$non_overlapping_tris. In this case, no action is necessary. If, however, the display type is anything else, the decomposition technique is set to gpr_$non_overlapping_tris with GPR_$PGON_DECOMP_TECHNIQUE.

In addition, this program uses the routine GPR_$RASTER_OP_PRIM_SET to establish the set of primitives that will be affected by the current raster operation. In this example, raster operations will affect lines and fills.

Pascal Example to Set the Decomposition Technique

```
PROGRAM example;

{ Required insert files }
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/error.ins.pas';
%list;


VAR
    size                : gpr_$offset_t;          { Size of the initial bitmap}
    init_bitmap         : gpr_$bitmap_desc_t;     { Descriptor of initial bitmap}
    hi_plane_id         : gpr_$plane_t;           { Highest plane in bitmap}
    delete_display      : BOOLEAN;
    status              : status_$t;              { Error code}
    disp_len            : INTEGER;                { Requested number of bytes to }
                                { be returned by gpr_$inq_disp_characteristics. }
    disp_len_returned   : INTEGER;                { Number of bytes returned }
    disp                : gpr_$disp_char_t;       { Returned display characteristics }
    acquire             : BOOLEAN;
    node_type           : gpr_$controller_type_t;       {The type of node the}
                                                        { application is running on }
    prim_set            : gpr_$rop_prim_set_t;          { The set of primitives that }
                                                        { raster operations will affect }

PROCEDURE check( st : status_$t );

    BEGIN
    if st.all<>status_$ok then
        pfm_$error_trap( status );
    END;



BEGIN  { Main }
    disp_len := 32;
    gpr_$inq_disp_characteristics (
                gpr_$direct     { Mode of operation }
                ,1              { Display unit or stream id of DM pad if any }
                ,disp_len       { Length of "disp" (the next argument) in BYTES }
                ,disp           { Returned display characteristics }
                ,disp_len_returned      { Number of BYTES actually written in "disp" }
                ,status                 { Returned status }
            );
    check(status);

    size.x_size := disp.x_window_size;
    size.y_size := disp.y_window_size;
    hi_plane_id := disp.n_planes - 1;
```

```
    gpr_$init(gpr_$direct,1,size,hi_plane_id,init_bitmap,status);
    check(status);

    node_type := disp.controller_type;
    CASE node_type of     { Determine the type of node the application is running on. }
        gpr_$ctl_mono_1,
        gpr_$ctl_mono_2,
        gpr_$ctl_color_1;
            gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);
    END;

    prim_set := [gpr_$rop_line, gpr_$rop_fill];
    gpr_$raster_op_prim_set(prim_set, status);      { Set raster operations for lines and fills. }
    check(status);

    DISCARD(gpr_$acquire_display(status));                { Acquire the display. }
    check(status);

    {*****************************************************}
    { Graphics application code goes here.}
    {*****************************************************}

    gpr_$release_display(status);
    check(status);

    gpr_$terminate(FALSE,status);
    check(status);

END.
```

FORTRAN Example to Set the Decomposition Technique

```
C Required insert files
%nolist
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'
%list

      INTEGER*2          mode
      INTEGER*2          maximum_display_length
      INTEGER*2          display_characteristics(28)
      INTEGER*2          actual_display_length
      INTEGER*4          status
      INTEGER*2          size(2)
      INTEGER*2          hi_plane_id
      INTEGER*4          init_bitmap
      INTEGER*2          node_type
      LOGICAL            acquire
      INTEGER*2          prim_set
      INTEGER*2          set_size

      mode = gpr_$direct
      maximum_display_length = 32

      CALL GPR_$INQ_DISP_CHARACTERISTICS
     +       (gpr_$direct
     +       ,stream_$stdout
     +       ,maximum_display_length
     +       ,display_characteristics
     +       ,actual_display_length
     +       ,status)
      CALL check(status)

      size(1) = display_characteristics(5)
      size(2) = display_characteristics(6)
      hi_plane_id = display_characteristics(15) - 1

      CALL gpr_$init(mode, stream_$stdout, size, hi_plane_id,
     +       init_bitmap,status)
      CALL check(status)

C If the node is not a DN3000, DN570, or DN580, set the decomposition
C technique to nonoverlapping_tris.
      node_type = display_characteristics(1)
      IF ((node_type .EQ. gpr_$ctl_mono_1) .OR.
     +       (node_type .EQ. gpr_$ctl_mono_2) .OR.
     +       (node_type .EQ. gpr_$ctl_color_1)) THEN
      CALL gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status)
      ENDIF
      CALL check(status)

C Set raster operations for lines and fills.
      setsize = 16
```

```
      CALL lib_$init_set(prim_set, setsize)
      CALL lib_$add_to_set(prim_set, setsize, gpr_$rop_line)
      CALL lib_$add_to_set(prim_set, setsize, gpr_$rop_fill)


C Acquire the display.
      acquire = gpr_$acquire_display(status)
      CALL check(status)

C ***************************************************
C Graphics application code goes here.
C ***************************************************

      CALL gpr_$release_display(status)
      CALL check(status)

      CALL gpr_$terminate(.FALSE.,status)
      CALL check(status)
      END


      SUBROUTINE check(st)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/error.ins.ftn'

      INTEGER*4  st

      IF (st .ne. status_$ok) then
          CALL pfm_$error_trap(st)
      ENDIF
      END
```

C Example to Set the Decomposition Technique

```
#nolist;   /* Required insert files */
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "/sys/ins/error.ins.c"
#list;

#define FALSE 0

gpr_$display_mode_t          mode = gpr_$direct;
short int                    maximum_display_length = 16;
gpr_$disp_char_t             display_characteristics;
short int                    actual_display_length;
status_$t                    status;

gpr_$offset_t                size_of_window;
gpr_$plane_t                 hi_plane_id;
gpr_$bitmap_desc_t           init_bitmap;

gpr_$controller_type_t       node_type;
gpr_$rop_prim_set_elems_t    prim_set;

short int                    setsize = 16;
void                         check(st)
status_$t                    st;
{
if (st.all != status_$ok)
    pfm_$error_trap(st);
}



main()
{
gpr_$inq_disp_characteristics (
    mode                  /* Mode of operation */
    ,1                    /* Display unit or stream id of DM pad if any */
    ,maximum_display_length      /* Length of display_characteristics in bytes. */
    ,display_characteristics     /* Returned display characteristics */
    ,actual_display_length       /* Returned number of bytes written
                                    to display_characteristics.*/
    ,status
    );
check(status);
size_of_window.x_size = display_characteristics.x_window_size;
size_of_window.y_size = display_characteristics.y_window_size;
hi_plane_id = display_characteristics.n_planes - 1;
 gpr_$init(mode,1,size_of_window,hi_plane_id,init_bitmap,status);
check(status);
/* Determine the type of node the application is running on. */
node_type = display_characteristics.controller_type;
switch (node_type)
```

```
        {
        case gpr_$ctl_mono_1  :
        case gpr_$ctl_mono_2  :
        case gpr_$ctl_color_1 :
             gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);
             check(status);
        }


    setsize = 16;
    lib_$init_set(prim_set, setsize);
    lib_$add_to_set(prim_set, setsize, gpr_$rop_line);
    lib_$add_to_set(prim_set, setsize, gpr_$rop_fill);
    gpr_$raster_op_prim_set(prim_set, status);      /* Set raster operations
                                                       for lines and fills. */
    check(status);

    gpr_$acquire_display(status);                   /* Acquire the display. */
    check(status);

/*****************************************************/
/***    Graphics application code goes here.              ***/
/*****************************************************/

    gpr_$release_display(status);
    check(status);

    gpr_$terminate(FALSE,status);
    check(status);

}
```

E.7.2. Sample Programs to Draw a Polygon

These sample programs demonstrate using the triangle technique and filling a polygon with the three available filling criteria. The polygon that is filled is displayed in Figure E–14. The arrows in the figure indicate the drawing direction from the starting point (indicated by a dot), and the numbers indicate winding numbers.

The programs begin by using a parity fill. In this way, only the odd numbered region is filled. Next, a nonzero fill is used. This filling criterion fills the polygon so that it is solid. Finally, the program uses a specific winding number fill. A winding number of 2 is used; this fills the three interior rectangles.

Different results can be achieved by changing the drawing direction of one or more of the polygons. In addition, this is an ideal polygon for testing the render–exact technique because it is rectilinear and axis aligned.

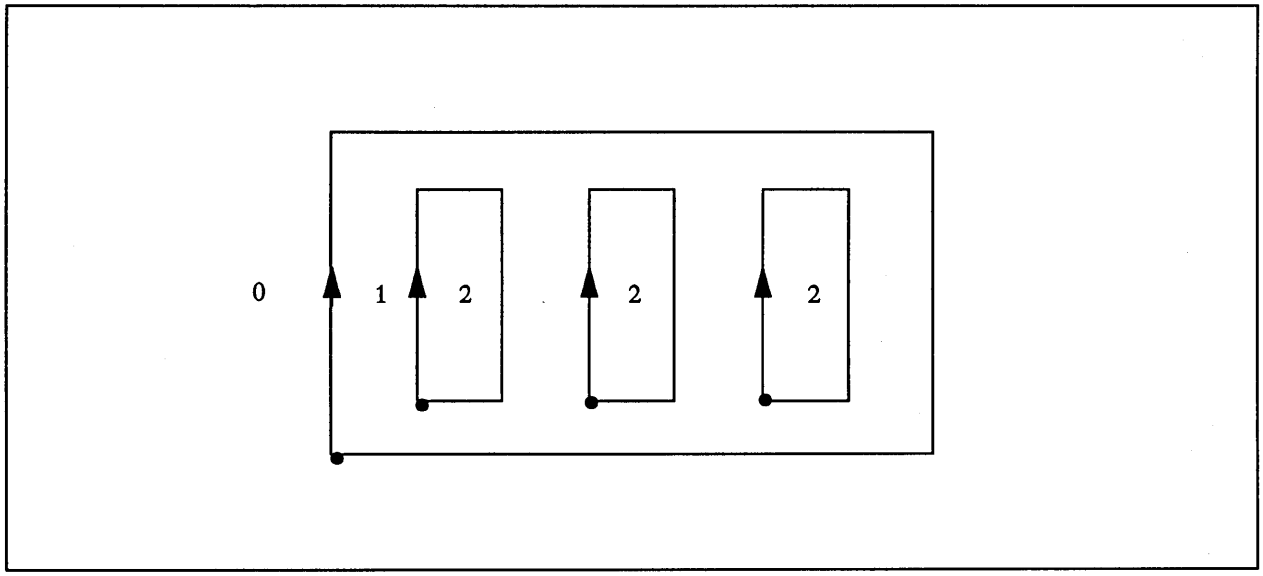


Figure E–14.

Pascal Example to Draw a Polygon

```
PROGRAM example;

{ Required insert files }
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;

const
one_second  = 250000;
five_seconds = 5 * one_second;

VAR
    size                : gpr_$offset_t;        { Size of the initial bitmap}
    init_bitmap         : gpr_$bitmap_desc_t;   { Descriptor of initial bitmap}
    hi_plane_id         : gpr_$plane_t;         { Highest plane in bitmap}
```

```
    delete_display          : BOOLEAN;
    status                  : status_$t;            { Error code}
    disp_len                : INTEGER;              { Requested number of bytes to }
                                                    { be returned by gpr_$inq_disp_characteristics. }
    disp_len_returned       : INTEGER;              { Number of bytes returned }
    disp                    : gpr_$disp_char_t;     { Returned display characteristics }
    acquire                 : BOOLEAN;
    node_type               : gpr_$controller_type_t;       { Node type }
    prim_set                : gpr_$rop_prim_set_t; { The set of primitives that raster
                                                     operations will affect.}
    x, y                    : gpr_$coordinate_t;
    x_array, y_array        : gpr_$coordinate_array_t;
    list_size               : integer;
    t_list                  : ARRAY [1..30] OF gpr_$triangle_t;
    n_triangles             : integer;
    winding_set             : gpr_$triangle_fill_criteria_t;
    n_positions             : integer;
    pixel_array             : ARRAY[1..1] OF integer32;
    window                  : gpr_$window_t;
    pause                   : time_$clock_t;

PROCEDURE check( st : status_$t );

    BEGIN
    if st.all<>status_$ok then
        pfm_$error_trap( status );
    END;



BEGIN  { Main }
    disp_len := 30;
    gpr_$inq_disp_characteristics (
                gpr_$direct             { Mode of operation }
                ,1                      { Display unit or stream id of DM pad if any }
                ,disp_len               { Length of "disp" (the next argument) in
                                                                        BYTES }
                ,disp                   { Returned display characteristics }
                ,disp_len_returned      { Number of BYTES actually written in "disp" }
                ,status                 { Returned status }
            );
    check(status);

    size.x_size := disp.x_window_size;
    size.y_size := disp.y_window_size;
    hi_plane_id := disp.n_planes - 1;

    gpr_$init(gpr_$direct,1,size,hi_plane_id,init_bitmap,status);
    check(status);

    node_type := disp.controller_type;
```

```
CASE node_type of              { Determine the type of node the application is running on. }
    gpr_$ctl_mono_1,
    gpr_$ctl_mono_2,
    gpr_$ctl_color_1,
        gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);
END;

gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);

prim_set := [gpr_$rop_line, gpr_$rop_fill];
gpr_$raster_op_prim_set(prim_set, status);      { Set the  raster operations }
                                                { for lines and fills. }
check(status);

WITH window do
BEGIN
    window_base.x_coord := 200;
    window_base.y_coord := 200;
    window_size.x_size := 1;
    window_size.y_size := 1;
END;

DISCARD(gpr_$acquire_display(status));              { Acquire the display. }
check(status);

gpr_$read_pixels(window,pixel_array, status);
check(status);

x := 50;
y := 600;
gpr_$start_pgon(x, y, status);
check(status);

x_array[1] := 50;       { Draw clockwise.}
y_array[1] := 100;

x_array[2] := 750;
y_array[2] := 100;

x_array[3] := 750;
y_array[3] := 600;
n_positions := 3;

gpr_$pgon_polyline(x_array, y_array,n_positions, status);
check(status);

x := 150;
y := 500;
gpr_$start_pgon(x, y, status);
check(status);

x_array[1] := 150;      { Draw clockwise.}
y_array[1] := 200;
```

```
x_array[2] := 250;
y_array[2] := 200;

x_array[3] := 250;
y_array[3] := 500;
n_positions := 3;

gpr_$pgon_polyline(x_array, y_array, n_positions, status);
check(status);
x := 350;
y := 500;
gpr_$start_pgon(x, y, status);
check(status);

x_array[1] := 350;      { Draw clockwise.}
y_array[1] := 200;

x_array[2]  := 450;
y_array[2]  := 200;

x_array[3] := 450;
y_array[3] := 500;
n_positions := 3;

gpr_$pgon_polyline(x_array, y_array, n_positions, status);
check(status);

x := 550;
y := 500;
gpr_$start_pgon(x, y, status);
check(status);

x_array[1] := 550;      { Draw clockwise.}
y_array[1] := 200;

x_array[2] := 650;
y_array[2] := 200;

x_array[3] := 650;
y_array[3] := 500;
n_positions := 3;

gpr_$pgon_polyline(x_array, y_array, n_positions, status);
check(status);

winding_set.wind_type := gpr_$parity;
gpr_$set_triangle_fill_criteria(winding_set,status);
check(status);

list_size := 30;
gpr_$close_return_pgon_tri(list_size, t_list, n_triangles, status);
check(status);
```

```
    { Draw the triangles with a parity fill. }
    gpr_$multitriangle(t_list, n_triangles, status);
    check(status);

    {Keep image displayed on screen for five seconds.}
    pause.low32 := five_seconds;
    pause.high16 := 0;
    time_$wait(time_$relative, pause, status);

    gpr_$clear(pixel_array[1], status);
    check(status);

    winding_set.wind_type := gpr_$nonzero;
    gpr_$set_triangle_fill_criteria(winding_set,status);
    check(status);

    { Draw the triangles with a nonzero fill. }
    gpr_$multitriangle(t_list, n_triangles, status);
    check(status);

    time_$wait(time_$relative,pause,status);
    check(status);

    gpr_$clear(pixel_array[1], status);
    check(status);

    winding_set.wind_type := gpr_$specific;
    winding_set.winding_no := 2;
    gpr_$set_triangle_fill_criteria(winding_set,status);
    check(status);

    { Draw the triangles with a specific winding number fill. }
    gpr_$multitriangle(t_list, n_triangles, status);
    check(status);

    time_$wait(time_$relative,pause,status);
    check(status);

    gpr_$release_display(status);
    check(status);

    gpr_$terminate(FALSE,status);
    check(status);

END.
```

FORTRAN Example to Draw a Polygon

```
%nolist
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'
%include '/sys/ins/error.ins.ftn'
%include '/sys/ins/time.ins.ftn'
%list

C     Timer Constants
      INTEGER*4 one_second
      INTEGER*4 five_seconds
      PARAMETER (one_second = 250000)
      PARAMETER (five_seconds = 1250000)

      INTEGER*2           pause(3)
      INTEGER*4           pause_low32
      INTEGER*2             pause_high16
      EQUIVALENCE (pause(1), pause_high16)
      EQUIVALENCE (pause(2), pause_low32)

      INTEGER*2           dev
      INTEGER*2           size(2)
      INTEGER*4           init_bitmap
      INTEGER*2           hi_plane_id
      LOGICAL             delete_display
      LOGICAL             acquire
      INTEGER*4           status

C     Requested number of bytes to be returned by gpr_$inq_disp_characteristics.
      INTEGER*2           disp_len
C     Number of bytes actually returned.
      INTEGER*2           disp_len_ret
      INTEGER*2           disp(30)
      INTEGER*2           node_type

C     The set of primitives that raster operations will affect.
      INTEGER*2           prim_set
      INTEGER*2           x,y
      INTEGER*2           x_array(50), y_array(50)
      INTEGER*2           n_positions
      INTEGER*2           setsize
      INTEGER*2           winding_set(2)
      INTEGER*2           window(4)
      INTEGER*4           pixel_array(1)
      INTEGER*2           list_size
      INTEGER*2           n_triangles
      INTEGER*2           t_list(175)


C Main subprogram
      dev = 1
```

```
      disp_len = 32
C Get the device characteristics.
      CALL gpr_$inq_disp_characteristics
     +     ( gpr_$direct
     +       , dev
     +       , disp_len
     +       , disp
     +       , disp_len_ret
     +       , status)
      CALL check(status)

      hi_plane_id = disp(15) - 1
      size(1) = disp(5)
      size(2) = disp(6)
      node_type = disp(1)

C Initialize GPR.
      CALL gpr_$init
     +     ( gpr_$direct
     +       , dev
     +       , size
     +       , hi_plane_id
     +       , init_bitmap
     +       , status)
      CALL check(status)

C If the device is not a DN3000, DN570, or DN580 then set the decomposition
C technique to non_overlapping_tris.
      IF ((node_type .NE. gpr_$ctl_mono_4) .OR.
     +       (node_type .NE. gpr_$ctl_color_4) .OR.
     +       (node_type .NE. gpr_$ctl_color_2) .OR.
     +       (node_type .NE. gpr_$ctl_color_3))
     +  CALL gpr_$pgon_decomp_technique
     +       ( gpr_$non_overlapping_tris
     +       , status)

      CALL check(status)

      setsize = 16
      CALL lib_$init_set(prim_set, setsize)
      CALL lib_$add_to_set(prim_set, setsize, gpr_$rop_line)
      CALL lib_$add_to_set(prim_set, setsize, gpr_$rop_fill)

C Draw each figure in a clockwise direction.
      x = 50
      y = 600
      CALL gpr_$start_pgon(x, y, status)
      CALL check(status)

      x_array(1) = 50
      y_array(1) = 100
      x_array(2) = 750
      y_array(2) = 100
```

```
x_array(3) = 750
y_array(3) = 600
n_positions = 3
CALL gpr_$pgon_polyline
+     ( x_array
+       , y_array
+       , n_positions
+       , status)
CALL check(status)

x = 150
y = 500
x_array(1) = 150
y_array(1) = 200
x_array(2) = 250
y_array(2) = 200
x_array(3) = 250
y_array(3) = 500
CALL gpr_$start_pgon(x, y, status)
CALL check(status)

CALL gpr_$pgon_polyline
+       (x_array
+       , y_array
+       , n_positions
+       , status)
CALL check(status)

x = 350
y = 500
x_array(1) = 350
y_array(1) = 200
x_array(2) = 450
y_array(2) = 200
x_array(3) = 450
y_array(3) = 500
CALL gpr_$start_pgon(x, y, status)
CALL check(status)
CALL gpr_$pgon_polyline
+       ( x_array
+       , y_array
+       , n_positions
+       , status)
CALL check(status)

x = 550
y = 500
x_array(1) = 550
y_array(1) = 200
x_array(2) = 650
y_array(2) = 200
x_array(3) = 650
y_array(3) = 500
```

```
      CALL gpr_$start_pgon(x, y, status)
      CALL check(status)
      CALL gpr_$pgon_polyline
     +     ( x_array
     +     , y_array
     +     , n_positions
     +     , status)
      CALL check(status)

C Use a parity filling algorithm.
      winding_set(1) = gpr_$parity
      CALL gpr_$set_triangle_fill_criteria(winding_set,status)
      CALL check(status)

      list_size = 25
      CALL gpr_$close_return_pgon_tri
     +     ( list_size
     +     , t_list
     +     , n_triangles
     +     , status)
      CALL check(status)

C Acquire the display.
      acquire = gpr_$acquire_display(status)
      CALL check(status)

      window(1) = 200
      window(2) = 200
      window(3) = 1
      window(4) = 1
      CALL gpr_$read_pixels(window,pixel_array, status)
      CALL check(status)

      CALL gpr_$multitriangle
     +     ( t_list
     +     , n_triangles
     +     , status)
      CALL check(status)

      pause_low32 = five_seconds
      pause_high16 = 0
      CALL time_$wait(time_$relative, pause, status)
      CALL check(status)

C Clear the display.
      CALL gpr_$clear(pixel_array(1), status)
      CALL check(status)

C Use a nonzero filling algorithm.
      winding_set(1) = gpr_$nonzero
      CALL gpr_$set_triangle_fill_criteria(winding_set,status)
      CALL check(status)
```

```
      CALL gpr_$multitriangle
     +        ( t_list
     +        , n_triangles
     +        , status)
      CALL check(status)

      pause_low32 = five_seconds
      pause_high16 = 0
      CALL time_$wait(time_$relative, pause, status)
      CALL check(status)

      CALL gpr_$clear(pixel_array(1), status)
      CALL check(status)

C Use a specific winding number filling algorithm.
      winding_set(1) = gpr_$specific
      winding_set(2) = 2
      CALL gpr_$set_triangle_fill_criteria(winding_set,status)
      CALL check(status)

      CALL gpr_$multitriangle
     +        ( t_list
     +        , n_triangles
     +        , status)
      CALL check(status)

      pause_low32 = five_seconds
      pause_high16 = 0
      CALL time_$wait(time_$relative, pause, status)
      CALL check(status)

      CALL gpr_$release_display(status)
      CALL check(status)
      STOP
      END


      SUBROUTINE check(st)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'
%include '/sys/ins/error.ins.ftn'


      INTEGER*4  st

      IF (st .ne. status_$ok) CALL pfm_$error_trap(st)

      RETURN
      END
```

C Example to Draw a Polygon

```
/* Required insert files */
#nolist;
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/time.ins.c"
#list;

#define t    5  /* Pause for this many seconds before displaying next image. */
#define time_to_pause (250000 * t)
#define FALSE 0

gpr_$offset_t                   size_of_initial_bitmap;
gpr_$bitmap_desc_t              description_of_initial_bitmap;
gpr_$plane_t                    highest_plane_in_bitmap;
short int                       delete_display;
short int                       acquire;
status_$t                       status;
/* Requested number of bytes to be returned into display_characteristics: */
short int                       display_length = 30;
gpr_$disp_char_t                display_characteristics;
/* Number of bytes actually returned: */
short int                       display_length_returned;
gpr_$controller_type_t          node_type;
/* The set of primitives that raster operations will affect: */
gpr_$rop_prim_set_elems_t       prim_set;
gpr_$coordinate_t               x,y;
gpr_$coordinate_array_t         x_array, y_array;
short int                       list_size;
gpr_$triangle_t                 t_list[30];
short int                       n_triangles;
gpr_$triangle_fill_criteria_t   winding_set;
short int                       n_positions = 3;
long int                        pixel_array[1];
gpr_$window_t                   window;
time_$clock_t                   pause;
short int                       setsize = 16;



void check(st)
status_$t  st;
{
if (st.all != status_$ok)
     pfm_$error_trap( st );
}



main()
```

```
{
    gpr_$inq_disp_characteristics (
            gpr_$direct
            ,1
            ,display_length
            ,display_characteristics
            ,display_length_returned
            ,status
    );
    check(status);
/* Use the information returned into display_characteristics: */
    size_of_initial_bitmap.x_size = display_characteristics.x_window_size;
    size_of_initial_bitmap.y_size = display_characteristics.y_window_size;
    highest_plane_in_bitmap       = display_characteristics.n_planes - 1;
   gpr_$init(gpr_$direct,
            1,
            size_of_initial_bitmap,
            highest_plane_in_bitmap,
            description_of_initial_bitmap,
            status);
    check(status);
/* Determine the type of node the application is running on: */
    node_type = display_characteristics.controller_type;
    switch (node_type)
        {
        case gpr_$ctl_mono_1  :
        case gpr_$ctl_mono_2  :
        case gpr_$ctl_mono_4  :
        case gpr_$ctl_color_1 :
        case gpr_$ctl_color_4 :
        gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);
        check(status);
        }

    gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);
    check(status);

    setsize = 16;
    lib_$init_set(prim_set, setsize);
    lib_$add_to_set(prim_set, setsize, gpr_$rop_line);
    lib_$add_to_set(prim_set, setsize, gpr_$rop_fill);

    gpr_$raster_op_prim_set(prim_set, status); /* Set the raster operations */
    /* for lines and fills. */
    check(status);
```

```
    window.window_base.x_coord = 200;
    window.window_base.y_coord = 200;
    window.window_size.x_size = 1;
    window.window_size.y_size = 1;

    gpr_$acquire_display(status);    /* Acquire the display. */
    check(status);

    gpr_$read_pixels(window, pixel_array, status);
    check(status);

/* Draw each figure clockwise.*/

    x = 50;          y = 600;
    x_array[0] = 50;
    y_array[0] = 100;
    x_array[1] = 750;
    y_array[1] = 100;
    x_array[2] = 750;
    y_array[2] = 600;
    gpr_$start_pgon(x, y, status);
    check(status);
    gpr_$pgon_polyline(x_array, y_array, n_positions, status);
    check(status);

    x = 150;
    y = 500;
    x_array[0] = 150;
    y_array[0] = 200;
    x_array[1] = 250;
    y_array[1] = 200;
    x_array[2] = 250;
    y_array[2] = 500;
    gpr_$start_pgon(x, y, status);
    check(status);
    gpr_$pgon_polyline(x_array, y_array, n_positions, status);
    check(status);

    x = 350;
    y = 500;
    x_array[0] = 350;
    y_array[0] = 200;
    x_array[1] = 450;
    y_array[1] = 200;
    x_array[2] = 450;
    y_array[2] = 500;
    gpr_$start_pgon(x, y, status);
    check(status);
    gpr_$pgon_polyline(x_array, y_array, n_positions, status);
    check(status);

    x = 550;
    y = 500;
```

```
    x_array[0] = 550;
    y_array[0] = 200;
    x_array[1] = 650;
    y_array[1] = 200;
    x_array[2] = 650;
    y_array[2] = 500;
    gpr_$start_pgon(x, y, status);
    check(status);
    gpr_$pgon_polyline(x_array, y_array, n_positions, status);
    check(status);

    winding_set.wind_type = gpr_$parity;
    gpr_$set_triangle_fill_criteria(winding_set,status);
    check(status);

    list_size = 30;
    gpr_$close_return_pgon_tri(list_size, t_list, n_triangles, status);
    check(status);

/* Draw the triangles with a parity fill. */
    gpr_$multitriangle(t_list, n_triangles, status);
    check(status);

/* Keep image displayed on screen. */
    pause.low32  = time_to_pause;
    pause.high16 = 0;
    time_$wait(time_$relative, pause, status);
    check(status);

    gpr_$clear(pixel_array[0], status);
    check(status);

    winding_set.wind_type = gpr_$nonzero;
    gpr_$set_triangle_fill_criteria(winding_set,status);
    check(status);

/* Draw the triangles with a nonzero fill. */
    gpr_$multitriangle(t_list, n_triangles, status);
    check(status);

    time_$wait(time_$relative,pause,status);
    check(status);

    gpr_$clear(pixel_array[0], status);
    check(status);

    winding_set.wind_type = gpr_$specific;
    winding_set.winding_no = 2;
    gpr_$set_triangle_fill_criteria(winding_set,status);
    check(status);

/* Draw the triangles with a specific winding number fill. */
    gpr_$multitriangle(t_list, n_triangles, status);
```

```
    check(status);

    time_$wait(time_$relative,pause,status);
    check(status);

    gpr_$release_display(status);
    check(status);

    gpr_$terminate(FALSE,status);
    check(status);
}
```

Appendix F

Display Configurations

This appendix describes the current DOMAIN hardware display configurations. It updates the configuration information provided in Chapter 8.

F.1. Monochromatic Display Configurations

There are three monochromatic display configurations. These configurations have one plane of display memory, and the only possible pixel values are zero and 1.

On displays that are not inverted a pixel value of 0 indicates that the pixel is black; a pixel value of 1 indicates that the pixel is white. On inverted displays that simulate a color map in software (DN100, DN3XX, and DN4XX), pixels with a value of zero are black, and pixels with a value of one are white. This is true regardless of the contents of locations zero and one in the color map. On inverted displays that have a color map in hardware (DN3000), a pixel value is used as an index into the color map. The color of the pixel (black or white) is determined by the value in the color map. This is consistent with the way pixel values function on color displays.

You can determine whether a monochromatic device simulates an inverted color map or has a color map in hardware with GPR_$INQ_DISP_CHARACTERISTICS. Invert, a value returned in the datatype gpr_$disp_char_t, has the following three possible values: gpr_$no_invert, gpr_$invert_simulate and gpr_$invert_hardware.

The available monochromatic display configurations are listed in Table F-1.

Table F-1. Monochromatic Display Configurations

MODEL	DIMENSIONS	
	Visible Display	Hidden Display
DN100	800 x 1024	224 x 1024
DN3XX	1024 x 800	1024 x 224
DN4XX	1024 x 800	1024 x 224
DN3000	1280 x 1024	768 x 224

F.2. Four–Plane Color Configurations

There are two four–plane color configurations. Bitmaps on devices with a four–plane color configuration can have one to four planes. The planes are numbered 0 – 3. Pixel values on four–plane configurations can have one to four bits: one bit for each plane. This allows a maximum of sixteen colors to be simultaneously displayed.

The available four–plane color display configurations are listed in Table F–2.

Table F–2. Four–Plane Color Configurations

MODEL	DIMENSIONS		
	Visible Display	Hidden Display	No. of Colors
DN3000	1024 x 800	1024 x 224	16
DN550/560	1024 x 800	1024 x 1024	16
DN6XX	1024 x 1024	1024 x 1024	16

F.2. Eight–Plane Color Configurations

There are three eight–plane color configurations. Bitmaps on devices with an eight–plane configuration can have one to eight planes. The planes are numbered 0 – 7. Pixel values on eight–plane configurations can have one to eight bits: one bit for each plane. This allows a maximum of 256 colors to be simultaneously displayed on eight planes.

The available eight–plane color display configurations are listed in Table F–3.

Table F–3. Eight–Plane Color Configurations

MODEL	DIMENSIONS		
	Visible Display	Hidden Display	No. of Colors
DN550/560	1024 x 800	1024 x 1024	256
DN570	1024 x 800	1024 x 219	256
DN6XX	1024 x 1024	1024 x 1024	256
DN580	1280 x 1024	224 x 1024	256

F.3. Limitations

The following GPR routines are only supported on DN550s, DN560s, and DN6XXs,.

GPR_$REMAP_COLOR_MEMORY_1

GPR_$COLOR_ZOOM

GPR_$SET_IMAGING_FORMAT

GPR_$INQ_IMAGING_FORMAT

GPR_$COLOR_ZOOM is supported on DN580s for zoom values of both x and y equal to one or two.

The maximum dimensions of a hidden-display-memory bitmap on all devices except the DN570 is 224 x 224. On the DN570, the dimensions of the largest hidden-display-memory bitmap is 224 x 219.

Index

A

Acquiring the display 4-19

Attributes, changing 6-7

Attributes, list of 6-5

Auto refresh 4-22

B

Bit block transfer 7-3, 7-5

Bitmap 2-1, 2-2

Bitmap attributes 6-3

Bitmap location 6-1

Bitmap size 6-1

Bitmap structure 6-1

Bitmaps in borrow mode 6-1

Bitmaps in direct mode 6-2

Bitmaps in frame mode 6-1

Bitmaps in no-display mode 6-2

Bitmaps on disk 7-1

Borrow-display mode 2-4

Borrow-nc mode 2-4

Button event 5-3

Button events 5-3

C

C examples C-1

Clipping 6-5

Color Display configurations 8-1

Color display formats 8-9

Color entry 8-8

Color format 8-3

Color map 8-3, 8-5, 8-7, 8-9

Color value 8-7

Color zoom 8-11

Coordinate origin 4-1, 6-5

Coordinate system 4-1

Creating attribute blocks 6-3

Current attribute block 6-3

Current bitmap 6-3

Current position 4-1

Cursor 5-1

Cursor control in display mode 5-1

Cursor control in frame mode 5-1

D

Direct mode 2-5

Disabled input events 5-3

Display controller 2-2

Display devices 2-2

Displaying colors 8-3

Draw value 6-5

E

Enabled input events 5-3

Entered window events 5-5

Error reporting 3-3

Event reporting 5-3

Event types 5-2

External bitmap file 9-1

External bitmaps 6-4, 7-2

F

Fill examples 4-13

Fill pattern 6-6

Fill routines 4-12

Fill value 6-5

FORTRAN examples D-1

Frame buffer 2-1

Frame mode 2-5

G

GMF error messages 9-2

GMF insert files 9-1

GMF programming example 9-2

GMF_$OPEN 9-1

GMF_$RESTORE_PLANE 9-1

GMF_COPY_PLANE 9-1

GPR_$ADDITIVE_BLT 7-4

GPR_$ALLOCATE_ATTRIBUTE_BLOCK 6-3

GPR_$ALLOCATE_BITMAP 7-1

GPR_$ARC_3P 4-2

GPR_$BIT_BLT 7-4

GPR_$CIRCLE 4-2

GPR_$CIRCLE_FILLED 4-12

GPR_$CLOSE_FILL_PGON 4-13

GPR_$CLOSE_RETURN_PGON 4-13

GPR_$COND_EVENT_WAIT 5-4, 5-5

GPR_$DISABLE_INPUT 5-4

GPR_$DRAW_BOX 4-2

GPR_$ENABLE_INPUT 5-4

Gpr_$entered_window 5-3

GPR_$EVENT_WAIT 5-4, 5-5

GPR_$GET_EC 5-4

GPR_$INQ_CHARACTER_WIDTH 4-23

GPR_$INQ_CONSTRAINTS 6-9
GPR_$INQ_COORDINATE_ORIGIN 6-9
GPR_$INQ_DRAW_VALUE 6-9
GPR_$INQ_FILL_BACKGROUND_VALUE 6-9
GPR_$INQ_FILL_PATTERN 6-9
GPR_$INQ_FILL_VALUE 6-9
GPR_$INQ_HORIZONTAL 4-23
GPR_$INQ_LINE_PATTERN 6-9
GPR_$INQ_LINESTYLE 6-10
GPR_$INQ_RASTER_OPS 6-10
GPR_$INQ_SPACE_SIZE 4-23
GPR_$INQ_TEXT 4-24, 6-10
GPR_$INQ_TEXT_EXTENT 4-24
GPR_$INQ_TEXT_OFFSET 4-24, 6-10
GPR_$INQ_TEXT_PATH 4-24
GPR_$INQ_TEXT_VALUES 4-24, 6-10
GPR_$INQ_WINDOW_ID 5-5
Gpr_$left_window 5-3
GPR_$LINE 4-2
GPR_$LOAD_FONT_FILE 4-23
GPR_$MULTILINE 4-2
GPR_$MULTITRAPEZOID 4-12
GPR_$OPEN_BITMAP_FILE 7-2
GPR_$PGON_POLYLINE 4-12
GPR_$POLYLINE 4-2
GPR_$RECTANGLE 4-12
GPR_$REPLICATE_FONT 4-23
GPR_$SET_ATTRIBUTE_BLOCK 6-4
GPR_$SET_BITMAP 7-1
GPR_$SET_CHARACTER_WIDTH 4-23
GPR_$SET_CLIP_WINDOW 6-8
GPR_$SET_CLIPPING_ACTIVE 6-8
GPR_$SET_COLOR_MAP 8-9
GPR_$SET_COORDINATE_ORIGIN 6-8
GPR_$SET_CURSOR_ACTIVE 5-1
GPR_$SET_CURSOR_ORIGIN 5-1
GPR_$SET_CURSOR_PATTERN 5-1
GPR_$SET_CURSOR_POSITION 5-1
GPR_$SET_DRAW_VALUE 6-8
GPR_$SET_FILL_BACKGROUND_VALUE 6-8
GPR_$SET_FILL_PATTERN 6-9
GPR_$SET_FILL_VALUE 6-9
GPR_$SET_HORIZONTAL 4-23
GPR_$SET_INPUT_SID 5-5
GPR_$SET_LINE_PATTERN 6-9
GPR_$SET_LINESTYLE 6-9
GPR_$SET_PLANE_MASK 6-9
GPR_$SET_RASTER_OP 6-9
GPR_$SET_SPACE_SIZE 4-23
GPR_$SET_TEXT_BACKGROUND 4-24
GPR_$SET_TEXT_BACKGROUND_VALUE 6-9
GPR_$SET_TEXT_FONT 4-23, 6-9
GPR_$SET_TEXT_PATH 4-24
GPR_$SET_TEXT_VALUE 4-24, 6-9
GPR_$SET_WINDOW_ID 5-4, 5-5
GPR_$SPLINE_CUBIC 4-2
GPR_$SPLINE_CUBIC_X 4-2
GPR_$SPLINE_CUBIC_Y 4-2
GPR_$START_PGON 4-12
GPR_$TEXT 4-24
GPR_$TRAPEZOID 4-12
GPR_$TRIANGLE 4-12
GPR_$UNLOAD_FONT_FILE 4-23
GPR_PIXEL_BLT 7-4
Graphics map file 9-1
Grey-scale 8-9

H

Hidden-display-memory bitmaps 6-4, 7-1, 7-2

I

Imaging display formats 8-10
Imaging format 8-1
Imaging formats 8-10
Initial bitmap 6-1
Initial bitmap location 6-1
Initializing GPR in FORTRAN 3-5
Initializing GPR in C 3-6
Initializing GPR in Pascal 3-4
Initializing the graphics package 3-1
Input operations 5-2, 5-4
Input routines 5-3
Insert files 3-1

K

Keyset 5-3
Keystroke event 5-3

L

Line drawing routines 4-2
Line style 6-6
Line-drawing examples 4-3
Listing attributes 6-5
Locator events 5-3
Locator stop event 5-3

M

Main memory bitmaps 6-4

Main-memory bitmaps 7-1

Making attribute blocks current 6-4

Masks 7-4

Monochromatic color map 8-9

N

No-display mode 2-6

O

Operation mode 2-3

P

Pixel array 8-9

Pixel value 2-1, 8-3, 8-5

Pixel values 8-9

Plane mask 6-6

Plane masks 7-4

Primary colors 8-3

R

Raster operation 6-7

Raster operation truth table 6-7

Raster operations 7-5

Raster table 6-7

Refresh procedure 4-22, 5-5

Releasing the display 4-19

Retrieving attributes 6-9

S

Scan line 2-1

T

Terminating GPR 3-3

Text background value 6-6

Text font 6-6

Text routines 4-23

Text valule 6-6

Three-board configuration 8-1

Two-board configuration 8-1

V

Variables 3-1

W

Window id 5-4

Window transition event 5-3