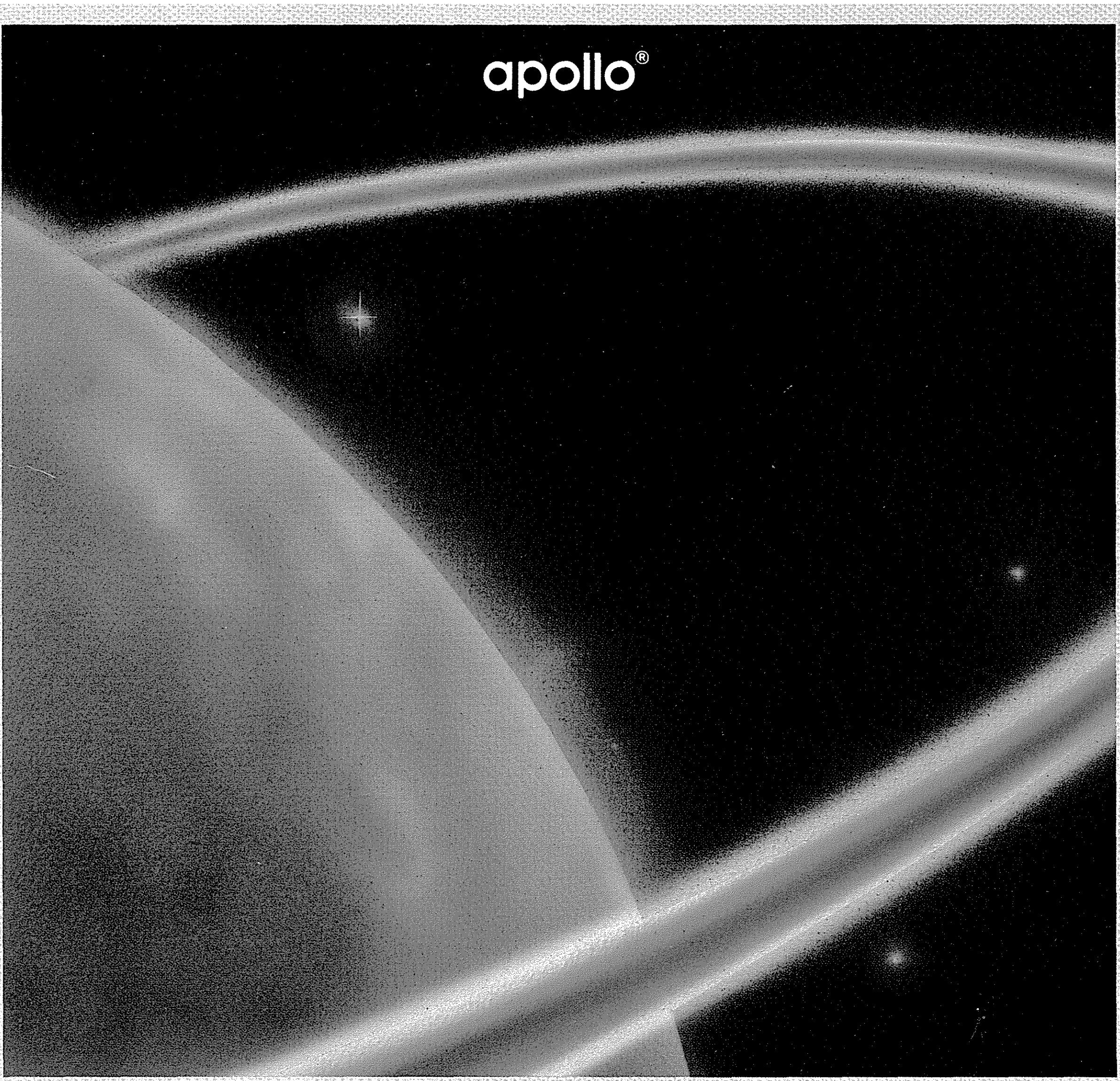
apollo®



D O M A I N

Order No. 005808
Revision 01

Programming with Domain Graphics Primitives

Programming with Domain Graphics Primitives

Order No. 005808
Revision 01

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: July, 1985
First Revision: February, 1987
Latest Printing: August, 1987

This document was produced using the Interleaf Workstation Publishing Software (WPS), and the InterCAD 2040 Electronic Illustrating System, a product of InterCAD Corporation. Interleaf and WPS are trademarks of Interleaf, Inc.

Apollo and Domain are registered trademarks of Apollo Computer Inc.

UNIX is a registered trademark of AT&T in the USA and other countries. Ada is a registered trademark of U.S. Government (Ada Joint Program Office).

3DGR, Aegis, D3M, DGR, Domain/Access, Domain/Ada, Domain/Bridge, Domain/C, Domain/ComController, Domain/CommonLISP, Domain/CORE, Domain/Debug, Domain/DFL, Domain/Dialogue, Domain/DQC, Domain/IX, Domain/Laser-26, Domain/LISP, Domain/PAK, Domain/PCC, Domain/PCC-Remote, Domain/PCI, Domain/SNA, Domain/X.25, DPSS/Mail, DSEE, FPX, GMR, GPR, GSR, NCK, NCS, Network Computing Kernel, Network Computing System, Open System Toolkit, OST, Personal Workstation, and Series 3000 are trademarks of Apollo Computer Inc.

Apollo Computer Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should in all cases consult Apollo Computer Inc. to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF APOLLO COMPUTER INC. HARDWARE PRODUCTS AND THE LICENSING OF APOLLO COMPUTER INC. SOFTWARE PROGRAMS CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN APOLLO COMPUTER INC. AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY APOLLO COMPUTER INC. FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY BY APOLLO COMPUTER INC. WHATSOEVER.

IN NO EVENT SHALL APOLLO COMPUTER INC. BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATING TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF APOLLO COMPUTER INC. HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

THE SOFTWARE PROGRAMS DESCRIBED IN THIS DOCUMENT ARE CONFIDENTIAL INFORMATION AND PROPRIETARY PRODUCTS OF APOLLO COMPUTER INC. OR ITS LICENSORS.

Preface

This manual describes the Domain graphics primitive resource (GPR) system. It shows how to use graphics primitive routines in applications programs.

We've organized this manual as follows:

Chapter 1	Presents an overview of GPR and its capabilities.
Chapter 2	Shows you how to get started writing GPR programs. It also explains the various display modes that GPR can run in.
Chapter 3	Describes how to use GPR routines to display geometric shapes such as lines, arcs, splines, circles, and polygons.
Chapter 4	Demonstrates how to print text in a graphics program.
Chapter 5	Details bitmaps.
Chapter 6	Details color on the GPR system.
Chapter 7	Explains how GPR analyzes input events such as moving the mouse or typing keys on the keyboard.
Chapter 8	Describes the cursor.
Chapter 9	Describes clip windows and plane masks.
Chapter 10	Details direct mode (which lets you run GPR programs in a window).
Chapter 11	Explains raster operations.
Appendix A	Illustrates the keyboards and contains keyboard charts.
Appendix B	Presents the Pascal program examples used in the manual translated into C.
Appendix C	Presents the Pascal program examples used in the manual translated into FORTRAN.
Appendix D	Lists the amount of visible and hidden display memory for each node type, and also lists the number of planes available for each node type.
Appendix E	Describes decomposition techniques.
Appendix F	Describes imaging mode on the DN550/560/600/660 nodes.
Glossary	Provides definitions for some of the terms used in this manual.

Related Manuals

This is one of three manuals that compose the GPR documentation set. The other two are

- *Domain Graphics Primitive Resource Call Reference* (007194) which details the syntax for every GPR call.

- *Domain GPR Quick Reference* (010430) which provides terse descriptions of each GPR call.

In addition, the following manuals may also prove useful to GPR programmers:

- *Programming with General System Calls* (005506) explains pad calls and how to emulate Pascal data types (for example, sets) in C and FORTRAN.

- *Domain C Language Reference* (002093) explains how to program in the Domain C language.

- *Domain Pascal Language Reference* (000792) explains how to program in the Domain Pascal language.

- *Domain FORTRAN Language Reference* (000530) explains how to program in the Domain FORTRAN language.

On-Line Examples

Because this manual is being published between software releases, we will not be able to provide customers with the on-line versions of the sample programs until the next software release. For details regarding these programs, look in the GPR section of the next standard release notes.

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. By using these formal channels you make it easy for us to respond to your comments.

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

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

For your documentation comments, we've included a Reader's Response form at the back of each manual.

Documentation Conventions

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

UPPERCASE	Bold, uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.
`output`	Typewriter font words are used for sample programs and command examples.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down <CTRL> while typing the character.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Contents

Chapter 1 Overview of GPR

Chapter 2 Getting Started with GPR Programming

Chapter 3 Drawing Figures

Chapter 4 Printing Text

Bitmaps 5–1

Chapter 6 Color

Chapter 7 Input Events

Chapter 8 The Cursor

Chapter 9 Clip Windows and Plane Masks

Chapter 10 Direct Mode

Chapter 11 Raster Operations

Appendix A Keyboards

Appendix B C Programs

Appendix C FORTRAN Programs

Appendix D Node–Dependent Data

Appendix E Decomposition and Rendering Techniques

Appendix F Imaging Mode

Glossary

Illustrations

Tables

Chapter 1

Overview of GPR

The Graphics Primitives Resource package (GPR for short) is a set of well over 100 routines callable from your Domain FORTRAN, Domain Pascal, or Domain C program. Although these routines are relatively primitive, you can use combinations of them to write powerful, high-performance graphics applications.

These routines are a standard piece of Domain software. The system stores these routines in an installed library at pathname /lib/gprlib. An installed library is an object file whose routines are accessed at runtime. In other words, you do not have to load these routines at bind time; the system will load them automatically at runtime.

1.1 Overview of GPR Capabilities

GPR can

- Draw lines, arcs, circles, splines, filled polygons, and unfilled polygons.
- Load text fonts and write text strings.
- Create and manipulate bitmaps in visible display memory, hidden display memory, main memory, and disk files.
- Perform high-speed bit block transfers (BLTs) between bitmaps or within a single bitmap.
- Monitor "events" such as mouse movement and keyboard input.
- Control the system color map.
- Generate a nondefault cursor pattern and place it anywhere on the screen.
- Refresh a window that has been obscured by other windows.
- Set "clipping" windows.
- Control raster operations.

1.2 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 also provides pad calls (which all begin with the prefix pad_$); these calls allow you to create pads and frames. However, you cannot use pad calls to generate graphics. For a description of pad calls, see *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 programs can issue pad calls. 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 GPR initialization routine gpr_$init.

NOTE: For monochromatic displays, there is a lower level of software called the display driver (SMD) that will become obsolete at SR9.6. For compatibility reasons, we will continue to support SMD calls until at least SR10. At some future release after SR10, SMD calls will no longer be present. Starting with SR9.6, we make no claim as to their performance and will not respond to any UCRs regarding SMD calls.

Chapter 2

Getting Started with GPR Programming

This chapter will get you started writing GPR programs in FORTRAN, Pascal, or C. We explain the essentials for a successful GPR program. This chapter also explains all the display modes, and the various trade–offs in choosing one display mode over another. This chapter demonstrates the calls shown below:

gpr_$init	gpr_$terminate
gpr_$inq_config	gpr_$inq_disp_characteristics

2.1 Writing GPR Application Programs

You can write a GPR program in FORTRAN, Pascal, or C. Regardless of the language, all GPR programs must have the minimum structure shown in Table 2–1.

Table 2–1. Structure of a GPR Program

FORTRAN	Pascal	C
program gpr_example	program gpr_example;	
%include ’/sys/ins/base.ins.ftn’	%include ’/sys/ins/base.ins.pas’;	#include “/sys/ins/base.ins.c”
%include ’/sys/ins/gpr.ins.ftn’	%include ’/sys/ins/gpr.ins.pas’;	#include “/sys/ins/gpr.ins.c”
.	.	.
.	.	.
.	.	.
call gpr_$init(. . .)	gpr_$init(. . .);	gpr_$init(. . .);
.	.	.
.(zero or more GPR	. (zero or more GPR	. (zero or more GPR
. calls or other code.)	. calls or other code.)	. calls or other code.)
call gpr_$terminate(. . .)	gpr_$terminate(. . .);	gpr_$terminate(. . .);

The minimum requirements for a GPR program are to:

- include two insert files.
- call the gpr_$init routine.
- call the gpr_$terminate routine.

In addition to these requirements, we also recommend that you call the gpr_$inq_disp_characteristics routine.

2.1.1 Including Insert Files

In order to write GPR application programs, you must include at least 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 must 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.

Another insert file which should be included in all GPR programs is an error insert file. The error insert file is required if you want to check for system errors. It is recommended that you check for system errors after every system call.

FORTRAN	Pascal	C
/sys/ins/error.ins.ftn	/sys/ins/error.ins.pas	/sys/ins/error.ins.c

2.1.2 Calling the gpr_$init Routine

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 and gpr_$inq_disp_characteristics until GPR is initialized.

When you call gpr_$init, you must specify where the graphics will be displayed (for example, in a window, in a frame, over the entire screen, etc.) through a parameter called the display mode. We detail the pros and cons of the different display modes later in this chapter.

2.1.3 Calling the gpr_$terminate Routine

Use gpr_$terminate to finish your GPR session. After calling gpr_$terminate you cannot call any other GPR routines until you reinitialize the package with gpr_$init. You can initialize and terminate GPR as often as you like within a graphics program; for example, you may initialize GPR in one display mode, perform some task, terminate GPR, then reinitialize GPR in another display mode and perform some other task.

2.1.4 Calling the gpr_$inq_disp_characteristics Routine

As noted earlier, this is not a required call, but we do strongly recommend it for most GPR programs. The gpr_$inq_disp_characteristics routine, like the less-sophisticated gpr_$inq_config routine, returns information about the node and environment that the GPR program is executing on. Therefore, you can use the gpr_$inq_disp_characteristics routine to make the program more portable among the various kinds of Apollo nodes.

For example, suppose you write a program and you want it to run on any Apollo node. Some nodes are 1-plane black and white, some are 4-plane color, some are 8-plane color, and some are 24-plane color. One of the pieces of information returned by gpr_$inq_disp_characteristics is a count of the number of planes on the target node. Therefore, you can write different color map routines for the different nodes and then branch depending on the amount of planes the target node is running on. In other words, you don't have to create a different binary file for each node type.

The information returned by gpr_$inq_disp_characteristics depends not only on the target node, but also on the environment that the program is executing on. For example, when you write a program that is to run in a window (i.e., a direct mode program), it is difficult to predict what size window the user will run the program in. However, the gpr_$inq_disp_characteristics can return this size and your program can react accordingly.

2.2 Sample GPR Programs

This section contains a simple GPR program written in Pascal, FORTRAN, and C. This should help you get started writing GPR programs. These programs simply initialize the graphics package, draw a line, and terminate the graphics package.

Every program example in this manual uses the following three routines:

init

This routine takes one argument (the display mode). This routine makes a call to gpr_$inq_disp_characteristics which returns valuable information. We are particularly interested in determining the size of the target window or target node. After obtaining this information, we call gpr_$init to initialize the graphics package.

check

This routine takes one argument (a string with a description of the place where the error occurred). This routine contains no GPR calls, and is just a general-purpose error-checking routine. It is a good programming practice to call the check routine after every GPR call to see whether or not an error occurred. If an error did occur, the check routine will print the error message.

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. For a detailed description of error checking on the Domain system, see the *Programming with General System Calls* manual.

The GPR insert file contains all the possible runtime error codes. For a description of each error code, see the *DOMAIN Graphics Primitives Resource Calls Reference* manual.

pause

This routine takes one argument (the time to pause). Like check, pause contains no GPR calls and is just a general-purpose pausing routine. This routine simply halts all processing for the specified period of time. We use this routine so that you can examine drawings before the graphics package terminates and the graphics disappear.

2.2.1 Pascal Example

We packaged the init, pause, and check routines into an include file named my_include_file.pas so that we would not have to redefine them for every program. Therefore, the program examples printed in this manual are more to the point. The following routines compose my_include_file.pas:

```
%include '//cascade7/sys/ins/error.ins.pas';
%include '//cascade7/sys/ins/time.ins.pas';

VAR
   status : status_$t;
   display_bitmap  : gpr_$bitmap_desc_t;
   display_bitmap_size : gpr_$offset_t;
   hi_plane        : gpr_$rgb_plane_t;
   display_characteristics : gpr_$disp_char_t;

PROCEDURE CHECK(IN messagex : string);
BEGIN
   if (status.all <> status_$ok then
       begin
           error_$print (status);
           writeln('error occurred while ',messagex);
       end;
END;


Procedure PAUSE(IN t : real);
VAR
   time : time_$clock_t;
BEGIN
   time.high16 := 0;
   time.low32  := trunc(250000 * t);

   time_$wait (time_$relative, time, status);
   check('In Procedure PAUSE.');
END;


Procedure init(IN mode : gpr_$display_mode_t);
VAR
   unit_or_pad        : static stream_$id_t := stream_$stdout;
   disp_len           : static integer16 := sizeof(gpr_$disp_char_t);
   disp_len_returned  : integer16;
   unobscured         : boolean;
BEGIN
   gpr_$inq_disp_characteristics(mode, unit_or_pad, disp_len,
                                 display_characteristics,
                                 disp_len_returned, status);
   check('in init after inquiring');

   display_bitmap_size.x_size := display_characteristics.x_window_size;
```

```
    display_bitmap_size.y_size := display_characteristics.y_window_size;
    hi_plane                   := display_characteristics.n_planes - 1;
    writeln('hi_plane = ', hi_plane);

    gpr_$init(mode, unit_or_pad, display_bitmap_size,
              hi_plane, display_bitmap, status);
    check('in gpr_$init');
 END;
```

Here is a simple Pascal GPR program:

```
 Program getting_started_with_gpr;
 {This program draws a line in borrow mode from the upper left corner of
  the screen to the lower right corner of the screen.
 }
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/gpr.ins.pas';
 %include 'my_include_file.pas';{Contains the init, check, and pause routines.}

 BEGIN
    init(gpr_$borrow);

    gpr_$line(display_bitmap_size.x_size, display_bitmap_size.y_size, status);
    check('drawing line');

 {Pause for 5 seconds, then terminate.}
    pause(5.0);
    gpr_$terminate(false, status);
 END.
```

2.2.2 C Example

We packaged the init, pause, and check routines into an include file named my_include_file.c so that we would not have to redefine them for every program. Therefore, the program examples printed in this manual are more to the point. The following routines compose my_include_file.c:

```
#include "//cascade7/sys/ins/error.ins.c"
#include "//cascade7/sys/ins/time.ins.c"

status_$t           status;
gpr_$bitmap_desc_t  display_bitmap;
gpr_$offset_t       display_bitmap_size;
gpr_$rgb_plane_t    hi_plane;
gpr_$disp_char_t    display_characteristics;

void check(messagex)
char *messagex;
{
   if (status.all)
   {   error_$print (status);
       printf("Error occurred while %s.\n", messagex);
   }
}

void pause(t)
float t;
{
time_$clock_t  time;

   time.high16 = 0;
   time.low32  = 250000 * t;

   time_$wait (time_$relative, time, status);
   check("pausing");
}

void init(mode)
gpr_$display_mode_t   mode;
{
static short int        unit = 1;
static short int        disp_len = sizeof(gpr_$disp_char_t);
       short int        disp_len_returned;
       short int        unobscured;


   gpr_$inq_disp_characteristics(mode, unit, disp_len,
                              display_characteristics,
                              disp_len_returned, status);
   check("in init after inquiring");

   display_bitmap_size.x_size = display_characteristics.x_window_size;
   display_bitmap_size.y_size = display_characteristics.y_window_size;
   hi_plane                   = display_characteristics.n_planes - 1;

   gpr_$init(mode, unit, display_bitmap_size,
             hi_plane, display_bitmap, status);
   check("in init after initializing");
}
```

Here is a simple C GPR program:

```
/* Name of Program -- getting_started_with_gpr */
/* This program draws a line in borrow mode from the upper left corner of
   the screen to the lower right corner of the screen.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, & pause functions.*/

main()
{
   init(gpr_$borrow);

   gpr_$line(display_bitmap_size.x_size, display_bitmap_size.y_size, status);
   check("drawing line");

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

2.2.3 FORTRAN Example

We packaged the init, pause, and check routines into a file named my_ftn_routines.ftn. You must compile this file and then bind the resulting object file with the main program unit. The following routines compose my_ftn_routines.ftn:

```
C**************************************************************
      subroutine CHECK(messagex)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/error.ins.ftn'
%include 'my_common_block.ftn'

      character*(*)  messagex

      if (status .ne. 0) then
         call error_$print (status)
         print 10, messagex
      endif
10    format('error occurred while ', A)
      END
C**************************************************************
      subroutine PAUSE(t)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/time.ins.ftn'
      integer*4     t

      integer*2  time(3)
      time(1) = 0
      time(2) = 4 * t
      time(3) = 0

      call time_$wait (time_$relative, time, status)
      call check('In Procedure PAUSE.')
      END
C**************************************************************
      subroutine INIT(mode)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2  mode
      integer*2  unit, disp_len, disp_len_returned
      logical    unobscured

      unit = 1
      disp_len = 31
      call gpr_$inq_disp_characteristics(mode, unit, disp_len,
     +       display_characteristics, disp_len_returned, status)
      call check('inquiring about the display characteristics')

      display_bitmap_size(1) = display_characteristics(5)
      display_bitmap_size(2) = display_characteristics(6)
      hi_plane               = display_characteristics(15) - 1
      print 20, hi_plane
20    FORMAT ('hi_plane = ', I2)
```

```
        call gpr_$init(mode, unit, display_bitmap_size, hi_plane,
       +               display_bitmap, status)
        call check('calling gpr_$init')
        END
```

Here is a simple sample FORTRAN GPR program:

```
        PROGRAM getting_started_with_gpr
  C This program draws a line in borrow mode from the upper left corner of
  C the screen to the lower right corner of the score.

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

        call init(gpr_$borrow)

        call gpr_$line(display_characteristics(5),
       +               display_characteristics(6),
       +               status)
        call check('drawing line')

  C Pause for 5 seconds, then terminate.
        call pause(5)
        call gpr_$terminate(.false., status)
        end
```

Notice that both my_ftn_routines.ftn and getting_started_with_gpr.ftn access an include file named my_common_block.ftn. This file contains the following declarations:

```
        integer*4  status
        integer*4  display_bitmap
        integer*2  display_bitmap_size(2)
        integer*2  display_characteristics(31)
        integer*2  hi_plane
        common /group1/ status, display_bitmap, display_bitmap_size,
       +                display_characteristics, hi_plane
```

You must compile my_ftn_routines.ftn and getting_started_with_gpr.ftn separately, bind them together, and then execute the resulting object file; for example:

```
  $ ftn my_ftn_routines.ftn
  $ ftn getting_started_with_gpr.ftn
  $ bind my_ftn_routines.bin getting_started_with_gpr.bin -binary simple
  $ simple
```

2.3 Display Modes

Both gpr_$init and gpr_$inq_disp_characteristics require an argument known as a "display mode" (sometimes called an "operation mode"). This section describes each display mode and compares the advantages and disadvantages of each.

The display modes are the following:

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

Most GPR routines can operate within any display mode, but there are some exceptions. For example, you cannot use clipping in frame mode.

NOTE: The display modes can also specify true–color modes or pseudo–color modes. We distinguish between these two modes in Chapter 6.

2.3.1 Borrow Mode

In borrow mode (sometimes called 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 mode is useful for programs that require exclusive use of the entire screen.

A variant of borrow mode, borrow–nc ("no clear") mode, allows you to borrow the full screen without setting all the pixels to zero. Therefore, borrow–nc mode is identical to borrow mode, except that it does not clear the screen. Borrow–nc mode is useful for copying what is on the screen into a file to save for later display or printing.

In Chapter 6, we distinguish between borrow mode and borrow_rgb mode.

Table 2–2 lists the advantages and disadvantages of the borrow modes.

Table 2–2. Advantages and Disadvantages of the Borrow Modes

Advantages	Disadvantages
You can use the entire screen as a display area. Borrow_nc mode allows you to initialize without clearing the bitmap. Borrow mode is probably the easiest mode to program in. (You don't have to acquire and release the display.)	You lose the features offered by the Display Manager while your program is running. For example, you cannot display multiple windows.

2.3.2 Direct Mode

Direct mode is similar to borrow mode, but the program borrows a window from the Display Manager instead of borrowing the entire screen. 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 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. You can only do graphics operations when the display is acquired. When the display is released, the Display Manager resumes control over display functions such as changing the window size and scrolling.

In Chapter 6, we distinguish between direct mode and direct_rgb mode.

Table 2–3 lists the advantages and disadvantages of direct mode.

Table 2-3. Advantages and Disadvantages of Direct Mode

Advantages	Disadvantages
Performance is as good as in borrow mode. You retain the use of the Display Manager. You can use any rectangular part of the screen. Allows use of high-level I/O calls such as READ and WRITE.	You must acquire the display before writing any graphics to the screen, and you must release it whenever you want to return control to the Display Manager. You must redraw (refresh) the window when the screen is redrawn.

2.3.3 Frame Mode

A frame mode program executes within a frame of a Display Manager pad. A graphics program executes more slowly in frame mode than in borrow 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 places some restrictions on the GPR operations that are allowed. The *DOMAIN Graphics Primitives Resource Calls Reference* manual describes the individual routines, including their restrictions.

Table 2-4. Advantages and Disadvantages of Frame Mode

Advantages	Disadvantages
Easy to use; you take care of the graphics calls, and the Display Manager takes care of everything else. Frame mode is appropriate for simple, noninteractive applications. Synchronization with other processes is handled by the Display Manager. 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.	Graphics programs run much slower in frame mode than in the other modes. There are restrictions on the operations you can perform 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 whenever the window is redrawn for any reason, may take a noticeable period of time to complete. You cannot create true-color graphics in frame mode.

NOTE: In general, we recommend that you avoid using frame mode because it is outdated and does not support full GPR functionality.

2.3.4 No-Display Mode

When the program selects gpr_$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 main memory 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.

Table 2-5. Advantages and Disadvantages of No-display Mode

Advantages	Disadvantages
You can perform graphic operations to the bitmap while bypassing the display. You can create bitmaps larger than the screen.	Images are not visible on the screen. You cannot 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

Drawing Figures

This chapter explains how to draw figures with GPR. You can use GPR to draw lines, arcs, splines, circles, and polygons. This chapter demonstrates the calls shown below:

gpr_$arc_c2p
gpr_$arc_3p
gpr_$circle
gpr_$circle_filled
gpr_$close_fill_pgon
gpr_$close_return_pgon
gpr_$close_return_pgon_tri
gpr_$draw_box
gpr_$inq_coordinate_origin
gpr_$inq_cp
gpr_$inq_draw_pattern
gpr_$inq_fill_pattern
gpr_$inq_line_pattern
gpr_$inq_linestyle
gpr_$line
gpr_$move
gpr_$multiline
gpr_$multitrapezoid
gpr_$multitriangle
gpr_$pgon_polyline
gpr_$polyline
gpr_$rectangle
gpr_$set_coordinate_origin
gpr_$set_draw_pattern
gpr_$set_draw_width
gpr_$set_fill_pattern
gpr_$set_line_pattern
gpr_$set_linestyle
gpr_$spline_cubic_p
gpr_$spline_cubic_x
gpr_$spline_cubic_y
gpr_$start_pgon
gpr_$trapezoid
gpr_$triangle

3.1 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_$set_coordinate_origin. If you lose track of the coordinate origin, you can call gpr_$inq_coordinate_origin.

If you initialize a 700 by 700 bitmap, the corners of your bitmap will have the coordinates displayed in Figure 3–1.

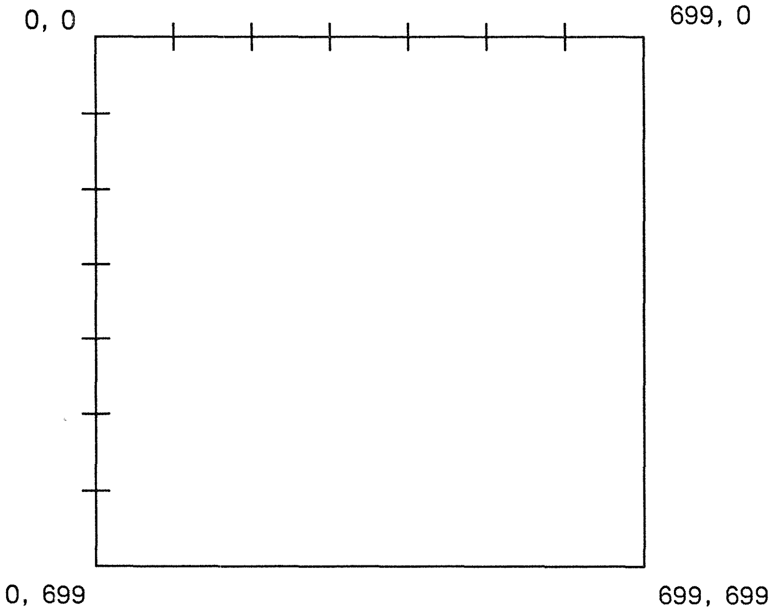


Figure 3–1. How GPR Views Coordinates

3.1.1 Current Position

Many drawing routines rely on the coordinates known as the **current position**; for example, the gpr_$line routine uses the current position as the starting point of a line. After an application program is initialized with gpr_$init, the current position is set to the coordinate origin.

Before calling a routine that depends on the current position, it is often necessary to change the current position. There are two ways to do this. The first is by calling certain GPR routines (e.g., gpr_$line, gpr_$polyline) which automatically generate a new current position upon completion. The second is by calling the gpr_$move routine which changes the current position without drawing a figure.

The gpr_$inq_cp routine returns the x and y coordinates of the current position.

3.2 Lines

It is quite simple to draw one or more lines with the GPR package. All you have to do is to initialize the graphics package (with gpr_$init) and then call one or more of the following routines:

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_$draw_box

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

3.2.1 A Program to Demonstrate gpr_$line and gpr_$move

If you run the following program, GPR will display the image shown in Figure 3-2.

```
Program simple_lines;
{This program demonstrates how to use the gpr_$line call,
 and how to change the current position with the gpr_$move call.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Draw a line from the coordinate origin (0,0) to the endpoint (300,300). }
   gpr_$line(300, 300, status);

{The current position is now set at (300,300).  Use the gpr_$move call to
 change the current position to (100,500). }
   gpr_$move(100, 500, status);

{Draw a line from the current position (100,500) to the endpoint (500,500).}
   gpr_$line(500, 500, status);

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

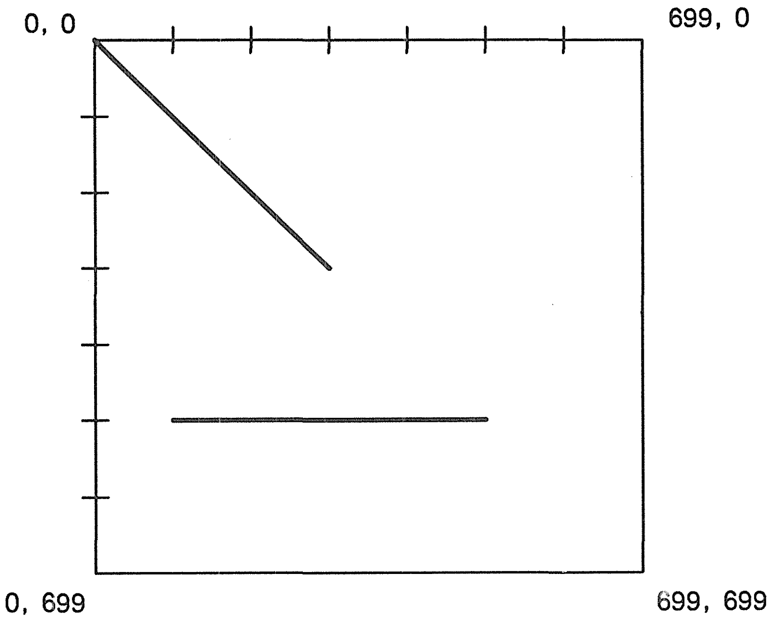


Figure 3-2. Using gpr_$line and gpr_$move to Create Two Lines

3.2.2 A Program to Draw Connected Lines

The next program uses the gpr_$polyline call to draw the three connected lines shown in Figure 3-3. Note that this program could have been written using a gpr_$move and three gpr_$line calls; however, the gpr_$polyline call is usually quicker than a series of gpr_$line calls.

```
Program connected_lines;
{This program draws several connected lines.  It demonstrates the
 gpr_$polyline call.}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
VAR
   array_of_x_coordinates : gpr_$coordinate_array_t := [200, 300, 400];
   array_of_y_coordinates : gpr_$coordinate_array_t := [300, 400, 200];
   number_of_end_points   : integer16 := 3;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
   init(gpr_$borrow);

{Establish the current position at (0,300).  If we do not call gpr_$move, the
 current position will be (0,0).}
   gpr_$move(0, 300, status);

{Draw three connected lines.  Notice that each endpoint becomes the startpoint
 for the next line.}
   gpr_$polyline(array_of_x_coordinates, array_of_y_coordinates,
                 number_of_end_points, status);
{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

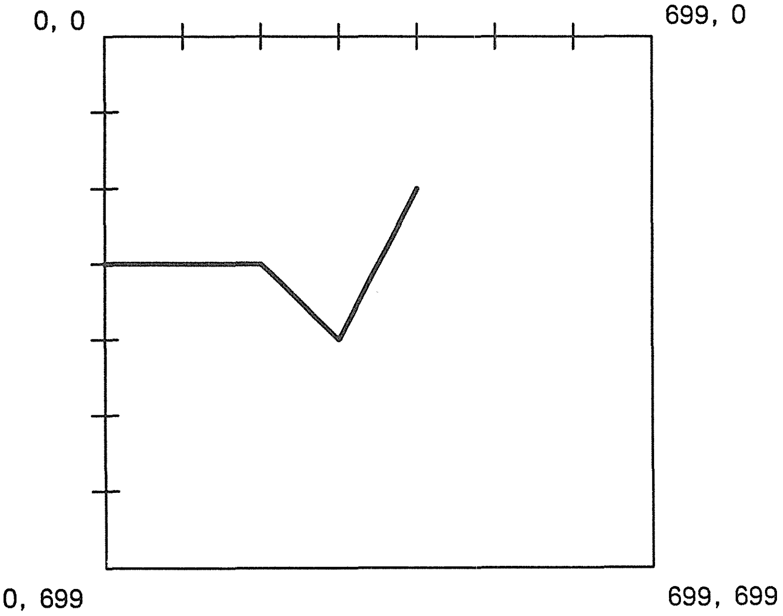


Figure 3-3. Using gpr_$polyline to Create Three Connected Lines

3.2.3 A Program to Draw Disconnected Lines

The next program uses gpr_$multiline to draw the three disconnected lines shown in Figure 3-4.

```
Program disconnected_lines;
{This program draws three disconnected lines.  It demonstrates the
 gpr_$multiline call.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
 array_of_x_coordinates : gpr_$coordinate_array_t :=[100,400,100,400,100,400];
 array_of_y_coordinates : gpr_$coordinate_array_t :=[100,100,200,200,300,300];
 number_of_points       : integer16 := 6;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Draw three disconnected lines.  Line1 runs from (100,100) to (400, 100);
 Line2 runs from (100,200) to (400,200); Line3 runs from (100,300) to
 (400,300).  Notice that we don't have to use gpr_$move to establish the
 current position.}
   gpr_$multiline(array_of_x_coordinates, array_of_y_coordinates,
                  number_of_points, status);

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

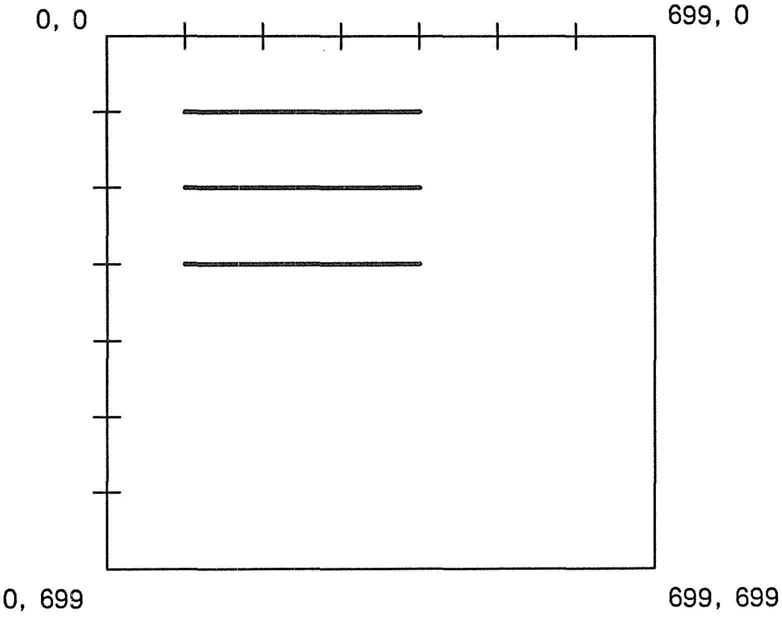


Figure 3-4. Using gpr_$multiline to Draw Three Disconnected Lines

3.3 Circles

To draw a circle, use either of the following GPR routines:

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

gpr_$circle_filled Draws and fills a circle with a specified radius around a specified center point. The current position is not updated.

The following program draws the two circles shown in Figure 3-5.

```
Program circles_example;
{This program draws two circles.  It demonstrates the gpr_$circle and
 gpr_$circle_filled calls.}

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   center : gpr_$position_t;
   radius : 1..32767;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Draw an unfilled circle.}
   center.x_coord := 200;
   center.y_coord := 200;
   radius         := 100;
   gpr_$circle(center, radius, status);

{Draw a filled circle.}
   center.x_coord := 400;
   center.y_coord := 400;
   radius         := 100;
   gpr_$circle_filled(center, radius, status);

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

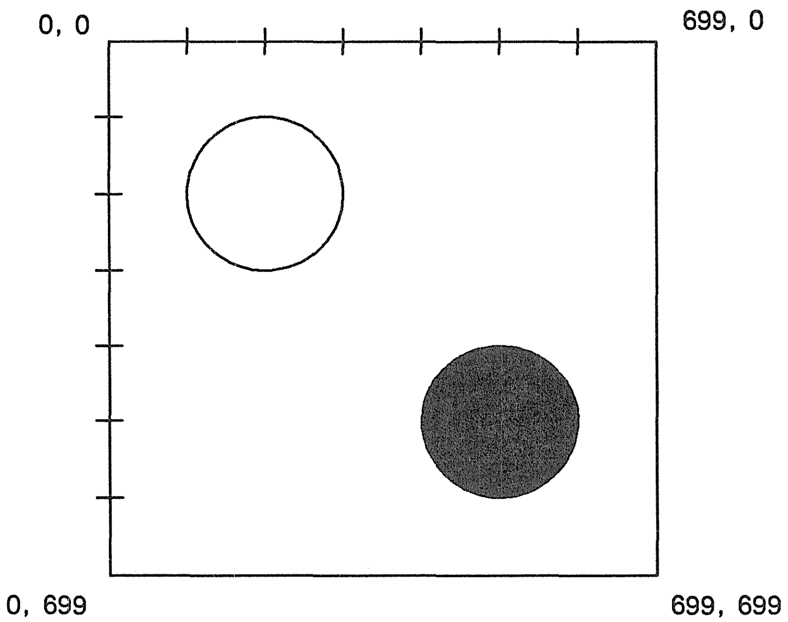


Figure 3–5. Using gpr_$circle and gpr_$circle_filled to Draw Two Circles

3.4 Arcs

An arc is a section of a circle. The section could be as small as one pixel of the circle, or as large as the entire circle. The GPR routines that draw arcs are

gpr_$arc_c2p Draws an arc from the current position to the point where the arc intersects a user-defined line.

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.

The following program draws the two arcs that appear in Figure 3–6.

```
Program arcs_example;
{This program draws two arcs.  It demonstrates the gpr_$arc_c2p and
 gpr_$arc_3p routines.}

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/gpr.ins.pas';

VAR
   center : gpr_$position_t;
   p2     : gpr_$position_t;
   direction : gpr_$arc_direction_t;
   option    : gpr_$arc_option_t;
   point2, point3 : gpr_$position_t;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

                {***Demonstration of gpr_$arc_3p ***}
{The gpr_$arc_3p call draws an arc through any three noncolinear points.
 The three points are the current position, point2, and point3.
 The system draws the arc from the current position through
 point2 and completes the arc at point3.
}
   gpr_$move(200, 200, status);                     {set the current position.}
   point2.x_coord := 300;  point2.y_coord := 300;   {set point2 of the arc.}
   point3.x_coord := 200;  point3.y_coord := 400;   {set point3 of the arc.}
   gpr_$arc_3p(point2, point3, status);             {draw the arc.}

                {***Demonstration of gpr_$arc_c2p ***}
{The gpr_$arc_c2p call draws an arc between two points.
 The radius of the arc is the distance from the current position to center.
 The system starts the arc at the current position and revolves it in a
 clockwise or counterclockwise direction.  The end point of the arc lies
 on an imaginary ray beginning at center and passing through position p2.
 The 'option' parameter is only meaningful if the current position
 is both the start and end point of the arc. In this case, 'option'
 tells the routine whether to draw a full circle or to draw nothing at all.
}
   gpr_$move(600, 400, status);                     {set the current position.}
   center.x_coord := 600; center.y_coord := 600;    {set 'center'}
   p2.x_coord     := 500; p2.y_coord     := 600;    {set 'p2'}
   direction := gpr_$arc_ccw;                   {draw the arc counterclockwise}
   option    := gpr_$arc_draw_full;                 {ignored in this case.}
   gpr_$arc_c2p(center, p2, direction, option, status);
   check('drawing arc_c2p');

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

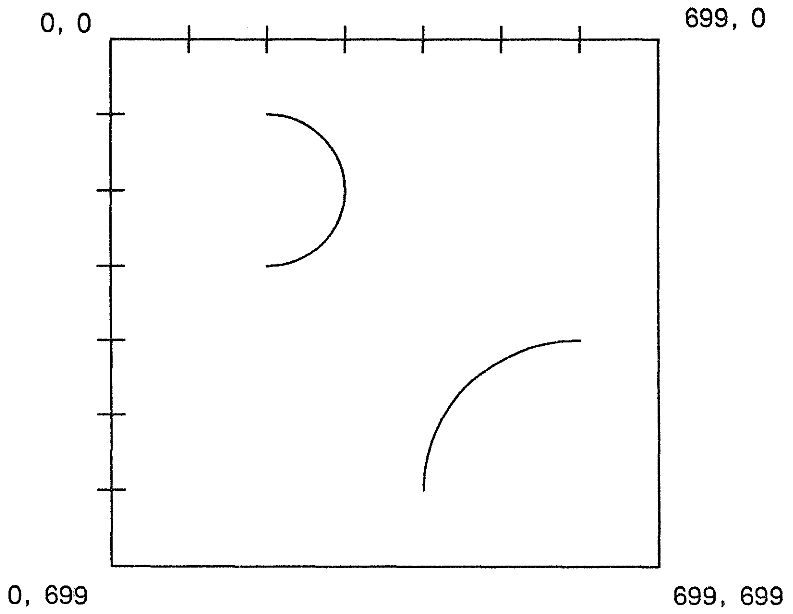


Figure 3-6. Using gpr_$arc_3p and gpr_$arc_c2p to Draw Two Arcs

3.5 Splines

Splines are curves. You pass a list of points to a spline routine, and the routine uses regression formulas to draw curves that best fit the points. Use the following three routines to generate splines:

gpr_$spline_cubic_p Draws a parametric cubic spline from the current position through a list of control points.

gpr_$spline_cubic_x Draws a cubic spline as a function of x from the current position through a list of control points.

gpr_$spline_cubic_y Draws a cubic spline as a function of y from the current position through a list of control points.

After completing any of these calls, the system updates the current position to the coordinates of the last control point.

The following program draws a spline:

```
Program spline_example;
{This program draws a cubic spline as a function of x.
 It demonstrates the gpr_$spline_cubic_x routine.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   array_of_x_pts : array[1..5] of gpr_$coordinate_t := [100,200,300,400,500];
   array_of_y_pts : array[1..5] of gpr_$coordinate_t := [ 20, 80,180,320,500];
   npoints        : integer16 := 5;
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
```

```
BEGIN
   init(gpr_$borrow);

{Draw spline as a function of x.  Starting position of spline will be
 the current position of [0,0] }
   gpr_$spline_cubic_x(array_of_x_pts, array_of_y_pts, npoints, status);
   check('drawing spline');

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

3.6 Filled Polygons

A filled polygon is a closed figure having three or more sides whose interior is partially or totally drawn covered You can use the following routines to generate filled polygons:

gpr_$rectangle Draws and fills a rectangle.

gpr_$triangle Draws and fills a triangle.

gpr_$trapezoid Draws and fills a trapezoid.

gpr_$multitrapezoid Draws and fills one or more trapezoids.

gpr_$multitriangles Draws and fills one or more triangles.

gpr_$start_pgon Defines the starting position of a polygon.

gpr_$pgon_polyline Defines a series of line segments forming part of a polygon boundary.

gpr_$close_fill_pgon Closes and fills the currently open polygon.

gpr_$close_return_pgon Closes the currently open polygon (without filling it) and returns the list of trapezoids within its interior.

gpr_$close_return_pgon_tri Closes the currently open polygon (without filling it) and returns the list of triangles within its interior.

The way that GPR fills a polygon depends on the values of the following attributes:

- Current fill value (described later in this chapter and in Chapter 6).
- Background fill value (described later in this chapter).
- Raster operation (described in Chapter 11).
- Tile pattern (described in Section 3.6.1.).

By default, GPR will fill a filled polygon in solid white on a monochrome display, and in solid red (if you are using the default color map) on a color display.

The following program creates the three filled polygons that appear in Figure 3-7.

```
Program triangle_rectangle_trapezoid;
{This program draws a triangle, rectangle, and trapezoid.  It demonstrates
 the gpr_$triangle, gpr_$rectangle, and gpr_$trapezoid calls.
}
%include ´/sys/ins/base.ins.pas´;
%include ´/sys/ins/gpr.ins.pas´;

VAR
   triangle_vertex1, triangle_vertex2, triangle_vertex3 : gpr_$position_t;
   rectangle : gpr_$window_t;
   trapezoid : gpr_$trap_t;
%include ´my_include_file.pas´;{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Draw a filled triangle.}
   triangle_vertex1.x_coord := 100;  triangle_vertex1.y_coord := 100;
   triangle_vertex2.x_coord := 300;  triangle_vertex2.y_coord := 100;
   triangle_vertex3.x_coord := 200;  triangle_vertex3.y_coord := 200;
   gpr_$triangle(triangle_vertex1, triangle_vertex2, triangle_vertex3,status);

{Draw a filled rectangle.}
   rectangle.window_base.x_coord := 100; rectangle.window_base.y_coord := 300;
   rectangle.window_size.x_size  := 200; rectangle.window_size.y_size  := 300;
   gpr_$rectangle(rectangle, status);

{Draw a filled trapezoid.  In GPR, a trapezoid is a four-sided polygon
 with parallel bottom and top sides.}
   trapezoid.top.x_coord_l := 300;  {x coordinate of the top left point.}
   trapezoid.top.x_coord_r := 500;  {x coordinate of the top right point.}
   trapezoid.top.y_coord   := 200;  {y coordinate of the top line segment.}
   trapezoid.bot.x_coord_l := 400;  {x coordinate of the bottom left point.}
   trapezoid.bot.x_coord_r := 650;  {x coordinate of the bottom right point.}
   trapezoid.bot.y_coord   := 500;  {y coordinate of the bottom line segment.}
   gpr_$trapezoid(trapezoid, status);

{Pause for 5 seconds, then terminate.}
   pause(5);
   gpr_$terminate(false, status);
END.
```

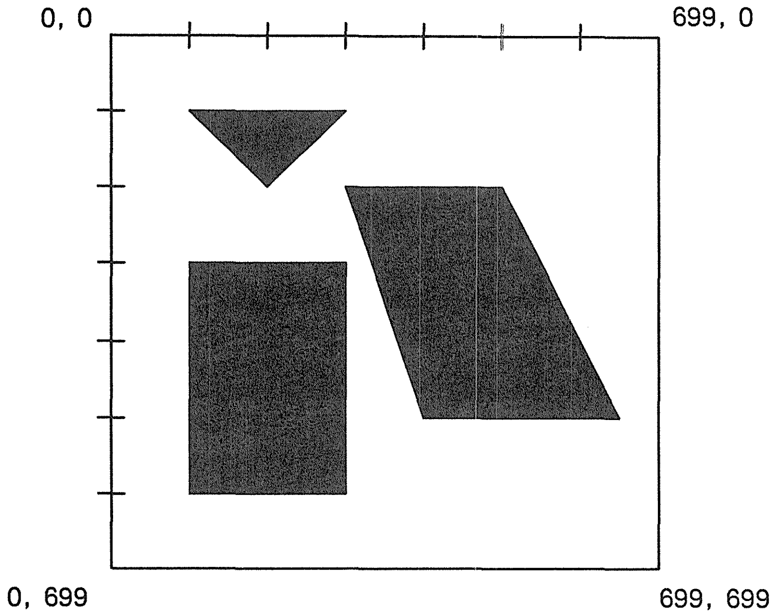


Figure 3-7. Using gpr_$triangle, gpr_$rectangle, and gpr_$trapezoid

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_$close_fill_pgon, gpr_$close_return_pgon, and gpr_$close_return_pgon_tri close a polygon by decomposing it. See Appendix E for a discussion of decomposition techniques.

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 following program draws and fills a five-sided polygon with vertices at points (100,300), (100,600), (600,600), (600,300), and (350,100). (The polygon is shown in Figure 3-8.) The routine gpr_$start_pgon sets the starting position of the polygon at (100,300). The routine gpr_$pgon_polyline defines four line segments. The endpoints of the first line are (100,300), (100,600); the endpoints of the second line are (100,600), (600,600); the endpoints of the third line are (600,600), (600,300); and the endpoints of the fourth line are (600,300), (350,100). The routine gpr_$close_fill_pgon closes the polygon by defining a line from the point (350,100) to the point (100,300) and then fills the polygon.

```
Program polygons;
{The program draws a five-sided polygon.  It demonstrates the gpr_$start_pgon,
 gpr_$pgon_polyline, and gpr_$close_fill_pgon routines.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   array_of_x_coords : gpr_$coordinate_array_t := [100,600,600,350];
   array_of_y_coords : gpr_$coordinate_array_t := [600,600,300,100];
   number_of_points_in_array : integer16 := 4;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
   init(gpr_$borrow);

{Set the starting point of the polygon.}
   gpr_$start_pgon(100, 300, status);

{Set the other four points of the polygon.}
   gpr_$pgon_polyline(array_of_x_coords, array_of_y_coords,
                      number_of_points_in_array, status);
   check('calling pgon_polyline');

{Connect the five points and fill it with the current fill color.}
   gpr_$close_fill_pgon(status);
   check('calling close_fill_pgon');

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

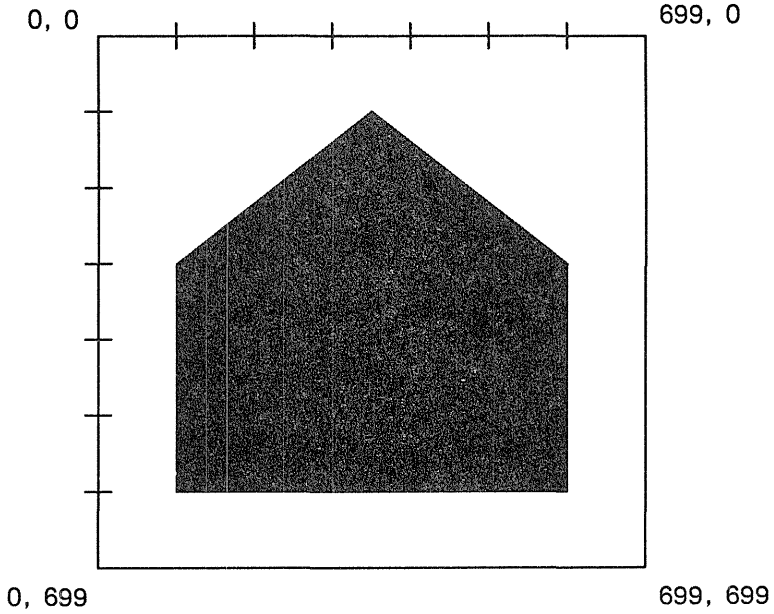


Figure 3-8. Using gpr_$start_pgon, gpr_$pgon_polyline, and gpr_$close_fill_pgon to Create Filled Polygons.

3.6.1 Creating a Tile Pattern

GPR fills circles, rectangles, triangles, trapezoids, and triangles with the current **tile pattern**. By default, the tile pattern is solid, meaning that every pixel within the borders of the image is filled with the same color. However, you can create a nondefault tile pattern. In the following sections, we explain the method for creating a tile pattern on a monochromatic node and the two methods for creating a tile pattern on color nodes.

Tile Patterns on Monochromatic Nodes

To create a nondefault tile pattern on a monochromatic node, follow these steps:

1. Allocate a 32x32 bitmap to store the tile pattern. (Chapter 5 explains how to allocate bitmaps.)
2. Make this bitmap current and generate a pattern of 1's and 0's in it. You might, for example, use a combination of gpr_$move and gpr_$line commands to create a dotted or lined tile pattern.
3. Make the display bitmap current.
4. Call the gpr_$set_fill_pattern routine to specify that your 32x32 bitmap contains the tile pattern.

After establishing a tile pattern, you merely have to call one of the fill routines (e.g., gpr_$circle_filled), and the system will fill the figure using the nondefault tile pattern.

Tile Patterns on Color Nodes -- Method 1

In method 1 for creating a nondefault tile pattern on a color node, you follow the same four steps described for creating a tile pattern on a monochromatic nodes. Note that you must specify a 32x32x1 bitmap; that is, the bitmap you allocate should have only 1 plane (hi_plane = 0). In addition to steps 1 through 4, you will probably also want to follow two additional steps:

5. Call gpr_$set_fill_value. At runtime, every bit with a value of "1" will actually be painted with the given fill value color.
6. Call gpr_$set_fill_background_value. At runtime, every bit with a value of "0" will actually be painted with the given fill background value color.

The following program demonstrates how to create a nondefault tile pattern for a color node:

```
Program tile_pattern;
{This program creates a nondefault tile pattern.  It demonstrates the
 gpr_$set_fill_value, gpr_$set_fill_background_value, and
 gpr_$set_fill_pattern routines.  You must run this program on a color node.}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
CONST
   fill_value = 3;
   background_value = 5;
VAR
   center : gpr_$position_t := [300, 300];
   radius : integer16 := 300;
   x, y   : integer16;
   scale  : integer16 := 1;
   size_of_tile_pattern          : gpr_$offset_t := [32,32];
   attribute_block_descriptor    : gpr_$attribute_desc_t;
   bitmap_desc_of_tile_pattern   : gpr_$bitmap_desc_t;
   hi_plane_of_mmb               : gpr_$rgb_plane_t := 0;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
   init(gpr_$borrow);

{Generate a 32x32 main memory bitmap to hold the tile pattern.  It is
 essential that the hi_plane value be 0.  If it is not 0, then the calls
 to set the fill value and fill background value will have no effect.}
   gpr_$allocate_attribute_block(attribute_block_descriptor, status);
   gpr_$allocate_bitmap(size_of_tile_pattern, hi_plane_of_mmb,
                        attribute_block_descriptor,
                        bitmap_desc_of_tile_pattern, status);
   gpr_$set_bitmap(bitmap_desc_of_tile_pattern, status);

{Set every 16th bit in the bitmap.}
   for x := 0 to 31 do begin
       for y := 0 to 31 do begin
         if ((x MOD 4 = 0) AND (y MOD 4 = 0)) then begin
            gpr_$move(x, y, status);
            gpr_$line(x, y, status);
         end;
      end;
   end;

{Make the display bitmap current and then set 3 attributes in its attribute
 block.  Every 16th bit will be painted with the fill value; the other 15 bits
 will be painted with the background fill value.}
   gpr_$set_bitmap(display_bitmap, status);
   gpr_$set_fill_value(fill_value, status);
   gpr_$set_fill_background_value(background_value, status);
   gpr_$set_fill_pattern(bitmap_desc_of_tile_pattern, scale, status);

{Draw a filled circle using the current tile pattern.}
   gpr_$circle_filled(center, radius, status);

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

Tile Patterns on Color Nodes -- Method 2

In method 2 for creating a nondefault tile pattern on a color node, you must follow these steps:

1. Allocate a 32x32xn bitmap to store the tile pattern where n is the number of planes that you want to use. (Chapter 5 explains how to allocate bitmaps.)

2. Make this bitmap current and then draw the pattern into it. You might, for example, use a combination of gpr_$move and gpr_$line routines to create a dotted or lined tile pattern. You control the colors of the tile pattern with the gpr_$set_draw_value routines.

3. Make the display bitmap current.

4. Call the gpr_$set_fill_pattern routine to specify that your 32x32xn bitmap contains the tile pattern.

NOTE: If you use method 2, the gpr_$set_fill_value and gpr_$set_fill_background_value routines will have no effect on the tile pattern.

3.7 Wide Lines, Arcs, Circles, and Splines

GPR provides the following calls to adjust the width of a line, arc, circle, or spline:

gpr_$set_draw_width Specifies the line width in pixels for all the "draw" calls listed in Table 3-1 at the end of this chapter.

gpr_$inq_draw_width Returns the draw width.

By default, all lines, curves, arcs, and splines are one pixel wide. You can, however, specify thicker lines with the gpr_$set_draw_width routine. By using wide lines, you can highlight selected areas of a figure.

When you specify a nondefault width, GPR tries to balance the drawing so that half the pixels fall on either side of the direct route between the specified endpoints. For example, if you specify a vertical line with a draw width of three pixels, GPR will draw one line between the endpoints, one line to the right of this line, and one line to the left of this line. Obviously, it is easier for GPR to balance lines of odd pixel widths than even pixel widths. If you specify an even pixel width for a line, the system will draw the extra pixel above and/or to the left of the line. If you specify an even pixel width for an arc or a circle, the system will draw the extra pixel on the inside (i.e., towards the center) of the arc or circle.

See Section 3.8 for an example demonstrating gpr_$set_draw_width.

3.8 Solid and Dashed Lines

GPR provides the following calls for specifying the draw or line pattern:

gpr_$set_draw_pattern Specifies the pattern (e.g., solid line, dashes) the system will use for all "draw" calls. The draw calls are listed in Table 3-1 at the end of this chapter.

gpr_$inq_draw_pattern Returns a description of the draw pattern.

gpr_$set_line_pattern Specifies the pattern the system will use for all "line" calls. The line calls are listed in Table 3-2 at the end of this chapter.

gpr_$inq_line_pattern Returns a description of the line pattern.

gpr_$set_linestyle	Specifies either a solid line or a dashed line to be used for all "line" calls. The line calls are listed in Table 3–2 at the end of this chapter.
gpr_$inq_linestyle	Returns a description of the current linestyle.

The gpr_$set_draw_pattern and gpr_$set_line_pattern calls take identical parameters. The only difference between the calls is that gpr_$set_draw_pattern affects more calls than gpr_$set_line_pattern. gpr_$set_line_pattern affects only a subset of the calls that gpr_$set_draw_pattern affects.

The gpr_$set_linestyle routine is a simpler version of the gpr_$set_line_pattern routine. Using the gpr_$set_line_pattern call, you can specify an almost infinite variety of line patterns; using the gpr_$set_linestyle call, you can specify only two (solid or dashed).

Line style, line pattern, and draw pattern are all attributes in attribute blocks. (See Chapter 5 for a definition of attributes and attribute blocks.) However, these are special attributes because the system views them hierarchically. Draw pattern takes precedence over the other two attributes. That is, if you set a draw pattern, then the system will ignore any existing line pattern or line style. Line pattern and line style have equal precedence. If you set both a line pattern and a line style, GPR will only honor the most recently set attribute. For example, if you call gpr_$set_line_pattern first and gpr_$set_linestyle second, then GPR will ignore the value set by gpr_$set_line_pattern.

> NOTE: If you call gpr_$set_linestyle or gpr_$set_line_pattern *after* you call gpr_$set_draw_pattern, the system will return the error message gpr_$style_call_not_active.

To help illustrate the pattern and width routines, we provide the following example which generates the images shown in Figure 3–9.

```
Program widths_and_patterns;
{This program draws one thick line and two dashed lines.  It demonstrates how
 to use gpr_$set_draw_width, gpr_$set_line_pattern, and gpr_$set_draw_pattern.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   width_of_line_in_pixels : integer16 := 11;
   repeat_count : integer16;
   pattern      : gpr_$line_pattern_t;
   length       : 0..64;
   style        : gpr_$linestyle_t;
   scale        : integer16;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
   init(gpr_$borrow);

{The following sequence draws a line 11-pixels thick spanning x from 0 to 300
 and y from 245 to 255 inclusive.
}
   gpr_$set_draw_width(width_of_line_in_pixels, status);
   gpr_$move(  0, 250, status);
   gpr_$line(300, 250, status);

{The following sequence creates a dashed line.  The dashed line pattern
 repeats itself every 150 pixels (which is equal to the repeat count times
 the length).  The pattern consists of 100 pixels set to the draw value,
 followed by 50 pixels set to the background value.  We specified this pattern
 by setting the length equal to 3 which instructed gpr_$set_line_pattern to
 evaluate only the three most significant bits in pattern.  The three most
 significant bits are set to ′110′.  When each bit is magnified by the repeat
 count, the pattern becomes fifty 1′s, fifty 1′s, and fifty 0′s.
}
   repeat_count := 50;
   pattern[1] := 2#1100000000000000;
   length := 3;
   gpr_$set_line_pattern(repeat_count, pattern, length, status);
   gpr_$set_draw_width(1, status);
   gpr_$move(  0, 450, status);
   gpr_$line(500, 450, status);

{The gpr_$set_linestyle call is an alternative to the gpr_$set_line_pattern
 call.  It is simpler to use, but is consequently less powerful.  The
 following sequence creates a pattern that repeats itself every 100 pixels,
 consisting of 50 drawn pixels followed by 50 background pixels.
}
   style := gpr_$dotted;
   scale := 50;
   gpr_$set_linestyle(style, scale, status);
   gpr_$move(0, 650, status);
   gpr_$line(650, 650, status);

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

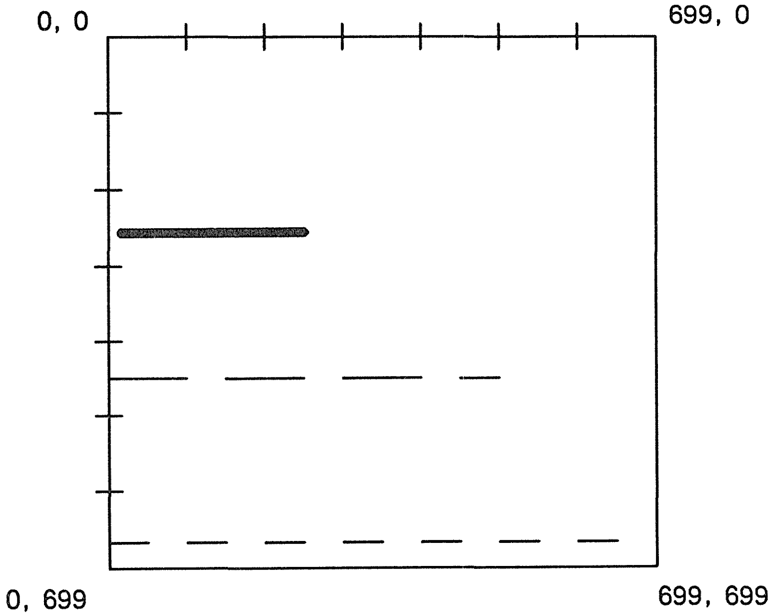


Figure 3-9. Using gpr_$draw_width, gpr_$set_line_pattern, and gpr_$set_linestyle

3.9 Summary

You can build complex figures by calling the simple line, curve, arc, spline, circle, and polygon routines presented in this chapter.

In order to simplify the presentation, we categorized the material geometrically. However, GPR has its own categories, and these categories are important for certain concepts such as raster operations. Therefore, we conclude this chapter with Tables 3–1, 3–2, and 3–3 which break the calls into the categories that GPR relies on.

Table 3–1. Draw Calls

gpr_$arc_c2p
gpr_$arc_3p
gpr_$circle
gpr_$draw_box
gpr_$line
gpr_$multiline
gpr_$polyline
gpr_$spline_cubic_p
gpr_$spline_cubic_x
gpr_$spline_cubic_y

Table 3–2. Line Calls

gpr_$draw_box
gpr_$line
gpr_$multiline
gpr_$polyline
gpr_$spline_cubic_p
gpr_$spline_cubic_x
gpr_$spline_cubic_y

Table 3–3. Fill Calls

gpr_$circle_filled
gpr_$close_fill_pgon
gpr_$multitrapezoid
gpr_$multitriangle
gpr_$rectangle
gpr_$trapezoid
gpr_$triangle

Chapter 4

Printing Text

GPR supports many calls for printing text. You can print text in any bitmap. You can also mix text with graphics images in any bitmap. The *DOMAIN System Utilities* manual contains background information about text in the DOMAIN system. This chapter demonstrates the calls shown below:

- gpr_$inq_character_width
- gpr_$inq_draw_width
- gpr_$inq_horizontal_spacing
- gpr_$inq_space_size
- gpr_$inq_text
- gpr_$inq_text_extent
- gpr_$inq_text_offset
- gpr_$inq_text_path
- gpr_$load_font_file
- gpr_$replicate_font
- gpr_$set_character_width
- gpr_$set_horizontal_spacing
- gpr_$set_space_size
- gpr_$set_text_font
- gpr_$set_text_path
- gpr_$text
- gpr_$unload_font_file

4.1 Fundamental Text Operations

GPR supports the following fundamental routines for writing text:

gpr_$load_font_file Prepares a font for possible use in printing text.

gpr_$set_text_font Selects the current font from among all the prepared fonts.

gpr_$text Writes a string with the current font. The system writes the string into the current bitmap, beginning at the current position and proceeding in the current direction.

gpr_$set_text_path Specifies the direction (i.e., right, left, up, down) that subsequent strings should be written.

gpr_$unload_font_file Unloads a font from the font storage area.

gpr_$inq_text Returns the descriptor of the currently set text font.

The simplest method for printing text is as follows:

1. Call gpr_$load_font_file to prepare a font file for use. Fonts are stored in the /sys/dm/fonts directory.

2. Call gpr_$set_text_font to establish the font as current.

3. Call gpr_$move to specify the coordinates within the current bitmap where the system should begin printing the string.

4. Call gpr_$text to print the string using the current font.

To demonstrate this method, we present the following program:

```
Program simple_text_example;
{This program writes a simple string to the display.  It demonstrates the
 gpr_$load_font_file, gpr_$set_text_font, and gpr_$text routines.
}
%include ´/sys/ins/base.ins.pas´;
%include ´/sys/ins/gpr.ins.pas´;

VAR
   pathname_of_font : name_$pname_t;
   pathname_length  : integer16;
   font_id          : integer16;
   string_to_write  : string := ´Hello from Apollo computer.´;
   string_length    : 1..80  := 27;

%include ´my_include_file.pas´;{Contains the init, check, and pause routines.}
BEGIN
   init(gpr_$borrow);

{Load a font.}
   pathname_of_font := ´f9x15´;  {Pathname /sys/dm/fonts/f9x15 must exist.}
   pathname_length  := 5;
   gpr_$load_font_file(pathname_of_font, pathname_length, font_id, status);
   check(´loading a load_font_file´);
   gpr_$set_text_font(font_id, status);
   check(´setting the text font´);

{Write the string so that it begins at position 100,100}
   gpr_$move(100,100,status);
   gpr_$text(string_to_write, string_length, status);

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

gpr_$text can only print strings. Therefore, in order to print numerical data you must first convert it to a character array.

> NOTE: A GPR program can only use fonts that have been explicitly prepared and set; there is no default font for GPR programs.

4.1.1 Using More Than One Font in the Same Display

A GPR program can print strings in many fonts. Use gpr_$load_font_file to prepare the fonts you intend to use. Then use gpr_$set_text_font to select the current font from among the list of prepared fonts. gpr_$text will use the current font until you call gpr_$set_text_font again; only one font can be current at one time. In general, you do not have to unload a font file (with gpr_$unload_font_file) unless you create a hidden-display memory bitmap. The following program prints a string in three different fonts:

```
Program three_fonts;
{This program writes a string to the display in three different fonts.  It
 demonstrates the gpr_$load_font_file, gpr_$set_text_font, gpr_$text, and
 gpr_$inq_text_extent routines.  The program uses the gpr_$inq_text_extent
 call to measure the string.
}
%include ´/sys/ins/base.ins.pas´;
%include ´/sys/ins/gpr.ins.pas´;

CONST
   output_string = ´The rain in Spain falls mainly on the plain.´;
   string_length = sizeof(output_string);
TYPE
   font_sizes = (large, medium, small);
VAR
   font_id : array[font_sizes] of integer16;
   size_in_pixels  : gpr_$offset_t;
   count : integer16 := 0;
   pick : font_sizes;
%include ´my_include_file.pas´;{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Load three fonts into the font storage area of display memory.}
   gpr_$load_font_file(´f9x15´, 5, font_id[large], status); {load large font}
   gpr_$load_font_file(´f7x13´, 5, font_id[medium], status);{load medium font}
   gpr_$load_font_file(´f5x7´,  4, font_id[small], status); {load small font}

   for pick := large to small do begin
{Establish one of the three fonts as the current font.}
         gpr_$set_text_font(font_id[pick], status);
         count := count + 1;
         gpr_$move(100, count * 100, status);
{Write the string to the display in the current font.}
         gpr_$text(output_string, string_length, status);

{Calculate how much space (in pixels) the string requires.}
         gpr_$inq_text_extent(output_string, string_length, size_in_pixels,
                              status);
```

```
{Write the space information.  Writeln sends information to stdout, not to
 the display.}
         write('The ', pick:0, ' font requires ');
         writeln(size_in_pixels.x_size:0, ' pixels.');
   end;

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

Potential Conflict Between Fonts and Hidden-Display Memory Bitmaps

GPR stores the fonts that it is *using* inside hidden-display memory (HDM). (By *using*, we mean that at least one string has been written with this font.) There is no practical limit to the amount of fonts that GPR can use because the system will swap fonts in to and out of HDM when HDM becomes too full. Therefore, there is no reason to call gpr_$unload_font_file, *unless your program also attempts to create a HDM bitmap.* If HDM is filled up with fonts, then the system will return the error gpr_$no_more_space when you attempt to create a HDM bitmap.

Another conflict between fonts and HDM can cause a runtime error. If your program creates several HDM bitmaps and then begins using fonts, there may not be enough space to store fonts.

We offer two pieces of advice. First, in general, it is not a good idea for a program to create HDM bitmaps and use several fonts. Second, if you choose to mix HDM bitmaps and multiple fonts, you should probably call gpr_$unload_font_file to remove fonts from HDM.

4.1.2 Specifying the Print Direction (Up, Down, Left, Right)

By default, gpr_$text prints text from left to right. You can, however, control the print direction with the gpr_$set_text_path routine. By using this routine, you can print text from left to right, right to left, bottom to top, or top to bottom. The following program demonstrates this capability.

```
Program text_direction;
{This program writes a simple string to the display in each of the four
 possible printing directions.  It demonstrates gpr_$set_text_path.
}
%include ´/sys/ins/base.ins.pas´;
%include ´/sys/ins/gpr.ins.pas´;

TYPE
   words       = array[1..6] of char;
VAR
   font_id   : integer16;
   direction : gpr_$direction_t;
   strings   : array[1..4] of words:=[´ UP   ´, ´ DOWN ´, ´ LEFT ´, ´ RIGHT´];
   count     : integer16 := 0;

%include ´my_include_file.pas´;{Contains the init, check, and pause routines.}

BEGIN
      init(gpr_$borrow);

{Load a font and make it current.}
      gpr_$load_font_file(´f9x15´, 5, font_id, status);
      gpr_$set_text_font(font_id, status);

{The following sequence loops through each of the four possible print
 directions, printing one string in each direction.  Note that the starting
 position for each string is (500,400)
}
      for direction := gpr_$up to gpr_$right do begin
          gpr_$move(500, 400, status);
          gpr_$set_text_path(direction, status);
          count := count + 1;
          gpr_$text(strings[count], 6, status);
      end;

{Pause for 5 seconds, then terminate.}
      pause(5.0);
      gpr_$terminate(false, status);
END.
```

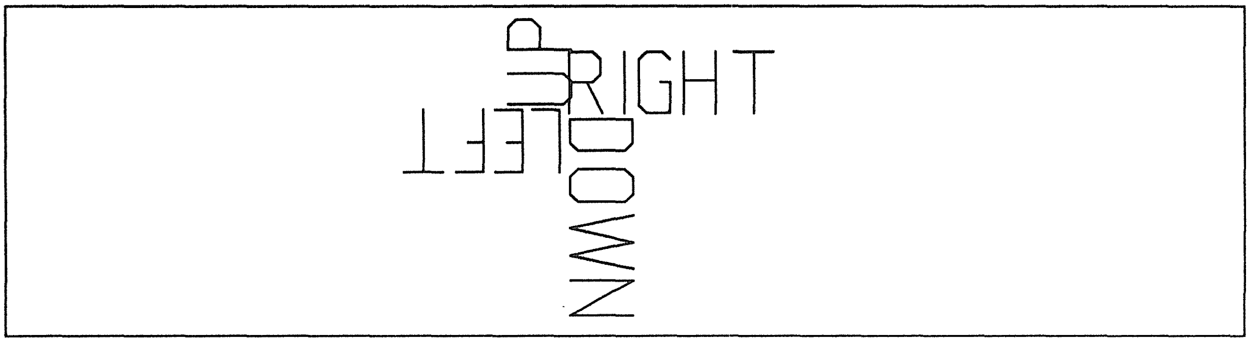


Figure 4-1. Text Can Be Printed in Four Different Directions

4.2 Modifying a Font's Characteristics

GPR supports the following calls to modify a font's characteristics:

gpr_$replicate_font Creates a modifiable copy of 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_$set_space_size Specifies the width of the space to be displayed when a requested character is not in 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.

In order to modify a font's characteristics, you must follow these steps:

1. Call gpr_$load_font_file to specify the font you want to make temporary changes to.
2. Call gpr_$replicate_font. This call makes a copy of an existing font thereby allowing you to make changes to the copy without harming the original.
3. Call gpr_$set_text_font to make this font the current one.
4. Call gpr_$set_character_width, gpr_$set_horizontal_spacing, and/or gpr_$set_space_size to change the font's characteristics.

The following program demonstrates these steps:

```
Program modifiable_fonts;
{This program creates a modifiable copy of a font and then changes the gap
 between characters and the width after every period.  It demonstrates the
 gpr_$replicate_font, gpr_$set_character_width, and
 gpr_$set_horizontal_spacing routines.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   pathname_of_font : name_$pname_t := 'f9x15';
   pathname_length  : integer16     := 5;
   font_id, replicated_font_id : integer16;
   string_to_write  : string := 'Hi there.Good morning.My name is Apollo.';
   string_length    : 1..80  := 43;
   character_to_change : char;
   new_width_in_pixels  : -127..127;
   horizontal_spacing_in_pixels : -127..127;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Prepare a font, make a copy of the font, and then make the copy of the
 font current.}
   gpr_$load_font_file(pathname_of_font, pathname_length, font_id, status);
   gpr_$replicate_font(font_id, replicated_font_id, status);
   gpr_$set_text_font(replicated_font_id, status);

{Tighten the gap between characters by one pixel. This will cut off the right
 edge of wide characters.}
   horizontal_spacing_in_pixels := -1;
   gpr_$set_horizontal_spacing(replicated_font_id,
                                horizontal_spacing_in_pixels, status);

{But, leave a gap of 45 pixels after every period.}
   character_to_change := '.';
   new_width_in_pixels := 45;
   gpr_$set_character_width(replicated_font_id, character_to_change,
                             new_width_in_pixels, status);

{Write the string.}
   gpr_$move(100,100,status);
   gpr_$text(string_to_write, string_length, status);

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

4.3 Finding the Dimensions of a Text String

GPR supports two routines that return the size of a text string.

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.

Figure 4–2 illustrates how the system measures width and height for gpr_$inq_text_extent. Figure 4–3 illustrates how the system measures x and y offsets fŏr gpr_$inq_text_offset.

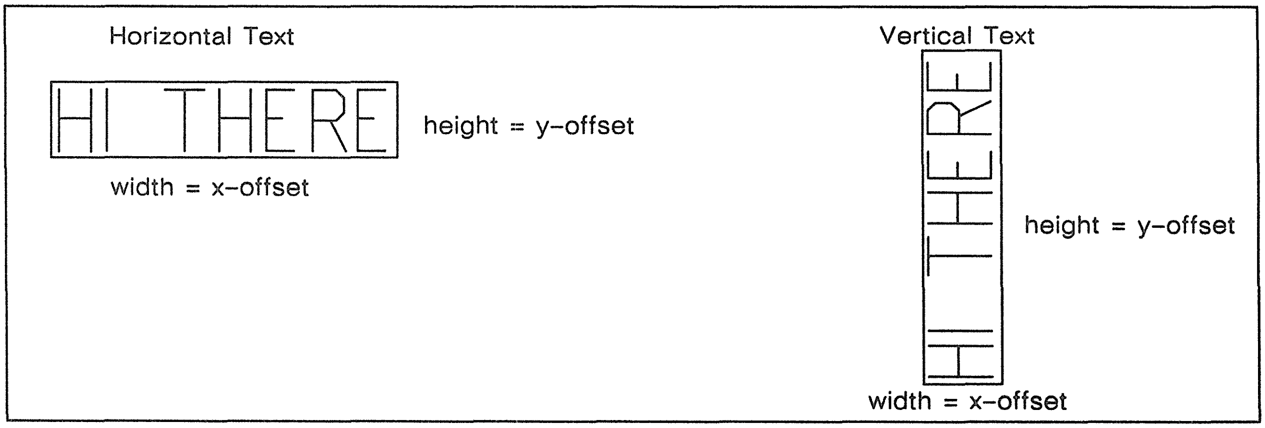
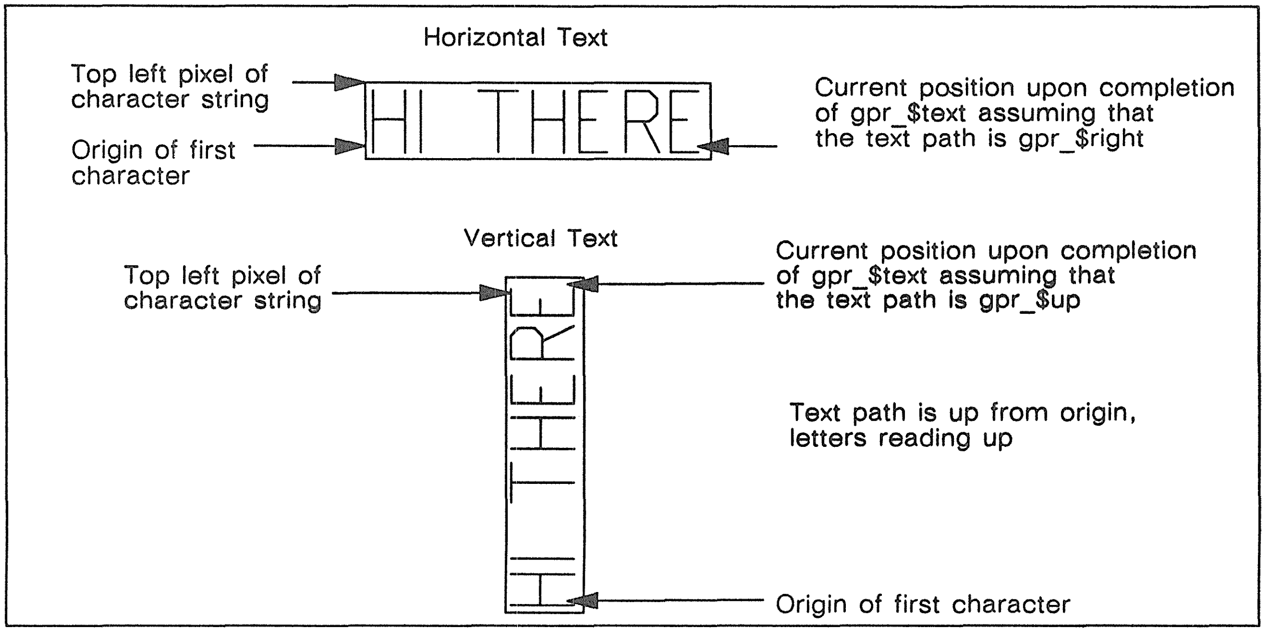


Figure 4–2. Measurements Returned by gpr_$inq_text_extent



Horizontal Text
Top left pixel of character string
Origin of first character
HI THERE
Current position upon completion of gpr_$text assuming that the text path is gpr_$right
Vertical Text
Top left pixel of character string
HI THERE
Current position upon completion of gpr_$text assuming that the text path is gpr_$up
Text path is up from origin, letters reading up
Origin of first character

Figure 4–3. Measurements Returned by gpr_$inq_text_offset

4.4 Setting Text Color

The following routines control the color that GPR will use to print text:

gpr_$set_text_value Specifies the color to use for writing text.

gpr_$set_text_background_value Specifies the color to use for the text background.

gpr_$inq_text_values Returns the text and text background colors.

We detail all color operations in Chapter 6.

Chapter 5

Bitmaps

This chapter explains what bitmaps are, how to create them, how to display them, and how to transfer the information in them to other bitmaps. The chapter describes the GPR routines shown below:

gpr_$additive_blt
gpr_$allocate_attribute_block
gpr_$allocate_bitmap
gpr_$allocate_bitmap_nc
gpr_$allocate_buffer
gpr_$allocate_hdm_bitmap
gpr_$attribute_block
gpr_$bit_blt
gpr_$deallocate_attribute_block
gpr_$deallocate_bitmap
gpr_$deallocate_buffer
gpr_$enable_direct_access
gpr_$inq_bitmap
gpr_$inq_bitmap_dimensions
gpr_$inq_bitmap_pointer
gpr_$inq_bitmap_position
gpr_$inq_bm_bit_offset
gpr_$inq_visible_buffer
gpr_$open_bitmap_file
gpr_$pixel_blt
gpr_$read_pixels
gpr_$remap_color_memory
gpr_$remap_color_memory_1
gpr_$select_color_frame
gpr_$select_display_buffer
gpr_$set_attribute_block
gpr_$set_bitmap
gpr_$set_bitmap_dimensions
gpr_$write_pixels

5.1 Overview of Bitmaps

A bitmap is a two- or three-dimensional array of bit values. The pattern of bit values directly correspond to a pattern that can be displayed on your screen. The Domain system supports the following five kinds of bitmaps:

- Display memory bitmaps
- Hidden display memory bitmaps
- Main memory bitmaps
- External file bitmaps
- Buffer bitmaps

We describe the five kinds of bitmaps in separate sections. Briefly, you can store bitmaps in visible display memory, main memory, hidden display memory, or in an external storage device (usually, a disk drive). However, only bitmaps in display memory are visible on the screen. To see the contents of any other bitmap, you must use a bit-block transfer (BLT) to copy the bitmap to display memory. We detail BLTs in Section 5.9 later in this chapter.

Many GPR routines depend on something called the **current bitmap**. For example, draw routines always draw figures into the current bitmap. We describe the current bitmap in Section 5.8 later in this chapter.

Every bitmap is associated with at least one **attribute block**. An attribute block consists of a group of characteristics (for example, text color for this bitmap, fill color for this bitmap) that operations performed on the bitmap will have. We detail attribute blocks in Section 5.10 later in this chapter.

5.2 Display Memory Bitmaps

A display memory bitmap is a bitmap stored in visible display memory. The current contents of visible display memory translate into the images you currently see on your screen. The **display controller** continuously reads the contents of visible display memory, and translates the bit patterns into the images you see on your screen.

The **refresh frequency** is the number of times per second that the display controller scans every bit in visible display memory. You can find the refresh frequency of the target node by calling gpr_$inq_disp_characteristics. By calling the gpr_$wait_frame routine, you can force the system to wait for the current refresh cycle to end before executing operations that modify the display. (In general, this call is helpful only when changing the color map; see Chapter 6 for information about the color map.)

Any figure you write to a display memory bitmap is almost instantly pictured on your screen. Because the display memory bitmap is so easy to use, many GPR programmers never use any other kind of bitmap. It is perfectly acceptable to use only display memory bitmaps in a program, but we do suggest that you read the rest of this chapter to understand the advantages and disadvantages of using other kinds of bitmaps.

The maximum size of a display memory bitmap is equal to the amount of visible display memory your node supports, which is shown in Table D-1 in Appendix D.

The only way to create a display memory bitmap is to call gpr_$init and specify a borrow, direct, or frame mode. The `getting_started` program in Chapter 2 contains a sample gpr_$init call.

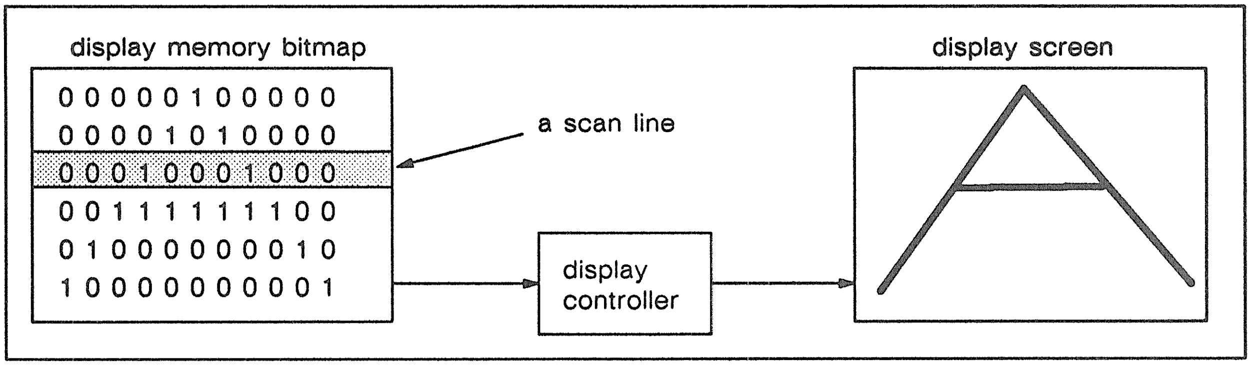
The following tutorial explains how the Domain system translates a display memory bitmap into a displayed image. Experienced graphics programmers will probably want to skip over the tutorial, but new graphics programmers may find the information useful.

5.2.1 How Images Are Displayed: a Tutorial

If you look very carefully at the display screen, you will notice that it is composed of approximately one million tiny dots. These dots are called **pixels** which is short for "picture elements." On a monochromatic display screen, a pixel can only be black or white. On a color display screen, a pixel can be just about any color you desire (in fact, you have 16.7 million colors to choose from).

Every pixel corresponds to one or more bits in the display memory bitmap. For a monochromatic display screen, every pixel is represented by exactly one bit in the display memory bitmap. If that bit has a value of 1, then the pixel will be illuminated (i.e., appear white) on the screen. If that bit has a value of 0, then the pixel will *not* be illuminated (i.e., appear black). For a color display, every pixel is ordinarily represented by 4, 8, or 24 bits in the display memory bitmap (though it is possible to represent a color pixel in a different number of bits such as 2 or 3). We detail color displays in Chapter 6.

The display controller is the interface between the display memory bitmap and the display screen. Its function is to read successive bits 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. Figure 5-1 shows how the display controller translates a display memory bitmap containing 1's and 0's into an image of the letter "A."



```
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 1 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0
0 0 1 1 1 1 1 1 1 0 0
0 1 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0 1
```

Figure 5-1. How a Raster Graphic System Generates an Image

5.3 Hidden Display Memory (HDM) Bitmaps

A hidden display memory bitmap is stored in hidden display memory (HDM). Every Apollo node contains some HDM, though the amount of HDM varies considerably as shown in Table D-1 of Appendix D. The advantage of a HDM bitmap is its location: it is part of display memory, but its 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 bitmaps.

You can access HDM in two ways:

- You can allocate an HDM bitmap by calling the gpr_$allocate_hdm_bitmap routine.
- On certain nodes, you can elongate the initial borrow mode bitmap with gpr_$set_bitmap_dimensions so that the initial bitmap spans both display memory and hidden display memory.

We describe the two methods separately.

5.3.1 Allocating an HDM Bitmap By Calling gpr_$allocate_hdm_bitmap

The gpr_$allocate_hdm_bitmap routine allocates a section of HDM for a user bitmap. You can call this routine from a borrow or direct mode, but you cannot call it from frame mode. The largest HDM bitmap you can allocate is 224 x 219 on DN570s and 224x224 on all other nodes (except the DN600 or DN660 in 8-bit imaging mode which do not support HDM bitmaps). The number of HDM bitmaps you can create is limited by the total amount of HDM that your node supports. If you attempt to allocate an HDM bitmap and there is insufficient free HDM, then the system will return the error gpr_$no_more_space.

When you call gpr_$allocate_hdm_bitmap, GPR does not clear the previous contents for you. Therefore, you will probably want to clear the bitmap yourself by making the HDM bitmap current and then calling the gpr_$clear routine. If you do not clear the HDM bitmap, it will probably contain character sets, icons, tile patterns, and other operating system artifacts. Because the operating system will copy these images to disk before allocating the bitmap for your personal use, you need not worry about destroying important system information when you call gpr_$clear.

To help demonstrate the creation of an HDM bitmap with gpr_$allocate_hdm_bitmap, consider the following program:

```
Program hidden_memory_bitmaps;
{This program creates one hidden display memory bitmap, writes a circle to
 it, and then blts the bitmap to display memory so that the circle becomes
 visible.  It demonstrates the gpr_$allocate_hdm_bitmap, gpr_$allocate_bitmap,
 gpr_$set_bitmap, and gpr_$pixel_blt routines.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   size_of_hdm_bitmap            : gpr_$offset_t;
   attribute_block_descriptor    : gpr_$attribute_desc_t;
   hdm_bitmap_descriptor, bitmap_descriptor  : gpr_$bitmap_desc_t;
   center_of_circle              : gpr_$position_t      := [25,25];
   radius_of_circle              : integer16            := 20;
   source_window                 : gpr_$window_t;
   destination_origin            : gpr_$position_t;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
    init(gpr_$borrow);

{The following sequence creates a 224 by 224 HDM bitmap.  Notice how
 the attribute_block_descriptor returned by gpr_$allocate_attribute_block
 is used as an input parameter in gpr_$allocate_hdm_bitmap.  The value for
 hi_plane comes from 'my_include_file.pas'
}
    gpr_$allocate_attribute_block(attribute_block_descriptor, status);
    size_of_hdm_bitmap.x_size := 224;
    size_of_hdm_bitmap.y_size := 224;
    gpr_$allocate_hdm_bitmap(size_of_hdm_bitmap, hi_plane,
                             attribute_block_descriptor,
                             hdm_bitmap_descriptor, status);
    check('allocating hdm bitmap');

{The current bitmap is now the display bitmap. Therefore, if we call a
 drawing or text routine, all data will be written to the display
 bitmap.  However, we want to draw a filled circle in the HDM bitmap.
 Before drawing the filled circle, we must make the HDM bitmap current
 by calling the gpr_$set_bitmap routine.  Since the gpr_$allocate_hdm_bitmap
 routine does not clear the data stored in hidden memory, we must
 clear it with the gpr_$clear routine.
}
    gpr_$set_bitmap(hdm_bitmap_descriptor, status);
    gpr_$clear(0, status);
    gpr_$circle_filled(center_of_circle, radius_of_circle, status);

{We now blt the entire HDM bitmap to a portion of display memory.
 The gpr_$pixel_blt command blts the specified pixels from the HDM
 bitmap to the current bitmap.  Therefore, before we call gpr_$pixel_blt,
 we must set the current bitmap back to the display bitmap.
}
    source_window.window_base.x_coord :=   0;  {Here, we specify the section}
    source_window.window_base.y_coord :=   0;  {of the HDM bitmap           }
    source_window.window_size.x_size  := 224;  {that we want to blt.        }
    source_window.window_size.y_size  := 224;
    destination_origin.x_coord := 400;  {Here, we specify where in display  }
    destination_origin.y_coord := 400;  {memory the system will blt to.     }
    gpr_$set_bitmap(display_bitmap, status);{Make the display bitmap current.}
    gpr_$pixel_blt(hdm_bitmap_descriptor, source_window,
                    destination_origin, status);

{Pause for 5 seconds, then terminate.}
    pause(5.0);
    gpr_$terminate(false, status);
END.
```

5.3.2 Using HDM By Elongating the Initial Bitmap

If certain conditions are met, you can access HDM by elongating the initial bitmap. To do so, your program must be

- running on a DN550, DN560, DN600, or DN660 node
- running in interactive mode. (See Appendix F for the distinction between interactive and imaging modes.)
- initialized (with gpr_$init) to one of the borrow modes.

When you initialize the graphics package (with gpr_$init), the largest initial bitmap you can create is 1024x1024. You can, however, increase the size of this bitmap to 1024x2048 by calling gpr_$set_bitmap_dimensions. If you do elongate the bitmap, it will have the dimensions shown in Figure 5-2.

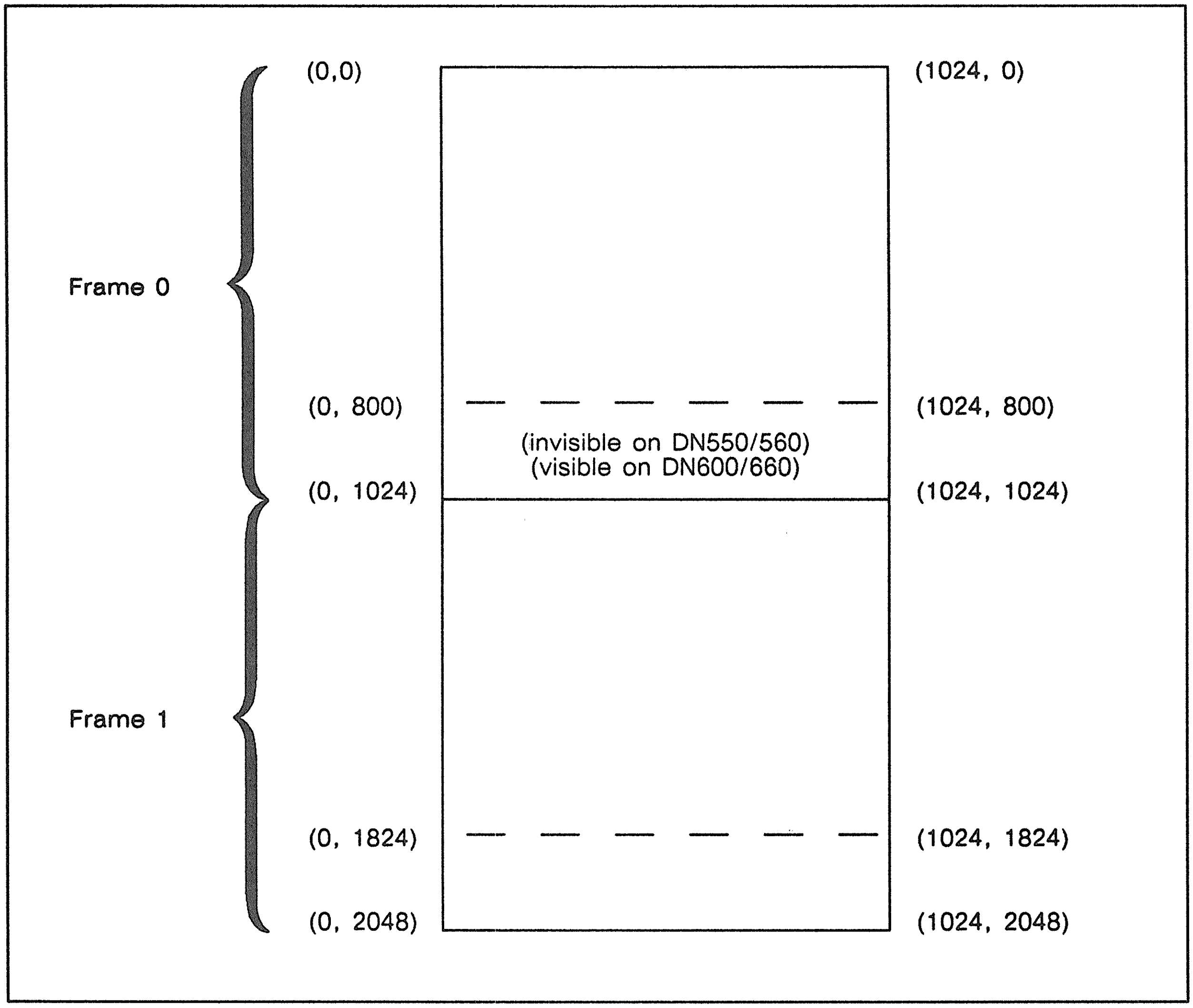


Figure 5-2. Bitmap Dimensions on an Elongated Initial Bitmap

NOTE: Don't confuse the terms "frame 0" and "frame 1" with "frame mode." See Chapter 2 for a description of frame mode.

You can draw figures and write text anywhere within the dimensions 1024x2048, just as you would do for a normal-sized display memory bitmap. However, by default, only the top 1024x800 pixel section is displayed on a DN550/560 and only the top 1024x1024 pixel section is displayed on a DN600/660. If you want to display some other part of the bitmap, you must either

- BLT the bottom part of the bitmap up to the top. When you do the BLT, the display memory bitmap will serve as both the source and destination bitmap.
- Call gpr_$select_color_frame to display either Frame 0 (the top 1024x1024) or Frame 1 (the bottom 1024x1024). Using this method, you can display part of a bitmap while drawing to an invisible portion of the bitmap, and then toggle the visible and invisible portions. This method is similar to, though less sophisticated than, the double-buffering methods described in Section 5.6.

To help demonstrate the use of gpr_$set_bitmap_dimensions and gpr_$select_color_frame we offer the following program:

```
Program toggling_display_frames;
{This program toggles between frame 0 and frame 1.  It demonstrates
 gpr_$set_bitmap_dimensions and gpr_$select_color_frame.  This program will
 only run on a DN550, DN560, DN600, or DN660.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   new_size_of_bitmap  : gpr_$offset_t;
   pictures, frame     : integer16;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
     init(gpr_$borrow);

{The gpr_$init call will not permit you to specify a borrow mode display
 bitmap larger than 1024 x 1024.  Because we want to create a 1024 x 2048
 bitmap, we must call gpr_$set_bitmap_dimensions after we call gpr_$init.
}
     new_size_of_bitmap.x_size := 1024;
     new_size_of_bitmap.y_size := 2048;
     gpr_$set_bitmap_dimensions(display_bitmap, new_size_of_bitmap, hi_plane,
                                status);
     check('resetting dimensions');
{It is a good idea to clear the entire display since HDM probably contains
 fonts,icons, etc.}
     gpr_$clear(0, status);

{We will now draw horizontal lines in frame 0 and vertical lines in frame 1,
 and toggle the two frames at .1 second intervals to examine these
 fast-changing images.
}
     for pictures := 1 to 200 do begin
        frame := pictures MOD 2;
        if frame = 0 then
              begin
                gpr_$move(0,  pictures, status);
                gpr_$line(200,pictures, status);
              end
        else  begin
                gpr_$move(pictures + 300, 1024, status);
                gpr_$line(pictures + 300, 1024 + 200, status);
              end;
        gpr_$select_color_frame(frame, status);
        pause(0.1);
     end;

 {Pause 5 seconds, then terminate.}
      pause(5.0);
      gpr_$terminate(false, status);
END.
```

Note that you can find the dimensions of the current bitmap by calling the gpr_$inq_bitmap_dimensions routine.

5.4 Main Memory Bitmaps

A **main memory bitmap** is a bitmap stored in main memory. You can create a main memory bitmap as large as 8,192 pixels by 8,192 pixels. In general, you can create as many main memory bitmaps as you want; the only limiting factor is the amount of main (physical) memory on your node.

The Domain system can perform an HDM to display memory blt faster than a main memory to display memory blt. However, you can create far more main memory bitmaps than HDM bitmaps. You can use main memory bitmaps to store such things as tiles, fonts, and pop-up menus.

5.4.1 How to Create a Main Memory Bitmap

To create a main memory bitmap, you can call gpr_$allocate_bitmap or gpr_$allocate_bitmap_nc. The only difference between the two commands is that gpr_$allocate_bitmap clears all the bits in the bitmap to zero when it creates the bitmap, but gpr_$allocate_bitmap_nc does not change any bits when it creates the bitmaps. Another way to create a main memory bitmap is to initialize GPR (with gpr_$init) in gpr_$no_display mode.

The following program demonstrates the creation of a main memory bitmap with gpr_$allocate_bitmap.

```
Program main_memory_bitmaps;
{This program creates one main memory bitmap, writes a circle in it, and blts
 the circle to display memory where it can become visible on the screen.
 It demonstrates gpr_$allocate_attribute_block, gpr_$allocate_bitmap,
 gpr_$set_bitmap, and gpr_$pixel_blt.}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
VAR
   size_of_main_memory_bitmap    : gpr_$offset_t;
   attribute_block_descriptor    : gpr_$attribute_desc_t;
   main_memory_bitmap_descriptor : gpr_$bitmap_desc_t;
   center_of_circle              : gpr_$position_t      := [25,25];
   radius_of_circle              : integer16            := 20;
   source_window                 : gpr_$window_t;
   destination_origin            : gpr_$position_t;
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
    init(gpr_$borrow);

{The following sequence creates a 50 by 50 main memory bitmap.  Notice how
 the attribute_block_descriptor returned by gpr_$allocate_attribute_block
 is used as an input parameter in gpr_$allocate_bitmap.  The value for
 hi_plane comes from 'my_include_file.pas'
}
    gpr_$allocate_attribute_block(attribute_block_descriptor, status);
    size_of_main_memory_bitmap.x_size := 50;
    size_of_main_memory_bitmap.y_size := 50;
    gpr_$allocate_bitmap(size_of_main_memory_bitmap, hi_plane,
                         attribute_block_descriptor,
                         main_memory_bitmap_descriptor, status);

{The current bitmap is now the display bitmap. Therefore, if we call a figure
 drawing or text writing routine, all data will be written to the display
 bitmap.  However, we want to draw a filled circle in the main memory bitmap.
 Before drawing the filled circle, we must make the main memory bitmap current
 by calling the gpr_$set_bitmap routine.
}
    gpr_$set_bitmap(main_memory_bitmap_descriptor, status);
    gpr_$circle_filled(center_of_circle, radius_of_circle, status);

{We now blt the entire main memory bitmap to a portion of display memory.
 The gpr_$pixel_blt command blts the specified pixels from the main memory
 bitmap to the current bitmap.  Therefore, before we call gpr_$pixel_blt,
 we must set the current bitmap back to the display bitmap.
}
    source_window.window_base.x_coord :=   0;  {Here, we specify the section}
    source_window.window_base.y_coord :=   0;  {of the main memory bitmap   }
    source_window.window_size.x_size  := 100;  {that we want to blt.        }
    source_window.window_size.y_size  := 100;
    destination_origin.x_coord := 400;  {Here, we specify where in display }
    destination_origin.y_coord := 400;  {memory the system will blt to.    }
    gpr_$set_bitmap(display_bitmap, status);{Make the display bitmap current.}
    gpr_$pixel_blt(main_memory_bitmap_descriptor, source_window,
                   destination_origin, status);
    pause(5.0);  {Pause for 5 seconds, then terminate.}
    gpr_$terminate(false, status);
END.
```

5.5 External File Bitmaps

An external file bitmap is a bitmap stored in a file. The file is usually stored on a disk. You can draw figures or write text to an external file bitmap, but the information won't be visible until you BLT it to display memory.

The big advantage of an external file bitmap is its permanence. By contrast, display memory bitmaps, HDM bitmaps, and main memory bitmaps are all temporary bitmaps. The big disadvantage of an external file bitmap is that it takes a relatively long time for the system to BLT an external file bitmap to display memory.

In general, you should store as an external file bitmap those bitmaps that take a relatively long time to construct. In other words, if you don't want to take the expense of recreating a bitmap, store it as an external file bitmap. For instance, you should probably store digitized photos as external file bitmaps. As a second example, consider the creation of the animated film *Quest: A Long Ray's Journey Into Light* produced by Apollo Computer Inc. Each frame of this movie was stored as a separate external file bitmap.

5.5.1 How to Create an External File Bitmap

The GPR system supports two different storage methods for external file bitmaps. You can create a plane-oriented bitmap or a pixel-oriented bitmap.

You create a **plane-oriented bitmap** by calling the gpr_$open_bitmap_file routine. In this kind of bitmap, you store the value of each pixel in n *sections*, where "n" is the number of planes on your display. Therefore, each section contains the values of each pixel in one plane. For instance, if you are creating an external file bitmap for an 8-plane node, the value of one pixel would be scattered across eight different regions of the bitmap. The advantage of plane-oriented bitmaps is that you can use the GPR draw, fill, and text operations to write images to a plane-oriented bitmap just as you would write images to a main memory bitmap. The disadvantage of plane-oriented bitmaps is that GPR requires a long time to BLT them from disk to display memory. We've printed a sample program that creates a plane-oriented bitmap in Chapter 6.

You also create a **pixel-oriented bitmap** by calling the gpr_$open_bitmap_file routine. In a pixel-oriented bitmap, the value of each pixel is stored in consecutive bits. For example, if you are creating an external file bitmap for an 8-plane node, the value of one pixel is stored in 8 consecutive bits in one section. The advantage of a pixel-oriented bitmap is that the GPR system can BLT a pixel-oriented bitmap from disk to display memory far faster than a plane-oriented bitmap. Moreover, you can BLT a pixel-oriented bitmap by calling the gpr_$pixel_blt routine which does the clipping and offsetting for you. In addition, because the picture is drawn one pixel at a time, you avoid the "rainbow effect" caused by creating the picture one plane at a time. The disadvantage of pixel-oriented bitmaps is that you cannot use the GPR draw, fill, clear, or text operations to write images to a pixel-oriented bitmap. Instead, you must set individual bytes to particular pixel values without the use of GPR routines. However, you can bypass this restriction by writing images to display memory with the GPR routines and then BLTing the images to the pixel-oriented bitmap.

The following program creates a pixel-oriented bitmap for an 8-plane color node. If you are running on a 4-plane node, you can make a few modifications to the program in order to get it running. All you have to do is set the pixel_size to 4 instead of 8, and redefine the line drawing routine so that it only draws 16 lines. For those of you lucky enough to have a 24-plane node, please look at the on-line program named pixel_oriented_bitmaps_24.

```
Program pixel_oriented_bitmaps;
{This program creates a pixel-oriented bitmap for a 4- or 8-plane color node.
 You should run this program twice.  The first time, select the 'create'
 option so that the program will create a file containing a bitmap.  The
 second time, select the 'display' option so that the program will display the

 contents of this file on your monitor.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

TYPE
    memvec         = array[ 0..16#7FFFFFFF ] of CHAR; {# of pixels trying to
read -- e.g., 40000}
    memptr         = ^memvec;
    color_range    = 0..255;
CONST
   x_size_of_bitmap = 200;
   y_size_of_bitmap = 200;
   number_of_bytes_per_pixel = 1;
VAR
   choice              : (create, display);
   pathname_of_bitmap  : name_$pname_t;
   attribs             : gpr_$attribute_desc_t;
   bytes_per_line      : integer16;
   storage             : memptr;
   disk_bitmap         : gpr_$bitmap_desc_t;

%include 'my_include_file.pas';{Contains the init, pause, and check routines.}
{****************************************************************************}
PROCEDURE access_external_bitmap(IN access : gpr_$access_mode_t);
VAR
   pathname_len : 1..name_$pnamlen_max;
   version  : gpr_$version_t;
   size     : gpr_$offset_t;
   groups   : integer16;
   header   : gpr_$bmf_group_header_array_t;
   created  : boolean;
BEGIN
   for pathname_len := 1 to name_$pnamlen_max do
      if pathname_of_bitmap[pathname_len] = ' ' then exit;
   pathname_len := pathname_len - 1;                            {pathname_len}
   version.major := 1;   version.minor := 1;                        {version}
   size.x_size := x_size_of_bitmap; size.y_size := y_size_of_bitmap;   {size}
   groups      := 1;                                                 {groups}
   header[0].n_sects := 1;                                           {header}
   header[0].pixel_size := hi_plane + 1;
   header[0].allocated_size := 8;
   header[0].bytes_per_line := 0;
   header[0].bytes_per_sect := 0;
   header[0].storage_offset := nil;

   gpr_$open_bitmap_file(access,pathname_of_bitmap,pathname_len,version,size,
                         groups, header, attribs, disk_bitmap, created, status);
   check('opening an external file bitmap');
   writeln('allocated_size = ', header[0].allocated_size);
   bytes_per_line := header[0].bytes_per_line;
   storage := header[0].storage_offset;
END;
{****************************************************************************}
Procedure SET_1_PIXEL_IN_EXTERNAL_BITMAP(IN x, y        : integer16;
                                         IN color_index : color_range);
VAR
   offset : integer32;
   i      : integer32;
BEGIN

{Convert a two-dimensional pixel position to a one-dimensional offset from
 the beginning of the bitmap.}
   offset := (y * bytes_per_line) + (x * number_of_bytes_per_pixel);

{Now, load the color index for this pixel into the bitmap.}
   storage^[ offset] := chr(color_index);
END;
{****************************************************************************}
Procedure DRAW_FIGURE; {This simple routine calculates the color value
for every pixel in the bitmap, but it does not actually store any values in
the bitmap.  The SET_1_PIXEL_IN_EXTERNAL BITMAP routine does that.}
VAR
   x,y,q   : integer16;
   color_index : color_range;
BEGIN
   for x := 0 to 200 do begin
       for y := 1 to 200 do begin
           color_index := (y MOD 16);
           SET_1_PIXEL_IN_EXTERNAL_BITMAP (x, y, color_index);
       end;
   end;
END;
{****************************************************************************}
Procedure CREATE_THE_BITMAP;
BEGIN
   gpr_$allocate_attribute_block(attribs,status);
   check('allocating an attribute block.');

   ACCESS_EXTERNAL_BITMAP(gpr_$create);
END;
{****************************************************************************}
Procedure DISPLAY_THE_BITMAP;
VAR
   dest_origin   : static gpr_$position_t    := [400,400];
   source_window : static gpr_$window_t := [[0,0][x_size_of_bitmap,
                                                  y_size_of_bitmap]];
BEGIN
   ACCESS_EXTERNAL_BITMAP(gpr_$readonly);

{BLT the external file bitmap from disk to display memory.}
   gpr_$set_bitmap(display_bitmap, status);
   gpr_$pixel_blt(disk_bitmap, source_window, dest_origin, status);
END;
{****************************************************************************}
BEGIN
   writeln('Do you want to create a bitmap or display a bitmap?');
   write('(Enter ''create'' or ''display'') -- ');     readln(choice);
```

```
    if choice = create
      then writeln('Be patient; it will take some time to create the bitmap.');
    writeln('What is the pathname of the file you want to ', choice:1, ' -- ');
    readln(pathname_of_bitmap);

    INIT(gpr_$borrow);

    if choice = create
       then begin
                 CREATE_THE_BITMAP;
                 DRAW_FIGURE;
            end
       else begin
                 DISPLAY_THE_BITMAP;
                 PAUSE(5.0);
            end;
    gpr_$clear(0, status);
    gpr_$terminate(false, status);
    check('terminating the graphics package');
 END.
```

5.6 Double-Buffer Bitmaps

You can use double buffering to improve the display of changing images.

For instance, suppose you display an image on the screen, and then rotate the image to produce a new image. If the figure is somewhat complex, the display will show several intermediate images as the system rotates the different components of the image one by one. These intermediate images are often undesirable. However, you can use double buffering to prevent the user from seeing the image before it is completed.

A **double-buffer** bitmap is actually two display memory bitmaps. One of these bitmaps (the **primary bitmap**) resides in planes 0 through 7 of display memory. The other bitmap (the **buffer bitmap**) resides in planes 8 through 15 of display memory. At any instant, one of these bitmaps is displayed (visible) and the other bitmap is not displayed (invisible). Therefore, while you display the visible bitmap, you can write to the invisible bitmap. When you have finished writing to the invisible bitmap, you toggle the two bitmaps so that the invisible bitmap becomes visible and vice-versa.

The primary bitmap and the buffer bitmap are always the same size. If you grow the primary bitmap, then the system will automatically grow the buffer bitmap to the same size, and vice-versa.

5.6.1 Availability of Double Buffering

Presently, double buffering is only available when all of the following conditions are met:

- The program is running on a DN590 or DN590-T node.
- The hardware video mode is 8-plane pseudo-color. (This is the default hardware video mode. The Display Manager command CDM controls the hardware video mode.)
- The initialization mode of the program must be a non-RGB mode (for example, gpr_$borrow is okay, but gpr_$borrow_rgb is not).

If any condition is not met, GPR will prevent the program from allocating a double buffer.

5.6.2 GPR Calls Controlling Double Buffering

You use the following GPR calls to control double buffering:

gpr_$allocate_buffer Creates a buffer bitmap.

gpr_$deallocate_buffer Removes the buffer bitmap.

gpr_$inq_visible_buffer Tells you whether it is the primary bitmap or the buffer bitmap that is visible.

gpr_$set_bitmap Specify the bitmap that you want to write to. The default writing buffer is the primary bitmap.

gpr_$select_display_buffer Makes either the primary bitmap or the buffer bitmap visible. In so doing, it makes invisible whichever bitmap was visible.

When you specify gpr_$init in gpr_$direct or gpr_$borrow mode, the system will create a primary bitmap in planes 0 through 7. If you call gpr_$allocate_buffer, the system will create a buffer bitmap in planes 8 through 15. Use the gpr_$select_display_buffer routine to control which of the two bitmaps is visible. Usually, you will want to write to the bitmap which is invisible. Use the gpr_$set_bitmap to control the bitmap that you will write to.

5.6.3 Nuances of Double-Buffer Programming

In general, double-buffer bitmaps are just like display memory bitmaps except for some minor differences regarding:

- Attribute blocks (see section 5.? later in this chapter for details)
- Clip windows (see Chapter 9 for details).
- Refresh procedures (see Chapter 10 for details).

5.6.4 Double Buffering Example

The following example demonstrates double buffering:

```
Program double_buffer_example;
{This program demonstrates double buffering techniques. It uses the
 gpr_$allocate_buffer, gpr_$set_bitmap, gpr_$inq_visible_buffer,
 gpr_$select_display_buffer routines.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

CONST
   images_to_display = 50;
TYPE
   phrase = array[1..14] of char;
VAR
   time_to_pause_between_images   : real;
   buffer_bitmap                  : gpr_$bitmap_desc_t;
   counter                        : integer16;
   what_to_do_to_invisible_bitmap : gpr_$double_buffer_option_t;

%include 'my_include_file.pas';{Contains the init, pause, and check routines.}
{****************************************************************************}
Procedure CREATE_A_BUFFER_BITMAP;
BEGIN
{This procedure allocates a buffer bitmap and then clears its contents.}
   gpr_$allocate_buffer(display_bitmap, buffer_bitmap, status);
   check('allocating a buffer bitmap');
   gpr_$set_bitmap(buffer_bitmap, status);
   gpr_$clear(0, status);
END;
{****************************************************************************}
Procedure WHICH_BITMAP_IS_VISIBLE(OUT visible_bitmap : gpr_$bitmap_desc_t;
                                  OUT invisible_bitmap : gpr_$bitmap_desc_t);
BEGIN
   gpr_$inq_visible_buffer(visible_bitmap, status);
   if visible_bitmap = display_bitmap
       then invisible_bitmap := buffer_bitmap
       else invisible_bitmap := display_bitmap;
END;
{****************************************************************************}
Procedure DRAW_PATTERN_IN_INVISIBLE_BITMAP(IN count : integer16);
{This procedure adds one circle to the bitmap that is currently invisible.}
VAR
    center  : gpr_$position_t;
    radius  : integer16;
    visible_bitmap, invisible_bitmap : gpr_$bitmap_desc_t;
BEGIN
    WHICH_BITMAP_IS_VISIBLE(visible_bitmap, invisible_bitmap);
{Make the invisible bitmap current so that we can draw to it:}
    gpr_$set_bitmap(invisible_bitmap, status);
    gpr_$set_fill_value( (count MOD 8) + 1, status);
{Draw a filled circle in the current bitmap.}
    radius := 80;
    center.x_coord := radius + (count * 12);
    if (odd(count))
       then center.y_coord := 100
       else center.y_coord := 400;
    gpr_$circle_filled(center, radius, status);
END;
```

```
{***************************************************************************}
Procedure MAKE_INVISIBLE_BITMAP_VISIBLE(IN op : gpr_$double_buffer_option_t);
VAR
   display_desc : gpr_$bitmap_desc_t;  {the bitmap to be displayed}
   option_desc  : gpr_$bitmap_desc_t;  {the bitmap the options apply to}
   option_value : linteger;            {a value for option use}
   options      : gpr_$double_buffer_option_t;
BEGIN
   WHICH_BITMAP_IS_VISIBLE(option_desc, display_desc);
   option_value := 0; {ignored unless options = gpr_$clear_buffer}
   options      := op;

{Toggle the bitmaps.}
   gpr_$select_display_buffer(display_desc, option_desc, option_value,
                              options, status);
   check('selecting a display buffer');
END;
{***************************************************************************}
Procedure LOAD_FONT;
VAR
   fontid : integer16;
BEGIN
    gpr_$load_font_file( '/sys/dm/fonts/f7x13', 19, fontid, status);
    check('loading the font file');
    gpr_$set_text_font( fontid, status);
    gpr_$set_text_value(7, status );
END;
{***************************************************************************}
Procedure PRINT_TEXT_IN_INVISIBLE_BITMAP(IN number : integer16);
CONST
  print_text_at_x = 350;
  print_text_at_y = 250;
VAR
   c : array[1..2] of char;
   str : array[1..14] of char;
   visible_bitmap, invisible_bitmap : gpr_$bitmap_desc_t;
BEGIN
   WHICH_BITMAP_IS_VISIBLE(visible_bitmap, invisible_bitmap);
   if invisible_bitmap = display_bitmap
       then str := 'DISPLAY BITMAP'
       else str := 'BUFFER BITMAP ';

   gpr_$move(print_text_at_x, print_text_at_y, status);
   gpr_$text(str, 14, status);

   c[1] := CHR((number DIV 10) + 48);
   c[2] := CHR((number MOD 10) + 48);
   gpr_$move(print_text_at_x, print_text_at_y + 20, status);
   gpr_$text(c, 2, status);
END;
{***************************************************************************}
BEGIN
    write('Enter the pause time (in seconds) between images -- ');
    readln(time_to_pause_between_images);

    writeln('Enter gpr_$clear_buffer, gpr_$undisturbed_buffer, ');
    write('or gpr_$copy_buffer -- ');
    readln(what_to_do_to_invisible_bitmap);

    INIT(gpr_$borrow);      {Create the primary bitmap.}
    CREATE_A_BUFFER_BITMAP; {Create the buffer bitmap.}
    LOAD_FONT;

    for counter := 1 to images_to_display do begin
          DRAW_PATTERN_IN_INVISIBLE_BITMAP(counter);
          PRINT_TEXT_IN_INVISIBLE_BITMAP(counter);
          MAKE_INVISIBLE_BITMAP_VISIBLE(what_to_do_to_invisible_bitmap); {and
                                          {make the visible bitmap invisible.}
          PAUSE(time_to_pause_between_images);
    end;

{Pause for 5 seconds; then terminate.}
    PAUSE(5.0);   {So user can examine the final image.}
    gpr_$terminate(false, status);
END.
```

5.7 Removing Bitmaps

To deallocate any kind of bitmap, call the gpr_$deallocate_bitmap routine.

5.8 The Current Bitmap

Your program can have many bitmaps allocated simultaneously. It is important to know which of these bitmaps is the **current bitmap**. All graphics output operations performed take place on the current bitmap. For example, the gpr_$line routine draws a line in the current bitmap.

When you initialize GPR with gpr_$init, the current bitmap is the initial bitmap. The initial bitmap remains the current bitmap until you designate another as current by calling the gpr_$set_bitmap routine (or call the gpr_$init routine again). Only one bitmap can be current at a time.

You can use the gpr_$inq_bitmap routine to learn which bitmap is the current bitmap.

5.9 Block Transfers (BLTs)

The system uses a bit block transfer (BLT) operation to move bits between two bitmaps or within a single bitmap. GPR supports the following three calls for performing BLT operations:

gpr_$pixel_blt Performs a BLT from *all* n planes of the specified bitmap to n planes of the current bitmap.

gpr_$bit_blt Performs a BLT from a single plane of the specified bitmap to a single plane of the current bitmap.

gpr_$additive_blt Performs a BLT from a single plane of the specified bitmap to *all* active planes of the current bitmap.

In general, the gpr_$pixel_blt will be the easiest call to use. This call copies the contents of an entire bitmap to another bitmap. You can simulate the gpr_$pixel_blt call by calling gpr_$bit_blt one time for each plane. If you are working with single-plane bitmaps, then gpr_$pixel_blt, gpr_$bit_blt, and gpr_$additive_blt will function identically.

The following example contrasts the three BLT calls:

```
Program blts;
{This program contrasts the three BLT calls.  It demonstrates the
 gpr_$bit_blt, gpr_$pixel_blt, and gpr_$additive_blt calls.  This program
 must be run on a color node.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   center              : gpr_$position_t        := [50,50];
   radius              : integer16              := 40;
   fill_value          : gpr_$pixel_value_t     := 6;
   main_mem_bm_size    : gpr_$offset_t          := [100, 100];
   attrib_desc         : gpr_$attribute_desc_t;
   main_mem_bitmap     : gpr_$bitmap_desc_t;
   source_bitmap_desc  : gpr_$bitmap_desc_t;
   source_window       : gpr_$window_t          := [[0, 0  ][100, 100]];
   destination_origin  : gpr_$position_t;
   source_plane        : gpr_$rgb_plane_t;
   destination_plane   : gpr_$rgb_plane_t;
   size                : gpr_$offset_t          := [500,500];

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Create a main memory bitmap.}
   gpr_$allocate_attribute_block(attrib_desc, status);
   gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc,
                        main_mem_bitmap, status);

{Draw a filled circle in the main memory bitmap.}
   gpr_$set_bitmap(main_mem_bitmap, status);
   gpr_$set_fill_value(fill_value, status);
   gpr_$circle_filled(center, radius, status);

   source_bitmap_desc := main_mem_bitmap;
   gpr_$set_bitmap(display_bitmap, status);

{BLT the circle from main memory to display memory three different ways.}
{First, BLT the contents of every plane in the main memory bitmap to every
 plane in the display memory bitmap.
}
   destination_origin.x_coord := 400; destination_origin.y_coord := 0;
   gpr_$pixel_blt(source_bitmap_desc, source_window, destination_origin,
                  status);
   check('pixel blt');

{Second, BLT the contents of plane 1 in the main memory bitmap to plane 3
 of the display memory bitmap.
}
   source_plane := 1;
   destination_plane := 3;
   destination_origin.x_coord := 400; destination_origin.y_coord := 200;
   gpr_$bit_blt(source_bitmap_desc, source_window, source_plane,
                destination_origin, destination_plane, status);
   check('bit blt');

{Third, BLT the contents of plane 0 in the main memory bitmap to every plane
  of the display memory bitmap.  Since plane 0 contains all zeros, the
  circle will probably be displayed in black and may therefore be invisible.
}
   source_plane := 0;
   destination_origin.x_coord := 400; destination_origin.y_coord := 400;
   gpr_$additive_blt(source_bitmap_desc, source_window, source_plane,
                     destination_origin, status);
   check('additive blt');

{Pause 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

5.10 Attributes and Attribute Blocks

Each bitmap is associated with an **attribute block**. An attribute block defines 15 **attributes** that specify the characteristics that operations performed on that bitmap will have. For instance, the attributes control characteristics such as whether lines are drawn with dashed or solid lines (line style attribute), the color of fills (fill color attribute), and the font used for text operations (text font ID attribute).

When you create a display memory bitmap or buffer bitmap, the system assigns an attribute block for you. The attribute block assigned will have all the default attribute values.

When you create a main memory bitmap, hidden display memory bitmap, or external file bitmap, you must specify an attribute block to be associated with the bitmap. Therefore, before creating one of these kinds of bitmaps, you must either call gpr_$allocate_attribute_block to create an attribute block or use the gpr_$attibute_block routine to get the attribute block descriptor associated with the display memory bitmap. The attribute block created by gpr_$allocate_attribute_block will have all the default attribute values.

It is not necessary to maintain a one-to-one correspondence between the number of bitmaps and the number of attribute blocks. For instance, one attribute block can be used by several bitmaps. Conversely, you can create several attribute blocks for only one bitmap; however, only one of these attribute blocks can be the **current attribute block** for the bitmap.

The system uses the attribute values of the current attribute block to set the characteristics of the current bitmap. When you initialize a display memory bitmap, the system sets the current attribute block to a default attribute block that it assigns. The default current attribute block of a main memory bitmap, display memory bitmap, or external file bitmap is the attribute block you specified when you created the bitmap. The only way to change the current attribute block for the current bitmap is by calling the gpr_$set_attribute_block routine. To find the current attribute block for the current bitmap, call the gpr_$attribute_block routine.

To change an attribute value, your program must call one of the attribute-setting routines. These routines change the attributes on the current attribute block only. Therefore, if you want to change an attribute value on an attribute block which is not current, you must first make it current by calling the gpr_$set_attribute_block routine.

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.

Before you change an attribute, you may want to know what value it currently has. GPR provides many **inquiry routines** to find attribute values; all inquiry routines begin with the prefix "gpr_$inq."

The following example shows how to set and employ two attribute blocks for the same bitmap.

```
Program attribute_blocks;
{This program allocates two different attribute blocks and assigns different
 attributes to them.  It demonstrates the gpr_$allocate_attribute_block and
 gpr_$set_attribute_block routines.   If you run this program on a monochrome
 display, you'll get one set of attributes, and if you run it on a color
 display, you'll get a different set of attributes.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
VAR
   attrib_desc1, attrib_desc2  : gpr_$attribute_desc_t;
   main_mem_bitmap    : gpr_$bitmap_desc_t;
   main_mem_bm_size   : gpr_$offset_t          := [400, 400];
   source_window      : gpr_$window_t          := [[0, 0][400, 400]];
   destination_origin : gpr_$position_t        := [100, 100];

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

 {Allocate two attribute blocks.  Both attribute blocks will begin with the
  default attributes.}
   gpr_$allocate_attribute_block(attrib_desc1, status);
   gpr_$allocate_attribute_block(attrib_desc2, status);

 {Allocate a main memory bitmap and make it current.  The third argument could
  have been attrib_desc1 or attrib_desc2; the results will be the same.}
   gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc1,
                        main_mem_bitmap, status);
   gpr_$set_bitmap(main_mem_bitmap, status);

 {Define nondefault attributes for a monochrome node.}
   if hi_plane = 0
      then begin
  {Associate the attributes in attrib_desc1 with the main memory bitmap.}
              gpr_$set_attribute_block(attrib_desc1, status);
  {Assign one nondefault attribute value to this attribute block.}
              gpr_$set_draw_width(20, status);
           end
 {Define nondefault attributes for a color node.}
      else begin
  {Associate the attributes in attrib_desc2 with the main memory bitmap.}
              gpr_$set_attribute_block(attrib_desc2, status);
  {Assign two nondefault attribute values to this attribute block.}
              gpr_$set_draw_width(4, status);
              gpr_$set_draw_value(3, status);
           end;

   {Draw a line in the main memory bitmap from 0,0 to 300,300}
      gpr_$line(300, 300, status);

   {BLT the main memory bitmap to the screen.}
      gpr_$set_bitmap(display_bitmap, status);
      gpr_$pixel_blt(main_mem_bitmap, source_window, destination_origin, status);

   {Pause 5 seconds, then terminate.}
      pause(5.0);
      gpr_$terminate(false, status);
   END.
```

To remove an attribute block allocated by gpr_$allocate_attribute_block, you must call gpr_$deallocate_attribute_block.

5.10.1 Attribute Blocks in Double-Buffer Bitmaps

When you create a buffer bitmap (with gpr_$allocate_buffer) the buffer bitmap will share the attribute block of the primary bitmap. You can, however, associate a different attribute block with the buffer bitmap by calling gpr_$allocate_attribute_block.

5.10.2 A List of Attributes Stored Inside Attribute Blocks

The following list describes all attributes stored inside attribute blocks, their defaults, how to change them, and how to inquire about their current value:

Clipping Window Size

Specifies a rectangular section of the bitmap, outside which no pixels can be modified. Each attribute block defines exactly one clip window.
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.
To change its value use: gpr_$set_clip_window
To find its value use: gpr_$inq_constraints
For more information see: Chapter 9

Clipping Window Enabled

Specifies whether or not the clip window is enabled or disabled.
Default: disabled in borrow and frame modes; enabled in direct modes.
To change its value use: gpr_$set_clipping_active.
To find its value use: not possible to find its value; program must keep track of it.
For more information see: Chapter 9

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.
Default: (0,0).
To change its value use: gpr_$set_coordinate_origin
To finds its value use: gpr_$inq_coordinate_origin
For more information see: Chapter 3

Draw Pattern

Specifies the pattern (i.e., solid or dashed) the system uses for draw routines.
Default: Solid.
To change its value use: gpr_$set_draw_pattern
To find its value use: gpr_$inq_draw_pattern
For more information see: Chapter 3

Draw Value

Specifies the color value the system uses for draw routines.
Default: 1.
To change its value use: gpr_$set_draw_value.
To find its value use: gpr_$inq_draw_value
For more information see: Chapter 6

Draw Width

Specifies the width (in pixels) the system uses for draw routines.
Default: 1 pixel wide.
To change its value use: gpr_$set_draw_width
To find its value use: gpr_$inq_draw_width
For more information see: Chapter 3

Enabled Events

Specifies the graphics event types that have been enabled.
Default: none.
To change its value use: gpr_$enable_input
To find its value use: not possible
For more information see: Chapter 7

Enabled Key Sets

Specifies the keys enabled for a gpr_$keystroke or gpr_$buttons event type.
Default: none.
To change its value use: gpr_$enable_input
To find its value use: not possible
For more information see: Chapter 7

Fill Background Value

Specifies the color value the system uses for drawing the background of tile fills.
Default: -2.
To change its value use: gpr_$set_fill_background_value
To find its value use: gpr_$inq_fill_background_value
For more information see: Chapter 3

Fill Pattern

Specifies the tile pattern the system uses for fill routines.
Default: Solid fill.
To change its value use: gpr_$set_fill_pattern
To find its value use: gpr_$inq_fill_pattern
For more information see: Chapter 3

Fill Value

Specifies the color value the system uses for fill routines.
Default: 1.
To change its value use: gpr_$set_fill_value
To find its value use: gpr_$inq_fill_value
For more information see: Chapter 3 and Chapter 6

Line Style

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.
To change its value use: gpr_$set_linestyle
To find its value use: gpr_$inq_linestyle
For more information see: Chapter 3

Obscured Option

Specifies what the system will do when a direct mode program tries to acquire an obscured window.
Default: gpr_$err_if_obs.
To change its value use: gpr_$set_obscured_opt
To find its value use: not possible.
For more information see: Chapter 10

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.
To change its value use: gpr_$set_plane_mask_32
To find its value use: gpr_$inq_constraints
For more information see: Chapter 9

Raster Operations

When doing a BLT, a fill, or a draw, the raster operation specifies how the bits from the source bitmap will interact with the bits from the destination bitmap.
Default: 3 (sets all destination bit values to source bit values).
To change its value use: gpr_$set_raster_op
To find its value use: gpr_$inq_raster_ops
For more information see: Chapter 11

Raster Operations Primitve Set

Specifies the primitive class(es) (BLTs, fills, or draws) which will be affected by the next gpr_$set_raster_op call, or the primitive class(es) for which gpr_$inq_raster_op will return the current raster operation.
Default: gpr_$rop_line and gpr_$rop_blt
To change its value use: gpr_$raster_op_prim_set
To find its value use: gpr_$inq_raster_op_prim_set
For more information see: Chapter 11

Refresh Procedures

Specifies the entry points of application-supplied procedures that refresh the displayed image in a direct window and hidden display memory.
Default: none.
To change its value use: gpr_$set_refresh_entry
To find its value use: gpr_$inq_refresh_entry
For more information see: Chapter 10

Text Background Value

Specifies the color value the system uses to set the text background.
Default: -2 (same as bitmap background).
To change its value use: gpr_$set_text_background_value
To find its value use: gpr_$inq_text_values
For more information see: Chapter 6

Text Font ID

Specifies the ID of the font the system will use to write text.
Default: No default. Program must load and set font.
To change its value use: gpr_$set_text_font
To find its value use: gpr_$inq_text_font
For more information see: Chapter 4

Text Path

Specifies the direction the system uses to write text.
Default: gpr_$right.
To change its value use: gpr_$set_text_path
To find its value use: gpr_$inq_text_path
For more information see: Chapter 4

Text Value

Specifies the color value the system uses when it writes 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.
To change its value use: gpr_$set_text_value
To find its value use: gpr_$inq_text_values
For more information see: Chapter 6

5.10.3 Attributes Stored with the Bitmap

Some attributes are associated with the bitmap instead of stored inside an attribute block. That is, whenever the bitmap is current, the system uses the values of the following attributes:

Acquire Time-Out Period

Specifies the maximum length of time that a direct mode program can acquire the display without releasing it.
Default: 60 seconds
To change its value use: gpr_$set_acq_time_out
To find its value use: not possible
For more information see: Chapter 10

Auto-Refresh

Specifies whether or not the Display Manager will automatically refresh the bitmap's window.
Default: False (i.e., auto-refresh not on)
To change its value use: gpr_$set_auto_refresh
To find its value use: not possible
For more information see: Chapter 10

Current Position

Specifies the coordinates within the bitmap at which certain writing operations will begin writing.
Default: Coordinate value (0,0) is the starting current position.
To change its value use: gpr_$move (though many other routines implicitly change the current position).
To find its value use: gpr_$inq_cp
For more information see: Chapter 3

5.11 Translating Bitmaps to and from Arrays

Some programmers prefer storing images in arrays rather than bitmaps. GPR supports a call named gpr_$read_pixels which copies a bitmap and converts it into an array. You can manipulate this array as you would manipulate any other array. Eventually, if you want the image is to be displayed, you must convert it back into a bitmap with the gpr_$write_pixels call.

Note that gpr_$read_pixels produces an array of 32-bit integers. That is, on a monochromatic display, one pixel is represented by one bit in a bitmap. However, if you convert the bitmap to an array, gpr_$read_pixel will store the value of that one pixel in a long integer. In other words, the array will take up 32 times more space than the bitmap. For a color display, one pixel will be represented by 4, 8, or 24 bits in a bitmap. When you convert the bitmap to an array, gpr_$read_pixel will also store the total plane value in a long integer. (See Chapter 6 for a description of total plane value.)

In general, it is far simpler to manipulate bitmaps with GPR calls than to convert the bitmaps to arrays and try to keep track of which array element corresponds to which pixel.

The following sample program converts a color bitmap to an array, manipulates the array, and converts the array back to a bitmap where it is displayed:

```
Program read_write_pixels;
{This program demonstrates how to manipulate a bitmap with the
 gpr_$read_pixel and gpr_$write_pixel routines.  This program will work
 only on a color node.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

CONST
   fill_color           = 2;
   new_fill_color       = 3;
   new_background_color = 4;
VAR
   i,j              : integer16;
   center           : gpr_$position_t := [50,50];
   radius           : integer16 := 40;
   source_window       : gpr_$window_t := [ [0,0][100,100] ];
   destination_window : gpr_$window_t := [ [0,200][100,100] ];
   pixel_array : array[1..100,1..100] of integer32;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{Draw a filled circle on the screen.}
   gpr_$set_fill_value(fill_color, status);
   gpr_$circle_filled(center, radius, status);

{Convert a 100x100xhi_plane portion of the display bitmap into an array.
 Note that the call does not change the existing bitmap in anyway.}
   gpr_$read_pixels(source_window, pixel_array, status);
   check('reading pixels');

{Examine each element of the array.  Give each element a new fill color
 and a new background color.  Pascal programmers should be aware that
 the array will appear backwards; that is, the first element is y and
 the second element is x.}
   for i := 1 to 100 do begin
      for j := 1 to 100 do begin
         if pixel_array[i,j] = fill_color
            then pixel_array[i,j] := new_fill_color
            else pixel_array[i,j] := new_background_color;
      end;
   end;

{Now convert the modified array back to a bitmap and put it in the designated
 region of the display bitmap.}
   gpr_$write_pixels(pixel_array, destination_window, status);
   check('writing pixels');

{Pause 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

5.12 Manipulating a Bitmap Through a Bitmap Pointer

The system permits you to modify the bits in a bitmap without using GPR calls. Modifying a bitmap in this way is far more complicated than issuing GPR draw or fill calls, so unless you're an experienced programmer it is probably best to avoid this method.

The following four calls are important if you want to manipulate a bitmap through a bitmap pointer:

gpr_$inq_bitmap_pointer Returns the starting virtual address of a plane in a bitmap.

gpr_$remap_color_memory Use this call in combination with the gpr_$inq_bitmap_pointer call. Use it to remap the specified plane of a display memory bitmap so that the plane's starting virtual address is equal to the address returned by gpr_$inq_bitmap_pointer.

gpr_$remap_color_memory_1 This call is identical to gpr_$remap_color_memory except that you use it only when manipulating the bits in Frame 1 of a DN550, DN560, DN600, or DN660. (See Section 5.3.2 for details about Frame 1.)

gpr_$enable_direct_access You should use this call before modifying the display memory bitmap, but after using the pointer returned by gpr_$inq_bitmap_pointer.

Basically, you can read or modify a particular bit in a bitmap by calculating its offset from the starting virtual address.

5.12.1 Manipulating a Main Memory Bitmap Through a Bitmap Pointer

The planes of a main memory bitmap are stored sequentially, with one plane immediately following the prior one. The following program demonstrates how to directly manipulate a main memory bitmap by locating its starting virtual address:

```
program inq_bm_ptrs_in_main_mem;
{This program demonstrates how to access the virtual addresses of a main
 memory bitmap.  It illustrates the gpr_$inq_bitmap_pointer routine.  This
 program allocates a main memory bitmap and then sets the value of every pixel
 associated with this bitmap to color index 5.  It then BLTs the bitmap to
 the screen.  This program must be run on a color node.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

CONST
  width_of_bitmap  = 480;  {This number should be divisible by 32.}
  height_of_bitmap = 320;
  array_size = width_of_bitmap DIV 32;
TYPE
  bitmap_setup     =  array[1..array_size] of integer32;    { = 480 bits }
VAR
  main_mem_bm_size   : gpr_$offset_t := [width_of_bitmap, height_of_bitmap];
  window             : gpr_$window_t := [[0,0]
                                        [width_of_bitmap, height_of_bitmap]];
  plane              : gpr_$rgb_plane_t;
  main_mem_bitmap    : gpr_$bitmap_desc_t;
  stor_line_width, scan, c : integer16;
  attrib_desc        : gpr_$attribute_desc_t;
```

```
    storage_ptr         :^bitmap_setup;
    dest_origin         : gpr_$position_t := [10,10];

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

begin
   init(gpr_$borrow);

{Create a main memory bitmap.}
   gpr_$allocate_attribute_block(attrib_desc, status);
   gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc,
                        main_mem_bitmap, status);

{Set every pixel in this bitmap to color index 5.  We do this by setting every
 bit in plane 0 and plane 2 to '1', and by setting every bit in the other
 planes to '0'.}
   gpr_$inq_bitmap_pointer(main_mem_bitmap, storage_ptr,
                           stor_line_width, status);
   for plane := 0 to hi_plane do begin
      for scan := 1 to height_of_bitmap do begin
         for c := 1 to array_size do begin  {array_size*32 = width_of_bitmap}
            if ((plane = 0) OR (plane = 2))
              then storage_ptr^[c] := 16#FFFFFFFF  {Set all 32 bits to '1'}
              else storage_ptr^[c] := 16#00000000; {Set all 32 bits to '0'}
         end;
{In Pascal, pointers point to bytes. Therefore, since stor_line_width is a
 2-byte integer, we must multiply it by 2 to get the number of bytes.}
         storage_ptr := UNIV_PTR(INTEGER32(storage_ptr) +(2*stor_line_width));
      end;
   end;

{BLT the main memory bitmap to the screen.}
   gpr_$pixel_blt(main_mem_bitmap, window, dest_origin, status);

{Pause 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(true,status);
end.
```

5.12.2 Manipulating a Display Memory Bitmap Through a Bitmap Pointer

The planes of a display memory bitmap are not necessarily stored sequentially. You must first use the gpr_$inq_bitmap_pointer routine to return a starting virtual address. You must then use the gpr_$remap_color_memory (or gpr_$remap_color_memory_1) routine to remap the planes of a display memory bitmap so that a particular plane will begin at the starting virtual address returned by gpr_$inq_bitmap_pointer. The following program demonstrates how to directly manipulate the bits in a color display:

```
program inq_bm_ptrs_in_disp_mem;
{This program demonstrates how to access the virtual address of a display
 memory bitmap.  It illustrates the gpr_$inq_bitmap_pointer,
 gpr_$enable_direct_access, and gpr_$remap_color_memory routines.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
CONST
  width_of_rectangle  = 480;
  height_of_rectangle = 320;
  array_size          = width_of_rectangle DIV 32;
TYPE
  bitmap_setup     =  array[1..array_size] of integer32;
VAR
  plane               : gpr_$rgb_plane_t;
  storage_line_width, scan_line, c :  integer16;
  storage_ptr_save, storage_ptr    : ^bitmap_setup;
  rectangle           : gpr_$window_t := [ [0,0][width_of_rectangle,
                                               height_of_rectangle]];

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

begin
    init(gpr_$borrow);

{Display a rectangle filled with color index 15.}
    gpr_$set_fill_value(15, status);
    gpr_$rectangle(rectangle, status);
    pause(1.0);

    gpr_$inq_bitmap_pointer(display_bitmap, storage_ptr_save,
                            storage_line_width, status);
    FOR plane := 0 to 3 do begin
{The first time through the loop, the system will remap color memory so that
 the starting virtual address of plane 0 will be equal to storage_ptr_save.
 The next time through the loop, the system will remap color memory so that
 the starting virtual address of plane 1 will be equal to storage_ptr_save,
 and so on with planes 2 and 3.  We must reestablish the value of storage_ptr
 each time through the loop.
}
       gpr_$remap_color_memory(plane, status);
       storage_ptr := storage_ptr_save;

{It is good programming practice to call gpr_$enable_direct_access before
 writing directly to the bitmap (i.e., before changing bits without using
 GPR calls).}
       gpr_$enable_direct_access(status);

       for scan_line := 1 to height_of_rectangle do begin
          for c := 1 to (width_of_rectangle DIV 32) do
{Assign 32 bits.  In hex, the constant is 16#00F0F0F0.}
             storage_ptr^[c] := 2#00000000111100001111000011110000;
          storage_ptr := UNIV_PTR( INTEGER32( storage_ptr ) +
                         (2 * storage_line_width));
       end;
       pause(1.0);
    END;

{Pause 2 seconds, then terminate.}
    pause(2.0);
    gpr_$terminate(true,status);
end.
```

5.12.3 Manipulating a Hidden Display Memory Bitmap Through a Bitmap Pointer

You manipulate a HDM bitmap in much the same way as you manipulate a display memory bitmap. The following program demonstrates the method:

```
PROGRAM inq_bm_ptrs_in_hdm;
{This program demonstrates how to access the virtual addresses of a HDM
 bitmap.  It illustrates the gpr_$inq_bitmap_pointer routine.  This program
 allocates a HDM and then sets the value of every pixel associated with
 this bitmap to color index 5 (which is yellow on the default color map).
 It then BLTs the HDM bitmap to the screen.  This program must be run on
 a color node.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

CONST
  width_of_bitmap       = 128;
  height_of_bitmap      = 64;
  array_size            = width_of_bitmap DIV 32;
TYPE
  hdm_mem_fmt           = array[1..array_size] of integer32; { = 64 bits }
VAR
  source_window         : gpr_$window_t := [[0,0],[128,64]];
  stor_line_width, index, plane, scan : integer16;
  attrib_desc           : gpr_$attribute_desc_t;
  bitm_size             : gpr_$offset_t := [1024,800];
  size_of_hdm_bitmap    : gpr_$offset_t := [width_of_bitmap,height_of_bitmap];
  hdm_bitmap            : gpr_$bitmap_desc_t;
  hdm_ptr_save,hdm_ptr  : ^hdm_mem_fmt;
  dest_origin           : gpr_$position_t := [10,10];
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
  init(gpr_$borrow);

 {Create hidden display memory bitmap.}
  gpr_$allocate_attribute_block(attrib_desc, status);
  gpr_$allocate_hdm_bitmap(size_of_hdm_bitmap, hi_plane, attrib_desc,
                           hdm_bitmap, status);

  gpr_$inq_bitmap_pointer(hdm_bitmap, hdm_ptr_save, stor_line_width, status);

  FOR plane := 0 to hi_plane do begin
    gpr_$remap_color_memory(plane, status);
    hdm_ptr := hdm_ptr_save;
    gpr_$enable_direct_access( status );
    for scan := 1 to height_of_bitmap do begin
      for index := 1 to array_size do begin
 {Set every bit in plane 0 and plane 2 to '1'.  Set every bit in all other
  planes to '0'}
        if (plane = 0) OR (plane = 2)
          then hdm_ptr^[index] := 2#11111111111111111111111111111111
          else hdm_ptr^[index] := 2#00000000000000000000000000000000;
        end;
      hdm_ptr := UNIV_PTR( INTEGER32( hdm_ptr ) + (2 * stor_line_width) );
    end;
  END;
  {BLT the HDM bitmap to the screen.}
    gpr_$pixel_blt(hdm_bitmap, source_window, dest_origin, status);

  {Pause 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(true,status);
  END.
```

Notice that we used gpr_$remap_color_memory routine to remap planes one at a time. However, if we had run this program on a DN550, DN560, DN600, or DN660 we would have to have used gpr_$remap_color_memory_1 instead.

Chapter 6

Color

This chapter explains how to use GPR calls to control the color of the images displayed on your monitor. In this chapter, we demonstrate the following calls:

gpr_$clear

gpr_$color_zoom

gpr_$inq_bitmap_file_color_map

gpr_$inq_color_map

gpr_$inq_draw_value

gpr_$inq_fill_background_value

gpr_$inq_fill_value

gpr_$inq_text_values

gpr_$set_bitmap_file_color_map

gpr_$set_color_map

gpr_$set_draw_value

gpr_$set_fill_background_value

gpr_$set_fill_value

gpr_$set_text_background_value

gpr_$set_text_value

gpr_$wait_frame

6.1 Overview of Color on the Domain System

This section is a tutorial describing the fundamentals of color display on the Domain system from a GPR programmer's perspective. As you will see, the color of a pixel depends on the interaction of the following components:

- The color monitor.
- The color map.
- The contents of the pixel's planes.
- Your GPR program.

Your GPR program writes bit values into a pixel's planes. The node totals the value of the pixel's planes. The node uses this value as an index to a particular entry into the system's color map. The particular entry in the color map describes the intensity levels of the red, green, and blue color guns of the color monitor for that pixel. The combination of the red, green, and blue color guns produces a particular color that you see on the screen.

6.1.1 Overview of the Color Monitor

The color monitor displays images by illuminating each pixel on the screen with a combination of three primary color guns.

Every Apollo color monitor contains three primary color guns: a red gun, a green gun, and a blue gun. Each color gun is calibrated to produce 256 (= 2^8) different color intensity levels from 0 (low) to 255 (high). Since there are three color guns, each monitor is capable of illuminating a pixel with one of 256^3 (~16.7 million) colors.

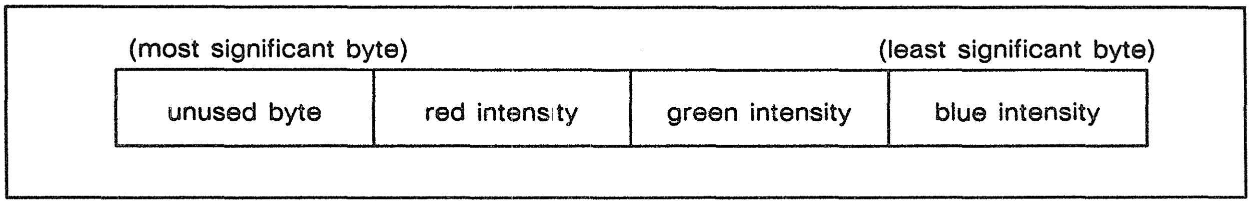
If one color gun projects full intensity (255) and the other two project minimum intensity (0), the pixel will display a primary color. If all three color guns project full intensity (255), the pixel looks white. If all three color guns project minimum intensity (0), the pixel is black (which is the nonilluminated color of the screen).

You cannot directly manipulate the color guns, but you can indirectly control them very easily through the color map.

6.1.2 Overview of the Color Map

Every Apollo color node contains exactly one physical **color map** (which is also called the **color table**). The color map contains a list of all the colors that your program can use. If a color is not stored in the color map, then it cannot be displayed on your monitor.

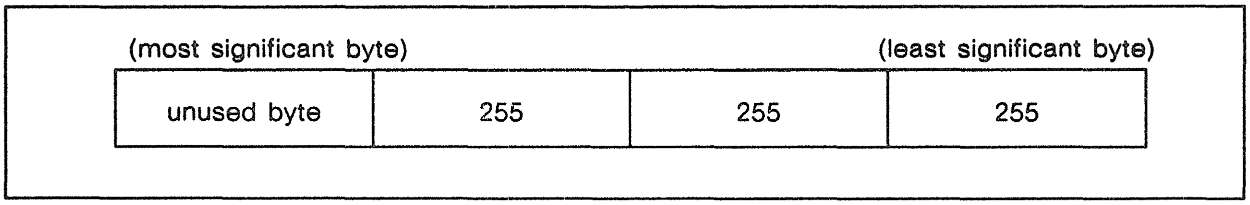
The color map is a 256-element array of four-byte integers. Each element of the array contains one color, or more precisely, each element of the array contains a red color gun intensity, a green color gun intensity, and a blue color gun intensity as shown in Figure 6-1. It is the combination of the three color gun intensities that produces one color.



(most significant byte)			(least significant byte)
unused byte	red intensity	green intensity	blue intensity

Figure 6-1. Format of an Entry in a Color Map

For example, an entry for white in a color map would look as shown in Figure 6-1.



(most significant byte)			(least significant byte)
unused byte	255	255	255

Figure 6-2. A Sample Entry for White in a Color Map

Table 6-1 contains the first 16 elements of a sample color map.

Table 6-1. A Sample Color Map

Color Index	Primary Color Intensities			Resulting Visible Color
	red	green	blue	
0	0	0	0	black
1	255	0	0	red
2	0	255	0	green
3	0	0	255	blue
4	0	255	255	cyan
5	255	255	0	yellow
6	255	0	255	magenta
7	255	255	255	white
8	100	100	100	dark gray
9	220	220	220	light gray
10	225	255	255	off white
11	0	255	255	cyan
12	255	200	255	off white
13	255	0	255	yellow
14	255	255	224	off white
15	255	0	0	red

The size of the color map is the same (256 elements) for every color node. However, the way the node uses the color map depends on the number of planes of display memory supported by the node.

6.1.3 Overview of a Pixel's Planes

Every Apollo node contains a specified number of planes of display memory. All monochromatic models contain only one plane. All current color models contain 4, 8, or 24 planes. But just what is a plane?

The best way to conceptualize planes is to think of them as the z-dimension of a bitmap. We will start with the simplest case: the monochrome node. Here, there is only one plane, so the bit pattern in the x and y dimensions of the display bitmap corresponds simply to the pixel pattern displayed on the monitor as shown in Figure 6-3. Note that 25 bits of display memory correspond to the 25 pixels on the monitor.

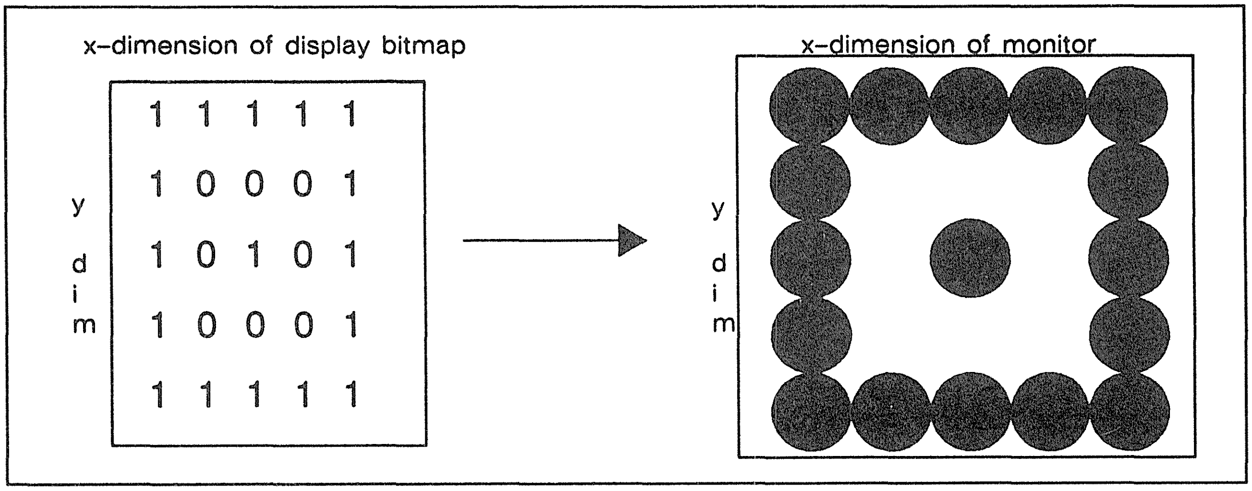


Figure 6-3. Relationship of Monochrome Display Memory Bitmap to Pixels on the Monitor

However, on a node containing many planes, the relationship between the contents of the display bitmap and the images displayed on the monitor is complicated by a third dimension.

On a four-plane node, for example, each pixel on the monitor corresponds to four bits in the display bitmap; that is, each pixel corresponds to one bit from plane 0, one bit from plane 1, one bit from plane 2, and one bit from plane 3. So the 25 pixels on the monitor require 100 bits of display memory. For example, consider the bit values shown in Figure 6-4. The **total plane value** of a pixel is the sum of the four bits comprising it. For example, the pixel in the lower left corner has a total plane value of 13 which comes from $(1x2^0) + (0x2^1) + (1x2^2) + (1x2^3)$. On a four-plane node, the total plane value will always equal a number between 0 and 15 inclusive. The system uses the total plane value as an index into the color map. For example, if a pixel has a total plane value of 5, then the system will paint this pixel with whatever color is stored in slot 5 of the color map. Therefore, the color displayed at a particular pixel depends on both the total plane value and the values in the color map. If we assume for a moment that the color map contains the values shown in Table 6-1, then the monitor will actually display the colors shown in Figure 6-5.

An eight-plane node works in much the same way as a four-plane node. However, on an eight-plane system, the total plane value for each pixel is the sum of the bit values in *eight* planes. Therefore, the total plane value on an eight plane system will fall between 0 and 255 inclusive. Note that 25 pixels on the monitor require 200 bits of display memory on an 8-plane system.

A four-plane node is capable of displaying a maximum of 16 colors at any one time. An eight-plane node is capable of displaying a maximum of 256 colors at any one time.

A 24-plane node requires 24 bits of display memory each pixel. However, a 24-plane system uses a somewhat different method of color lookup than a four-plane or eight-plane node. We detail this method in Section 6.4.

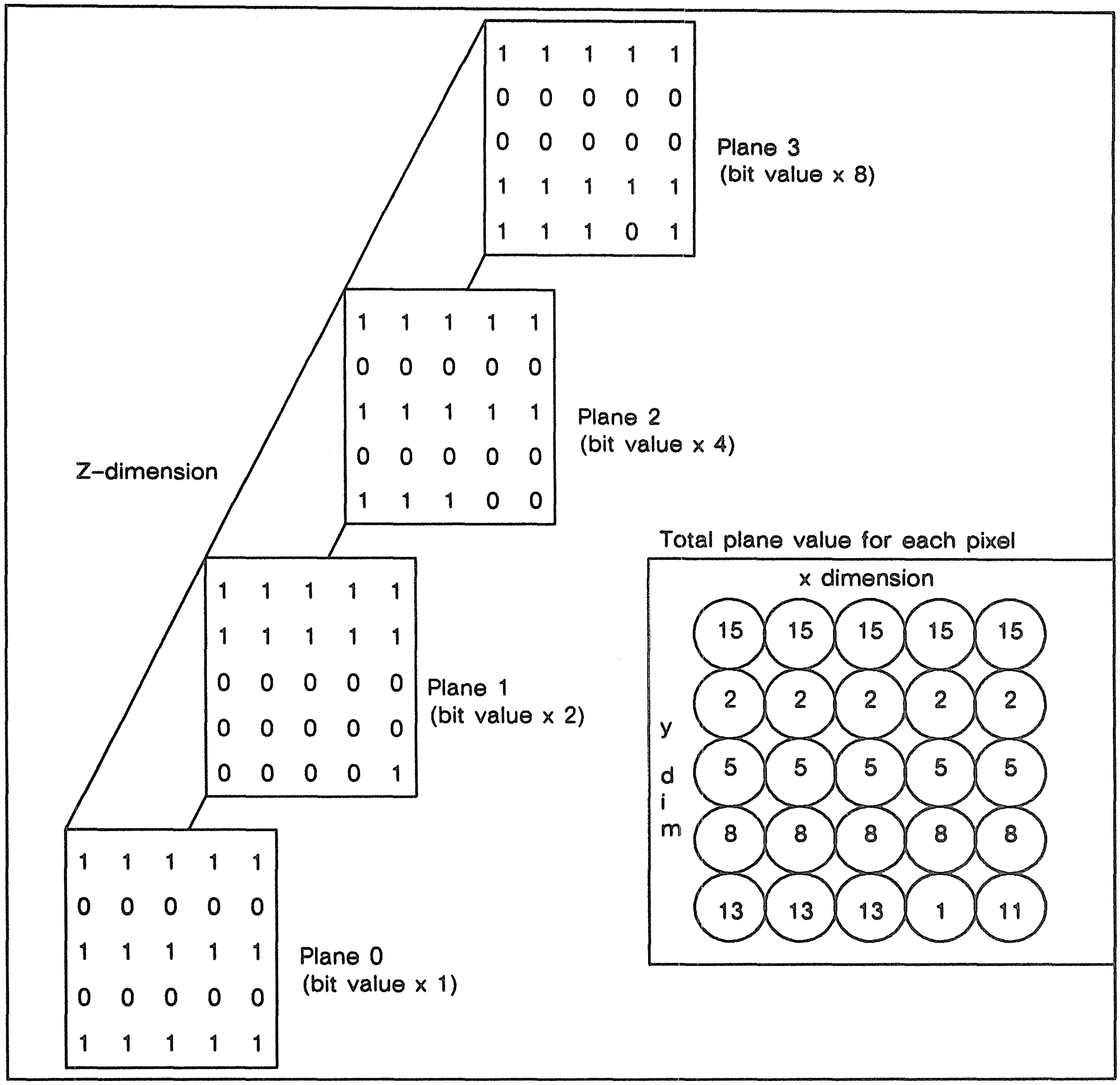
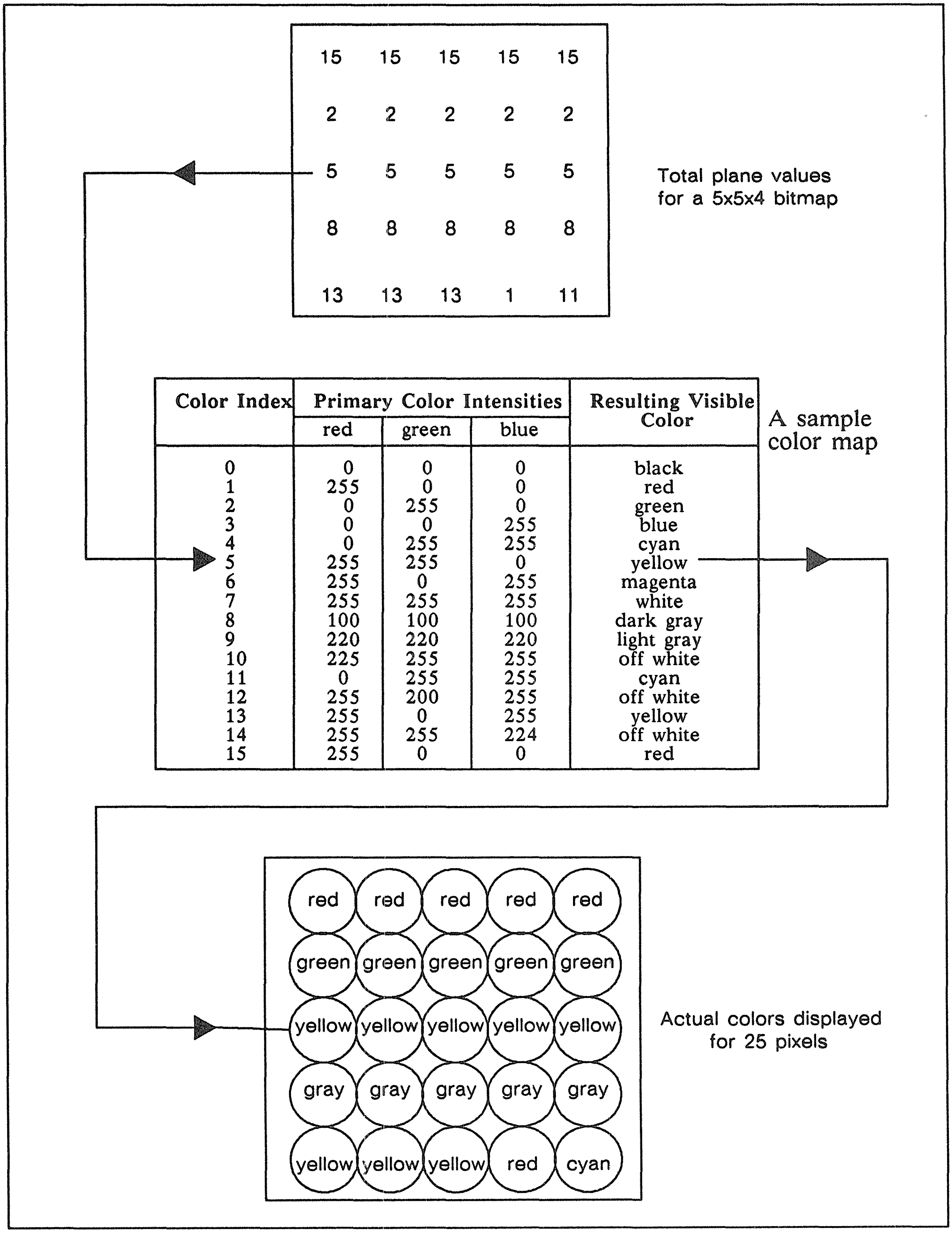


Figure 6–4. Each Pixel on a 4–plane System Is Represented by 1 Bit In Each of 4 Planes.



15	15	15	15	15
2	2	2	2	2
5	5	5	5	5
8	8	8	8	8
13	13	13	1	11

Total plane values for a 5x5x4 bitmap

Color Index	Primary Color Intensities			Resulting Visible Color
	red	green	blue	
0	0	0	0	black
1	255	0	0	red
2	0	255	0	green
3	0	0	255	blue
4	0	255	255	cyan
5	255	255	0	yellow
6	255	0	255	magenta
7	255	255	255	white
8	100	100	100	dark gray
9	220	220	220	light gray
10	225	255	255	off white
11	0	255	255	cyan
12	255	200	255	off white
13	255	0	255	yellow
14	255	255	224	off white
15	255	0	0	red

A sample color map

red	red	red	red	red
green	green	green	green	green
yellow	yellow	yellow	yellow	yellow
gray	gray	gray	gray	gray
yellow	yellow	yellow	red	cyan

Actual colors displayed for 25 pixels

Figure 6-5. The Colors of the Pixels given the Plane Values Shown in Figure 6-4 and the Color Map Shown in Table 6-1.

It would be a difficult programming task to directly manipulate the appropriate bits in all four planes. Fortunately, you do not have to do the calculations because GPR routines will do the calculations for you.

6.1.4 Overview of GPR Routines

In color programming, all GPR routines that produce images are divided into the following three categories:

- Draw routines. (These are all the GPR routines listed in Tables 3-1 at the end of Chapter 3.) The color of draw routines is controlled by the current **draw value**.
- Fill routines. (These are all the GPR routines listed in Table 3-3 at the end of Chapter 3.) The color of fill routines is controlled by the current **fill value**.
- Text routines. (Currently, the only text routine is gpr_$text.) The color of text routines is controlled by the current **text value**.

For example, the gpr_$line routine draws a line between two points. To implement gpr_$line on a monochrome node, the system writes ones in every bit in the bitmap that corresponds to the points on the line. Implementing gpr_$line on a color node is more complicated. In this case, the system must set the total plane value of the affected points equal to the current draw value.

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

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

6.2 GPR Routines Controlling Color Values

GPR provides the following calls to set or inquire about the color in various output operations:

gpr_$set_draw_value	Specifies the color index to use for all subsequent draw and line operations listed in Table 3-1 and Table 3-2 at the end of Chapter 3.
gpr_$inq_draw_value	Returns the current draw color index.
gpr_$set_text_value	Specifies the color index to use for all subsequent strings written with gpr_$text.
gpr_$set_text_background_value	Specifies the color index to use as background for all subsequent strings written with gpr_$text.
gpr_$inq_text_value	Returns the current text color and text background color index.
gpr_$set_fill_value	Specifies the color index to use for all subsequent fill operations listed in Table 3-3.
gpr_$inq_fill_value	Returns the current fill color index.
gpr_$clear	Sets every pixel in the current bitmap to the specified color index.

For example, consider the gpr_$set_draw_value. This call takes a color index as an argument. After you call gpr_$set_draw_value, every line, curve, spline, arc, and circle will be drawn with this color index *until* you call gpr_$set_draw_value again and specify a different color index. That is, if you want every line, curve, spline, arc, and circle to be drawn with exactly the same color, then you should call gpr_$set_draw_value only once.

The following example demonstrates the four gpr_$set routines:

```
Program setting_colors_in_figs;
{This program controls the colors used to draw a line, fill a triangle, and
 print text.  It demonstrates the gpr_$set_draw_value, gpr_$set_fill_value,
 gpr_$set_text_value, and gpr_$set_text_background_value.  Note that this
 program does not establish a new color map; therefore, the colors displayed
 on your screen will depend on whatever is stored in the current color map.}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{***********************************************************************************}
PROCEDURE draw_a_colored_line;
BEGIN
   gpr_$set_draw_value(1, status);
   gpr_$line(90, 150, status);
END;
{***********************************************************************************}
PROCEDURE fill_a_triangle_with_color;
VAR
  v : static array[1..3] of gpr_$position_t :=[[100,100],[100,300],[300,100]];
BEGIN
  gpr_$set_fill_value(2, status);
  gpr_$triangle(v[1], v[2], v[3], status);
END;
{***********************************************************************************}
PROCEDURE write_text_in_color;
VAR
   font_id : integer16;
BEGIN
   gpr_$load_font_file('/sys/dm/fonts/f9x15', 19, font_id, status);
   check('loading the font');
   gpr_$set_text_font(font_id, status);
   gpr_$set_text_value(3, status);  {write text with color index 3.}
{The default text background index is 0.}
   gpr_$move(20, 320, status);
   gpr_$text('Greetings from the written world', 32, status);

{But we can establish a nondefault text background index.}
   gpr_$set_text_background_value(7,status);
   gpr_$move(20, 350, status);
   gpr_$text('Greetings from the written world', 32, status);
END;
{***********************************************************************************}
BEGIN
   init(gpr_$borrow);
   draw_a_colored_line;
   fill_a_triangle_with_color;
   write_text_in_color;
   pause(5.0);  {Pause 5 seconds, then terminate.}
   gpr_$terminate(false, status);
END.
```

6.3 The Color Map in Detail

Earlier in this chapter, we provided a tutorial on the color map. In this section, we detail some of the more advanced concepts of the color map. You should also read the descriptions of the color table manager (CTM) routines in the *Domain System Call Reference.*

6.3.1 The Default Color Map in /sys/dm/color_map

Whenever you load the Display Manager (e.g., after shutting down and rebooting), the Display Manager loads color values into all 256 elements of the color map. To do this, the Display Manager uses the color values stored in the ASCII file named /sys/dm/color_map.

This file is part of the standard software package that comes with the Display Manager. The file's contents look like this:

```
0 0 0 0
1 255 0 0
2 0 255 0
3 0 0 255
4 0 255 255
5 255 255 0
6 255 0 255
7 255 255 255
8 240 255 190
9 30 155 0
10 255 240 180
11 210 20 0
12 255 255 230
13 60 60 200
14 240 240 170
15 150 90 0
```

Notice that each line contains four integers. The first integer is the color slot, the second integer is the color intensity for red, the third integer is the color intensity for green, and the fourth integer is the color intensity for blue. For example, color slot 3 contains no red, no green, and full blue. A complete summary of the default color map is printed in Table 6–2.

On an eight-plane node, the Display Manager always loads the color indices 0 0 0 (which corresponds to the color black) into slots 16 through 255 of the color map.

If you don't like the default color map we provide, you can create your own default color map. To change it, all you have to do is to edit file `/sys/dm/color_map` and then reload the Display Manager. Be careful with entries 8 through 15. These entries control the border color (slots 8, 10, 12, and 14) and fill color (slots 9, 11, 13, and 15) for Display Manager windows. If you don't provide suitable contrast between fill and border, window boundaries may become difficult to see. You should also be careful with slot 0. The Display Manager uses slot 0 as the background color for the parts of the screen not covered by windows. It also uses slot 0 as the text color in windows. The Display Manager uses slot 1 for the cursor color.

You can only use /sys/dm/color_map to control the first 16 slots of the color map. For example, you cannot use /sys/dm/color_map to load slot 200 of the color map on an eight-plane node.

Table 6-2. Default Color Map

Color Index	Primary Color Intensities			Constant Symbolizing These Intensities	Resulting Visible Color
	red	green	blue		
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	240	255	190	(no constant)	off white
9	30	155	0	(no constant)	dark green
10	255	240	180	(no constant)	off white
11	210	20	0	(no constant)	dark red
12	255	255	230	(no constant)	off yellow
13	60	60	200	(no constant)	dark blue
14	240	240	170	(no constant)	off white
15	150	90	0	(no constant)	brown
16 - 255	0	0	0	gpr_$black	black

6.3.2 How to Alter the Color Map in Your GPR Program

By default, your GPR program uses whatever colors are currently stored in the color map. You can change one, some, or all slots in the color map by calling the gpr_$set_color_map routine.

The following program demonstrates how to create a new color map for a GPR program. This program draws seven concentric circles. The outermost circle is drawn first in dark blue. Each additional circle is drawn in a lighter shade of blue except the innermost, 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 amounts of blue. Lighter shades of blue have increasing amounts of red and green with the maximum amount of blue.

```
Program color_circles;
{This program demonstrates how to build a new color map. It demonstrates
 the gpr_$set_color_map call.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{***************************************************************************}
FUNCTION  color_entry(IN red : integer16;
                      IN green : integer16;
                      IN blue : integer16) : integer32;
BEGIN
    color_entry :=lshft(red,16) ! lshft(green,8) ! blue;
END;
{***************************************************************************}
PROCEDURE create_color_map;
VAR
    start_index : static integer16 := 0;
    number_of_entries : static integer16 := 8;
    color_map : gpr_$color_vector_t;
BEGIN
    color_map[0] := color_entry (0,0,0); {color--black}
    color_map[1] := color_entry (0,0,255); {color--dark blue}
    color_map[2] := color_entry (50,50,255);
    color_map[3] := color_entry (75,75,255);
    color_map[4] := color_entry (100,100,255);
    color_map[5] := color_entry (150,150,255);
    color_map[6] := color_entry (200,200,255); {color--light blue}
    color_map[7] := color_entry (255,255,255); {color--white}

{Now that the array 'color_map' contains the correct values, establish it
 as the real system color map.}
    gpr_$set_color_map(start_index,number_of_entries, color_map, status);
END;
{***************************************************************************}
PROCEDURE draw_concentric_circles;{Draw 7 concentric circles - each a lighter}
VAR                               {color of blue. The last circle is white.}
    center      : static gpr_$position_t := [300,300];
    radius      : static integer := 300;
    index       : 1..7;
BEGIN
    for index := 1 to 7 do begin
      gpr_$set_fill_value(index,status);
      gpr_$circle_filled(center,radius,status);
      radius := radius - 40;
    end;
END;
{***************************************************************************}
BEGIN
   init(gpr_$borrow);
   create_color_map;
   draw_concentric_circles;

   pause(5.0); {Pause 5 seconds, then terminate.}
   gpr_$terminate(false, status);
END.
```

Note that you must pass an array of 32-bit integers as the third argument to gpr_$set_color_map. Therefore, you must take care to ensure that bits 0 through 7 contain the blue intensity, bits 8 through 15 contain the green intensity, and bits 16 through 23 contain the red intensity. To ensure that the intensities get in the correct bit positions, you can

- Use the LSHFT function in Pascal. For example, LSHFT(red,16) + LSHFT(green,8) + blue.
- Use the bit shift operator << in C. For example, (red << 16) + (green << 8) + blue.
- Multiply by constants in FORTRAN. For example, (red * 16#010000) + (green *16#000100) + blue.

NOTE: Some programmers like to call gpr_$set_wait_frame before setting the color map, so that the system will wait until the end of a refresh cycle before changing the color map.

6.3.3 The Color Map in Different Display Modes

The way your program uses the color map depends, in part, on the program's display mode.

Borrow mode is the safest place to run a program that calls gpr_$set_color_map. When you start a borrow mode program, the system automatically stores the values of the color map for safekeeping. When you terminate the program, the system automatically restores the original values. Therefore, any changes made to the color map during the execution of a borrow mode program will have no effect on the color map after the program ends.

By contrast, manipulating the color map in direct mode can potentially cause some problems. When you start a direct mode program, the system does not store the values of the color map (though you can). If you change any slots in the color map during the execution of the color map, then those values will remain in the color map even *after* the program terminates. In fact, these values will remain in the color map until you reload the Display Manager or run another direct mode program that manipulates the color map.

Given the preceding warnings, we offer the following advice: your direct mode program should call gpr_$inq_color_map before calling gpr_$set_color_map. The gpr_$inq_color_map routine stores the values of some or all of the current color map as an array. When you are finished with the special color map you created for your direct mode program, you merely reload (with gpr_$set_color_map) the array containing the original color map. The following example demonstrates this technique:

```
Program restore_color_table;
{This program demonstrates how to save the current color map, create a new
 color map, and then restore the original color map.  It demontrates the
 gpr_$inq_color_map and gpr_$set_color_map routines.}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
VAR
    start_index : integer16 := 0;
    number_of_entries : integer16 := 16;
    old_color_map, new_color_map : gpr_$color_vector_t;
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{****************************************************************************}
PROCEDURE draw_circles;  {Draw 16 circles to illustrate the current contents}
VAR                      {of the color map.}
    n : integer16;
    center: gpr_$position_t;
BEGIN
    center.x_coord := 50;
    for n := 0 to 15 do begin
        gpr_$set_fill_value(n, status);
        center.y_coord := 25*n;
        discard(gpr_$acquire_display(status));
        gpr_$circle_filled(center, 15, status);
        gpr_$release_display(status);
    end;
END;
{****************************************************************************}
PROCEDURE save_old_color_map;  {Save the color map the system was using when }
BEGIN                          {the program was called. }
    gpr_$inq_color_map(start_index, number_of_entries, old_color_map, status);
    check('inquiring about color map');
END;
{****************************************************************************}
PROCEDURE create_new_color_map; {Load a new color map.}
VAR
   n : integer16;
BEGIN
    new_color_map[0] := 0; {Keep color slot 0 black.}
    for n := 1 to 15 do
      new_color_map[n] := lshft(15*n,16) ! lshft(255,8) ! lshft(255-(15*n),0);
    discard(gpr_$acquire_display(status));
    gpr_$set_color_map(start_index, number_of_entries, new_color_map, status);
    gpr_$release_display(status);
END;
{****************************************************************************}
PROCEDURE restore_old_color_map;  {reload the color map the system was using}
BEGIN                             {before the program was executed.}
    discard(gpr_$acquire_display(status));
    gpr_$set_color_map(start_index, number_of_entries, old_color_map, status);
    gpr_$release_display(status);
END;
{****************************************************************************}
BEGIN
   init(gpr_$direct);
   save_old_color_map;      draw_circles;    pause(2.0);
   create_new_color_map;                     pause(2.0);
   restore_old_color_map;                    pause(2.0);
```

```
    gpr_$terminate(false, status);  {Terminate.}
  END.
```

6.3.4 Color Map for Monochromatic Displays

Some monochromatic nodes (namely, those with the gpr_$bw_800x1024 and gpr_$bw_1024x800 display configuration types) support a two-entry color map in hardware. Other monochromatic nodes (those with the gpr_$bw_1280x1024 display configuration type) simulate a two-entry color map with software.

On monochromatic devices that have the color map in hardware, a pixel value is used as an index into the color map, and the color of the pixel is determined by the color value in the color map. The default hardware color map appears in Table 6-3.

Table 6-3. Default Color Map for Monochromatic Displays

Color Index	Color Value	Constant Symbolizing This Value	Resulting Color
0	0	gpr_$black	black
1	16#FFFFFF	gpr_$white	white

On monochrome display devices that simulate a color map with software, a pixel value of 0 is always dark, and a pixel value of 1 is always painted white.

You can determine whether a monochromatic device simulates an inverted color map or has a color map in hardware by calling the gpr_$inq_disp_characteristics routine and examining the value returned into the invert field of the gpr_$disp_char_t record.

On a monochromatic node, there are no other choices besides black and white; that is, you cannot grey-scale output.

6.3.5 Color Map on External File Bitmaps

When you BLT a color external file bitmap to display memory, the system displays the bitmap using the colors currently stored in the color map. However, the colors currently stored in the color map may be substantially different than the colors expected when the external file bitmap was created. For this reason, GPR provides a way of storing a special color map inside an external file bitmap. Note that this is *not* the system color map. Rather, it is simply a storage place for color values that can optionally be loaded into the color map.

Use the gpr_$set_bitmap_file_color_map routine to store color values inside an external file bitmap. When you want to display the external file bitmap (by BLTing it to display memory), you call the gpr_$inq_bitmap_file_color_map to load the color values into an array. Then, call the gpr_$set_color_map routine to establish this array as the actual system color map. The following program demonstrates this technique:

```
Program plane_oriented_bitmaps;
{This program creates a plane-oriented bitmap.  It demonstrates the
 gpr_$open_bitmap_file, gpr_$set_bitmap_file_color_map, gpr_$set_color_map,
 and gpr_$inq_bitmap_file_color_map routines.  Run this program twice.
 The first time, select the 'create' option so that the program will create an
 external file bitmap.  The second time you run it, select the 'display'
 option so that the program will display the contents of this file on your
 screen.  You should run this program on a color node.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

TYPE
   color_range   = 0..255;
CONST
   x_size_of_bitmap = 200;
   y_size_of_bitmap = 200;
   n_of_colors      = 16; {The color map will contain this many colors.}
   draw_value       = 15;
VAR
   choice              : (create, display);
   pathname_of_bitmap : name_$pname_t;
   attribs             : gpr_$attribute_desc_t;
   disk_bitmap         : gpr_$bitmap_desc_t;

%include 'my_include_file.pas';{Contains the init, pause, and check routines.}
{*******************************************************************************
}
PROCEDURE access_external_bitmap(IN access : gpr_$access_mode_t);
VAR
   pathname_len : 1..name_$pnamlen_max;
   version  : gpr_$version_t;
   size     : gpr_$offset_t;
   groups   : integer16;
   header   : gpr_$bmf_group_header_array_t;
   created  : boolean;
BEGIN
   for pathname_len := 1 to name_$pnamlen_max do
      if pathname_of_bitmap[pathname_len] = ' ' then exit;
   pathname_len := pathname_len - 1;                            {pathname_len}
   version.major := 1;   version.minor := 1;                         {version}
   size.x_size := x_size_of_bitmap;size.y_size := y_size_of_bitmap;  {size}
   groups      := 1;                                                  {groups}
   header[0].n_sects :=  hi_plane + 1;                      {start of header}
   header[0].pixel_size     := 1;
   header[0].allocated_size := 0;
   header[0].bytes_per_line := 0;
   header[0].bytes_per_sect := 0;
   header[0].storage_offset := nil;                           {end of header}

   gpr_$open_bitmap_file(access, pathname_of_bitmap, pathname_len,
                         version, size, groups, header, attribs,
                         disk_bitmap, created, status);
   check('opening an external bitmap');
   writeln('allocated_size = ', header[0].allocated_size);
END;
{******************************************************************************}
```

```
Procedure DRAW_FIGURE_IN_FILE_BITMAP;  {Draw a line in the file bitmap.}
{You can use the GPR line calls to draw lines in a plane-oriented bitmap;
 however, these calls won't work in a pixel-oriented bitmap.}
BEGIN
   gpr_$set_bitmap(disk_bitmap, status);
   gpr_$set_draw_value(draw_value, status);
   gpr_$line(150, 150, status);
END;
{***************************************************************************}
Procedure CREATE_BITMAP_FILE_COLOR_MAP;
VAR
   counter          : integer16;
   blue_component   : static color_range := 0;
   red_component    : static color_range := 255;
   green_component  : color_range;
   color_map        : array[0..(n_of_colors-1)] of gpr_$pixel_value_t;
BEGIN
   color_map[0] := 0; {Keep slot 0 black}
   for counter := 1 to (n_of_colors - 1) do begin
      green_component := counter*16;
      color_map[counter] := ((red_component   * 16#010000) +
                             (green_component * 16#000100) +
                             (blue_component  * 16#000001));
   end;
{We now establish the array as the color map for the external file bitmap.}
   gpr_$set_bitmap_file_color_map(disk_bitmap,0,n_of_colors,color_map,status);
   check('setting the bitmap file color map');
END;
{***************************************************************************}
Procedure DISPLAY_THE_BITMAP;
VAR
   color_map     : array[0..255] of gpr_$pixel_value_t;
   source_window : static gpr_$window_t := [[0,0][x_size_of_bitmap,
                                                 y_size_of_bitmap]];
   dest_origin   : static gpr_$position_t    := [400,400];
BEGIN
   ACCESS_EXTERNAL_BITMAP(gpr_$readonly);  {open the file for reading.}

{Load the external file color map into array variable color_map.}
   gpr_$inq_bitmap_file_color_map(disk_bitmap,0,n_of_colors,color_map,status);

{Load the colors from array variable color_map into the system color map.}
   gpr_$set_color_map(0, n_of_colors, color_map, status);

{BLT the external file bitmap to display memory.}
   gpr_$set_bitmap(display_bitmap, status);
   gpr_$pixel_blt(disk_bitmap, source_window, dest_origin, status);
   check('blting from disk to display');
END;
{***************************************************************************}
BEGIN   {MAIN Procedure}
   writeln('Do you want to create a bitmap or display a bitmap?');
   write('(Enter ''create'' or ''display'') -- ');     readln(choice);
   if choice = create
     then writeln('Be patient; it will take some time to create the bitmap.');
   writeln('What is the pathname of the file you want to ', choice:1, ' -- ');
   readln(pathname_of_bitmap);
   INIT(gpr_$borrow);

   if choice = create
      then begin
               gpr_$allocate_attribute_block(attribs,status);
               ACCESS_EXTERNAL_BITMAP(gpr_$create);
               CREATE_BITMAP_FILE_COLOR_MAP;
               DRAW_FIGURE_IN_FILE_BITMAP;
           end
      else begin
               DISPLAY_THE_BITMAP;
               PAUSE(5.0);
           end;

   gpr_$terminate(false, status);
END.
```

6.4 24–Plane (True–Color) Programming

The DN590 is the first full-feature Apollo "true-color" node. Previous Apollo color nodes were "pseudo-color" nodes (though a special imaging format on some nodes did permit a very limited kind of true-color).

What is the difference between a pseudo-color node and a true-color node? On a pseudo-color node, the color value associated with a particular pixel is an index (i.e., a pointer) into a particular location on a color chart. The chart location contains the actual red, green, and blue intensities for that pixel. On a true-color node, the color value associated with a particular pixel *is* the actual red, green, and blue intensities for that pixel.

What is the practical difference between a pseudo-color node and a true-color node? Consider Table 6-4. It shows that the practical difference between pseudo-color and true-color is that a true-color node can simultaneously display far more colors than a pseudo-color node. The greater number of colors is important in many applications, for instance, in programs requiring accurate smooth shading over a wide color range. True-color uses enough colors so that the eye can perceive a digitized picture as a realistic representation of the image.

Table 6-4. Color Capabilities of Various Apollo Nodes

Node	Type of Display	Number of Planes	Max. number of colors on screen at one time	Total number of colors to choose from
DN3000C	pseudo-color	4	$2^4 = 16$	$2^{24} =$ 16.7 million
DN570/580	pseudo-color	8	$2^8 = 256$	$2^{24} =$ 16.7 million
DN590	true-color	24	$2^{24} =$ 16.7 million*	$2^{24} =$ 16.7 million

* Although the display hardware is capable of showing up to 16.7 million colors, the DN590 monitor contains only 1.3 million pixels, so it would be impossible to display more than 1.3 million colors at one time. A monitor containing an infinite number of pixels would be able to represent up to 16.7 million colors at one time.

6.4.1 The Two Video Modes of the DN590

As mentioned earlier, the DN590 contains 24-planes of display memory. However, there may be times when it is advantageous to use only 8 planes. Therefore, you can set the DN590 to run in either of the following two hardware "video modes":

- 8-plane (pseudo-color) hardware video mode. This is the default mode.
- 24-plane (true-color) hardware video mode.

In 8-plane hardware video mode, the value stored in planes 0 through 7 provides an index to a color chart. The remaining 16 planes of pixel depth are not used to drive video. 8-plane hardware video mode permits double-buffering.

In 24-bit true-color video mode, the 24 planes of pixel depth are also divided into three 8-plane banks of memory. However, in this mode, each bank contains the actual intensity of one of the monitor's primary colors (red, green, and blue). 24-plane hardware video mode prohibits double-buffering.

6.4.2 How to Set the Hardware Video Mode

You can control the video modes of the DN590 through a Display Manager command; you *cannot* control the video modes through a GPR call.

The Display Manager command that controls the video mode is called CDM (an acronym for Change Display Mode). It takes the following format:

```
CDM {-p 1 | -p 8}
```

To put the DN590 in 24-bit true-color video mode, you should specify

```
CDM -p 1
```

To put the DN590 in 8-bit pseudo-color video mode, you should specify:

```
CDM -p 8
```

If you specify CDM without an argument, it is equivalent to specifying CDM -p 8.

6.4.3 Pseudo-Color and True-Color Compatibility

It is not possible for the DN590 to simultaneously run in both hardware video modes. Therefore, what happens when you try to run pseudo-color programs in 24-plane hardware video mode? Furthermore, what happens when you try to run true-color programs in 8-plane hardware video mode? Table 6-5 summarizes the results.

Table 6-5. The Effect of Initialization Mode and Video Mode on Program Display

Program's initialization mode	Current hardware video mode	Program is displayed as
gpr_$frame	8	pseudo-color
gpr_$frame	24	monochrome
gpr_$borrow	8	pseudo-color
gpr_$borrow	24	pseudo-color
gpr_$direct	8	pseudo-color
gpr_$direct	24	monochrome
gpr_$borrow_rgb	8	true-color
gpr_$borrow_rgb	24	true-color
gpr_$direct_rgb	8	pseudo-color
gpr_$direct_rgb	24	true-color

Let's examine this table in more detail.

A borrow mode takes precedence over the current hardware video mode. For example, even if the current hardware video mode is 24, an initialization mode of gpr_$borrow will cause the program to be displayed in pseudo-color.

A frame or direct mode is a little more complex because of the potential for conflicts with other graphics programs. In general, if you want to display in pseudo-color, you should initialize to a non-RGB mode and set the hardware video mode to 8. Similarly, if you want to display in true-color, you should initialize to a RGB mode and set the hardware video mode to 24.

> NOTE: If a program requires a certain hardware video mode, you should call the CDM command *prior* to invoking the program. Your program should *not* attempt to change the hardware video mode.

Running Pseudo-Color Programs on a DN590

You do not have to recode or recompile any existing pseudo-color programs to run on a DN590 (assuming that you run the program in 8-plane hardware video mode). Your programs should run on the DN590 at very close to the same speed they would run on a DN580.

If you do recompile your pseudo-color program, your program should still work.

6.5 Writing True-Color Programs

This section explains what you need to know in order to write true-color programs. Remember these fundamental points as you make the transition from pseudo-color programmer to true-color programmer; namely:

- A true-color program must be initialized to one of the special RGB display modes.
- A true-color direct-mode program that contains a refresh procedure should definitely not change the hardware video mode at any time during the program's execution.
- A true-color program accesses 24 planes instead of 8.
- A true-color program accesses the color map differently than a pseudo-color program.

We detail these points separately in the following sections.

RGB Display Modes

In order to run a true-color program, you must specify one of the RGB (true-color) display modes when you initialize with gpr_$init. The three RGB display modes are:

gpr_$direct_rgb for running true-color programs in a window.

gpr_$borrow_rgb for running true-color programs while the entire display is borrowed.

gpr_$borrow_rgb_nc same as gpr_$borrow_rgb except that the system does not clear the screen when GPR is initialized.

Refresh Procedures

In general, you can write a refresh procedure for a true-color program using the same techniques that you would use to write a refresh procedure for a pseudo-color program. If your program does contain a refresh procedure, you should definitely not change the hardware video mode at any time during the program's execution.

Using 24-Planes

The DN590 has 24 planes accessible to the user. Many GPR calls (e.g., gpr_$init, gpr_$allocate_bitmap) take an input parameter named hi_plane which should be set equal to the number of accessible planes minus one. Therefore, if you are writing a true-color program, you must set hi_plane to 23 in *every* call requiring this parameter. If you set hi_plane to 23 in a gpr_$init call, but then set it equal to 7 somewhere else in the program, your program will not execute properly.

The gpr_$set_plane_mask call assumes an 8 plane system. We have not changed this call. However, we have added a new call named gpr_$set_plane_mask_32 to perform masking operations on all 24 planes.

Using the Color Map

The color map of all Domain color nodes is exactly the same size; namely, a 256-element array of 32-bit integers. However, a true-color program accesses the color map in a different way than a pseudo-color program. The color map of a true-color program can be thought of as a 256x3byte 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.

Consider, for example, a true-color program running in gpr_$borrow_rgb mode. When you initialize a program in gpr_$borrow_rgb mode, the system automatically loads the linear ramp values into the color map. This means that the red intensity, green intensity, and blue intensity are always identical to the index. After loading the linear ramp values, the color map looks as shown in Table 6-6.

Table 6-6. The Linear Ramp Color Map

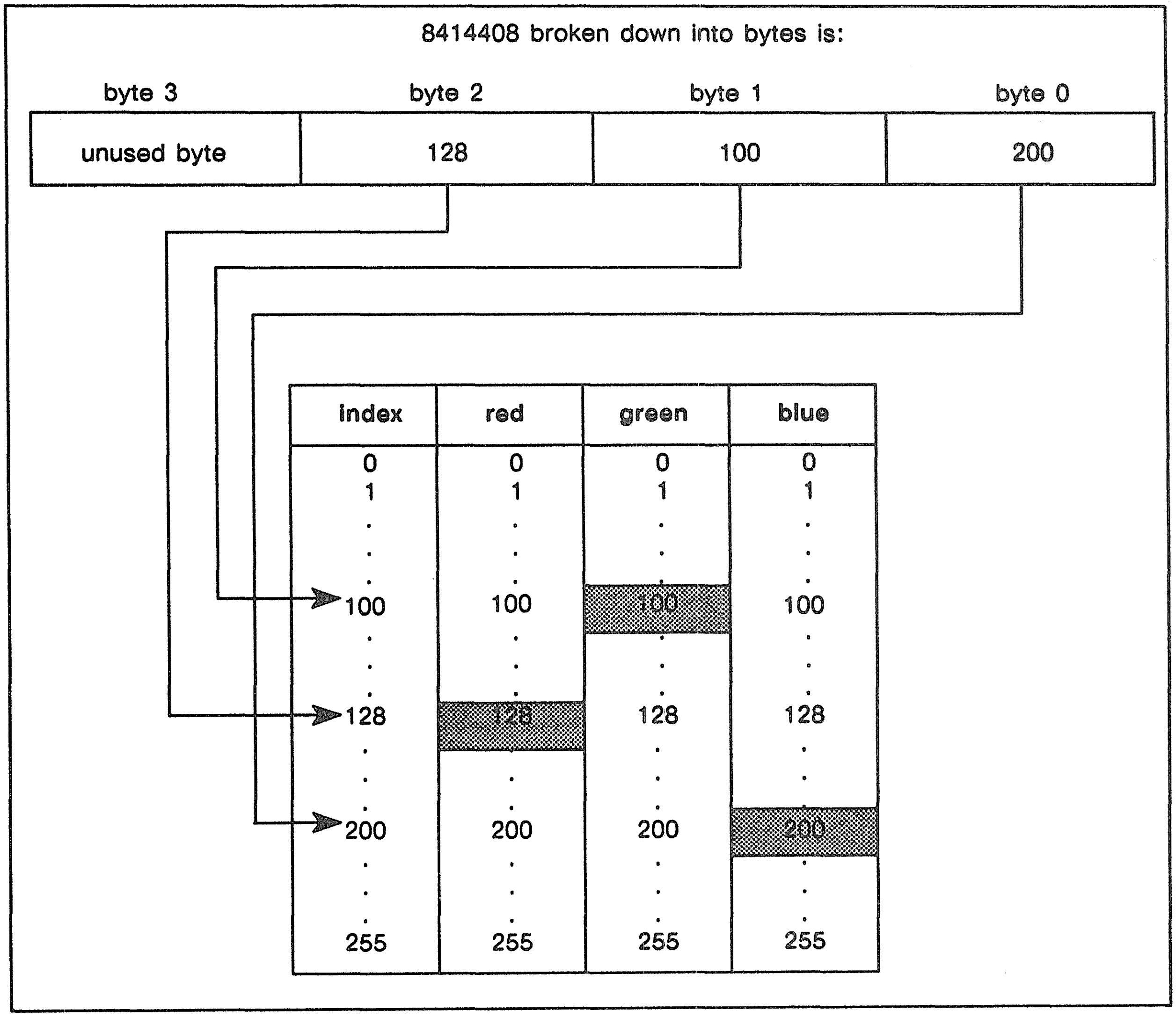
index	red	green	blue
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
.	.	.	.
.	.	.	.
.	.	.	.
255	255	255	255

Now, let's examine how the system reads the color map. Suppose you specify a draw value of 8414408 (which is 008064C8 in hexadecimal). A draw value of 8414408 would far exceed the pseudo-color limit of 255; however, for a true-color program, any number in the range 0 to 16.7 million is acceptable for a draw value, fill value, text value, etc. The system views the draw value 8414408 as shown in Figure 6-6.

	(unused byte)	(red index)	(green index)	(blue index)	
	byte 3	byte 2	byte 1	byte 0	
binary	00000000	10000000	01100100	11001000	
decimal	000	128	100	200	= 8414408
hex	00	80	64	C8	= 008064C8

Figure 6-6. How a Draw Value Breaks Down Into Bytes

As mentioned earlier, the color map is byte addressable for true-color programs. The system, therefore, views the draw value not as one index, but as three separate indices: index 128, index 100, and index 200. The system will obtain the red intensity by looking up index 128 in the first column of the color map. The system will obtain the green intensity by looking up index 100 in the second column of the color map. The system will obtain the blue intensity by looking up index 200 in the third column of the color map. The resulting three primary color intensities will produce the color of the line. By no coincidence, the red byte of index 128 *is* 128, the green byte of index 100 *is* 100, and the blue byte of index 200 *is* 200. Therefore, the draw value of 8414408 *is* the actual color of the line. The entire process is illustrated in Figure 6-7.



8414408 broken down into bytes is:

byte 3	byte 2	byte 1	byte 0
unused byte	128	100	200

index	red	green	blue
0	0	0	0
1	1	1	1
.	.	.	.
.	.	.	.
.	.	.	.
100	100	▨ 100	100
.	.	.	.
.	.	.	.
.	.	.	.
128	▨ 128	128	128
.	.	.	.
.	.	.	.
.	.	.	.
200	200	200	▨ 200
.	.	.	.
.	.	.	.
.	.	.	.
255	255	255	255

Figure 6-7. True-color Color Map Look-up

Now consider a different color, namely, yellow. Yellow is composed of a red intensity of 255, a green intensity of 255, and a blue intensity of 0. Therefore, the true-color value that corresponds to yellow is 16776960_{10} or $00FFFF00_{16}$. To specify a true-color draw value of yellow, your program would include the following line:

```
gpr_$set_draw_value(16#00FFFF00, status)
```

The Color Map in Different Display Modes

As we mentioned earlier, if you initialize in gpr_$borrow_rgb mode, the system automatically loads the linear ramp into the color map regardless of the current hardware video mode.

In gpr_$direct_rgb mode the system does not automatically load the linear ramp (or any default values) into the color map. However, before running a gpr_$direct_rgb mode program, you must set the hardware video mode to 24-plane (with CDM -p 1). When you issue the CDM -p 1 command, the system automatically loads the linear ramp into the color map. Once in 24-plane hardware video mode,

- a program initialized in gpr_$direct_rgb mode can call gpr_$set_color_map to change the color map.

- a program initialized in gpr_$direct mode can call gpr_$set_color_map, but the system will not honor the call. That is, the call neither generates an error nor changes the color map.

Note that most programs running in gpr_$borrow_rgb or gpr_$direct_rgb mode will not have to call gpr_$set_color_map. (Though some applications may do it because they require a gamma correction.)

If you return to 8-plane hardware video mode from 24-plane hardware video mode, the system automatically reloads the color map that was in effect when you issued CDM -p 1. (Note that is not necessarily the default color map.)

A True-Color Example

The following program demonstrates true-color programming techniques:

```
Program lines_in_true_color;
{This program demonstrates true-color programming.  It draws 1020 lines, each
 line having a different color.  Notice that we never call gpr_$set_color_map.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

CONST
   left_end_of_line  = 0;
   right_end_of_line = 1000;
VAR
   color_total : integer32;
   r : integer16;
   y_coord_of_line : integer16 := 0;
   red_component, green_component, blue_component : 0..255;

%include 'my_include_file.pas';{contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow_rgb);
   green_component := 255;

{This program draws 1020 lines, each line a different color.}
   for r := 1 to 4 do begin
        red_component := (r * 64) - 1;  {red_component = 63,127,191,255}
        for blue_component := 1 to 255 do begin

{Create a unique draw value.}
              color_total := ( lshft(red_component,16)
                             ! lshft(green_component,8)
                             ! lshft(blue_component,0));
              gpr_$set_draw_value(color_total, status);
```

```
{Draw the line with this draw value.}
              y_coord_of_line := y_coord_of_line + 1;
              gpr_$move(left_end_of_line, y_coord_of_line, status);
              gpr_$line(right_end_of_line, y_coord_of_line, status);
       end;
   end;

{Pause 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

We picked gpr_$set_draw_value in the examples because it was simpler. However, you would use the same techniques for gpr_$set_fill_value, gpr_$set_text_value, gpr_$set_text_background_value, and gpr_$set_fill_background_value.

6.6 Color Zoom Operations

Some Domain color displays have a hardware zoom feature to make an image larger. This feature only works on certain color displays (see Appendix D) and only in borrow mode. The gpr_$inq_disp_characteristics returns the maximum magnification possible on the target node.

The zooming is done by pixel replication. The zoom always starts at the upper left corner of the screen.

The GPR call that accesses this hardware feature is called gpr_$color_zoom. 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 6-8). If desired, you can keep the aspect ratio equal by making x and y equal.

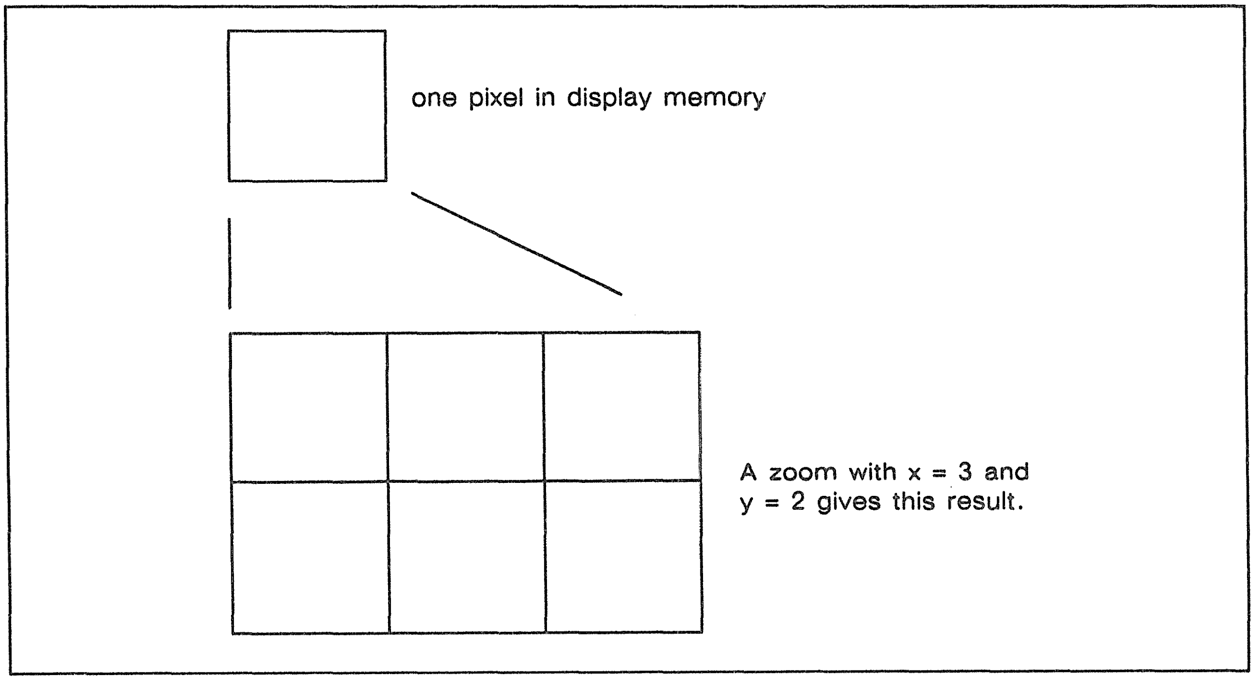


Figure 6-8. Color Zoom

The following program demonstrates the zoom feature:

```
Program zooming;
{This program demonstrates how to magnify (zoom) the display bitmap.
 It uses the gpr_$color_zoom call.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   x_mag_factor, y_mag_factor : integer16;
   center : gpr_$position_t := [100,100];
   radius : integer16 := 50;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{****************************************************************************}
BEGIN
   write('Enter the magnification factor in the X dimension -- ');
   readln(x_mag_factor);
   write('Enter the magnification factor in the Y dimension -- ');
   readln(y_mag_factor);

   init(gpr_$borrow);

   gpr_$circle_filled(center, radius, status);
   pause(1.0);

   if (x_mag_factor > display_characteristics.x_zoom_max)
      then writeln('X magnification factor is too high for this node.')
   else if (y_mag_factor > display_characteristics.y_zoom_max)
      then writeln('Y magnification factor is too high for this node.')
   else if (y_mag_factor = 1) AND (x_mag_factor <> 1)
      then BEGIN
              writeln('You cannot set the Y mag factor to 1 if the ');
              writeln('X mag factor is greater than 1');
           END
   else    BEGIN
              gpr_$color_zoom(x_mag_factor, y_mag_factor, status);
              check('zooming');
              pause(5.0);
           END;

{Terminate.}
    gpr_$terminate(false, status);
END.
```

Chapter 7

Input Events

The graphics primitives package includes a set of routines that enable graphics programs to accept input from various input devices. Examples of GPR input events are keyboard input and mouse movement. Input routines function in all display modes except gpr_$no_display. This chapter demonstrates the calls shown below:

gpr_$cond_event_wait

gpr_$disable_input

gpr_$enable_direct_access

gpr_$enable_input

gpr_$event_wait

gpr_$get_ec

gpr_$inq_window_id

gpr_$set_input_sid

gpr_$set_window_id

7.1 What is an Event?

An event occurs when input is generated in a frame, direct mode window, or borrowed display. The Domain system supports many different kinds of events; for example, there are time events, inter process communication (ipc) events, and graphics events. This chapter describes graphiçs events; for information on other kinds of events, see the *Programming with General System Calls* manual.

7.1.1 Kinds of Events

GPR uses the enumerated type gpr_$event_t to specify the currently supported GPR event types. Currently, gpr_$event_t defines the following eight GPR event types:

gpr_$keystroke

This type of event occurs when the operator types a specified keyboard character. Programs can select a subset of keyboard characters, called a keyset, to be recognized as keystroke events. In direct and frame mode, keys that do not belong to the keyset are processed normally by the Display Manager. In borrow mode, the system ignores keys that do not belong to the keyset.

gpr_$buttons

A gpr_$buttons event occurs when the operator presses a button on the mouse or bitpad puck. Programs can select a subset of the available buttons to be recognized as a gpr_$buttons event.

gpr_$locator, gpr_$locator_update

A gpr_$locator or gpr_$locator_update event occurs when the operator moves the mouse or uses the touchpad or bitpad. The two event types are very similar. The distinction between them is that gpr_$locator_update is not as detailed as gpr_$locator and is consequently less expensive to use.

gpr_$entered_window, gpr_$left_window

In direct or frame 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. In borrow mode, there is no window or frame to move into or out of; therefore, these events cannot occur in borrow mode.

gpr_$locator_stop

A locator stop event occurs when the operator stops moving the mouse or stop using the touchpad or bitpad in the bitmap.

gpr_$no_event

A gpr_$no_event is just that, namely, it indicates that no GPR event occurred. (This event can only occur when using the gpr_$cond_event_wait routine described later in this chapter.)

7.2 Reporting Input Events

Now that we have defined the GPR events that can take place, we can explain how to make a program report a particular event. Programs can wait for an event in two different ways. The first way involves GPR calls only, and we describe this method in this section. The second way involves EC2 calls combined with the gpr_$get_ec call, and this method is described in Section 7.3.

Basically, programming for GPR input events involves the following three steps:

- Enabling one or more GPR event types.
- Waiting for one of the enabled events to occur.
- Branching to the appropriate routine depending on which event did or did not occur.

We examine the first two steps in detail.

7.2.1 Enabling Input Events

The first step in reporting input events is to enable the events that you want the system to monitor for you. You can enable none, some, or all of the GPR event types. By default, none of the GPR event types are enabled. To enable a GPR event type, your program must call the gpr_$enable_input routine. Your program must call gpr_$enable_input one time for *each* event type you wish to enable.

When you enable an event type of gpr_$keystroke, you must specify precisely which keys you want enabled. By default, none of the keys are enabled. You can enable any key on the keyboard. When defining a keyset for a keystroke event, consult the system insert files /sys/ins/kbd.ins.pas, /sys/ins/kbd.ins.ftn, or /sys/ins/kbd.ins.c. These files contain the definitions for the non-ASCII keyboard keys in the range 128 through 255.

When you enable an event type of gpr_$buttons, you must also specify precisely which buttons you want enabled. By default, none of the buttons are enabled. The definitions for buttons are stored in /sys/ins/kbd.ins.pas, /sys/ins/kbd.ins.ftn, or /sys/ins/kbd.ins.c. Note that if a mouse has three buttons, then there are really six different button choices that you may enable because the system views the depressing of a button and the releasing of a button as two different events. Table 7-1 shows that each button event can be represented as either a constant (e.g., KBD_$M2D) or a character constant (e.g., 'b').

Table 7-1. Mouse Buttons

Event	Constant	Alternate Constant
Depressing the left button	KBD_$M1D	'a'
Releasing the left button	KBD_$M1U	'A'
Depressing the middle button	KBD_$M2D	'b'
Releasing the middle button	KBD_$M2U	'B'
Depressing the right button	KBD_$M3D	'c'
Releasing the right button	KBD_$M3U	'C'
Depressing the fourth button (on a mouse with 4 buttons)	KBD_$M4D	'd'
Releasing the fourth button (on a mouse with 4 buttons)	KBD_$M4U	'D'

To disable a previously enabled event, call the gpr_$disable_input routine.

Even though enabled input events are not attributes, but they are stored in attribute blocks in much the same way as attributes. However, you cannot set and inquire about input events in the same way that you can attributes. Since enabled input events are stored in attribute blocks, if you change 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.

NOTE: gpr_$no_event does not have to be enabled by gpr_$enable_input. The system enables gpr_$no_event by default.

7.2.2 Waiting for an Event to Occur

After enabling one or more event types, the next step is to wait for one of these events to occur. GPR provides two different routines for doing this: gpr_$event_wait and gpr_$cond_event_wait.

gpr_$event_wait suspends program execution until one of the events enabled by gpr_$enable_input occurs. The gpr_$event_wait routine returns the type of event that occurred, the character (if event type is gpr_$buttons, gpr_$keystroke, or gpr_$entered_window) associated with the event, and the position at which the event occurred. In direct mode, the returned position will be relative to the upper left corner of the window. In borrow mode, the returned position will be relative to the upper left corner of the screen. In frame mode, gpr_$event_wait does not return position information.

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).

The gpr_$locator and gpr_$locator_update event types are quite similar. The differences between the two events are subtle, but important. If multiple events occur between calls to gpr_$event_wait and gpr_$locator is enabled, then gpr_$event_wait will attempt to report every event that occurred. By contrast, if multiple events occur between calls to gpr_$event_wait and gpr_$locator_update is enabled, then gpr_$event_wait will discard all locator events except the most current one. In addition, the distinction between gpr_$locator and gpr_$locator_update is important for cursor tracking and we detail this distinction in Chapter 8.

GPR allows you to enable both gpr_$locator and gpr_$locator_update; however, if you do enable both, GPR will behave as if only gpr_$locator_update was enabled.

We conclude this section with two sample programs demonstrating event enabling and event reporting.

> NOTE: If your direct mode program calls gpr_$event_wait, gpr_$cond_event_wait, or ec2_$wait, you should see Section 10.1.1 in Chapter 10 for a description of time-outs.

7.2.3 Sample Event Programs

The following program demonstrates how to enable two event types and to wait for one of them to occur:

```
Program simple_events;
{This program enables two kinds of events.  It then waits for one of those
 events to occur and reports which one occurred. }
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/kbd.ins.pas';

VAR
   event_type : gpr_$event_t;
   key_set    : gpr_$keyset_t;
   event_data : char;
   position   : gpr_$position_t;
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{***************************************************************************}
Procedure enable_events;  {This procedure enables the three mouse buttons and
                           all lowercase letters.}
BEGIN
    event_type := gpr_$buttons;
    key_set := [KBD_$M1D, KBD_$M2D, KBD_$M3D];  {Depressing any of the three}
    gpr_$enable_input(event_type, key_set, status); { mouse buttons.}

    event_type := gpr_$keystroke;
    key_set := ['a'..'z'];    {All lowercase letters.}
    gpr_$enable_input(event_type, key_set, status);
END;
{***************************************************************************}
BEGIN
    init(gpr_$borrow);
    enable_events;

{Wait for the user to type a lowercase letter or to depress a mouse button.}
    discard(gpr_$event_wait(event_type, event_data, position, status));

    gpr_$terminate(false, status);

{Report on the event that occurred.}
    if event_type = gpr_$keystroke
        then writeln('You typed the letter ',event_data)
    else
             writeln('You hit a mouse button.');
END.
```

The following program leaves a line trail by creatively using locator events:

```
Program locator_events;
{This program enables and reports locator events.  It uses the
 gpr_$enable_input and gpr_$event_wait routines.  The program allows you to
 sketch lines.  To begin, depress the left-most mouse button, then move the
 cursor.  To end, release the left-most mouse button.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/kbd.ins.pas';

VAR
   event_type : gpr_$event_t;
   key_set    : gpr_$keyset_t;
   event_data : char;
   position   : gpr_$position_t;
   sketch_started : boolean := false;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{****************************************************************************}
Procedure enable_events;  {This procedure enables the tracking of the mouse
                           and the left-most mouse button.}
BEGIN
    event_type := gpr_$locator_update;
    gpr_$enable_input(event_type, key_set, status);

    event_type := gpr_$buttons;
    key_set := [KBD_$M1D, KBD_$M1U]; {The left-most mouse button.}
    gpr_$enable_input(event_type, key_set, status);
END;
{****************************************************************************}
BEGIN
    init(gpr_$borrow)

    enable_events;

    repeat
       discard(gpr_$event_wait(event_type, event_data, position, status));
       if (event_type = gpr_$buttons) AND (event_data = KBD_$M1D)
               then begin
                       gpr_$move(position.x_coord, position.y_coord, status);
                       sketch_started := true;
                    end
       else if (event_type = gpr_$locator_update) AND (sketch_started)
               then gpr_$line(position.x_coord, position.y_coord, status);
    until (event_type = gpr_$buttons) AND (event_data = KBD_$M1U);

{Pause 5 seconds; then terminate.}
    pause(5.0);
    gpr_$terminate(false, status);
END.
```

7.3 Using EC2 Calls

The gpr_$get_ec routine returns the event count associated with a graphic input event. Programs can use this routine with gpr_$cond_event_wait to wait for a combination of system events as well as GPR input events. See *Programming with General System Calls* for full details on event counts.

7.4 Input Events When a Program Runs in Multiple Windows

It is possible to create a program that displays graphics in more than one window. This section explains how to create such a program and how to interpret input events.

7.4.1 Creating a GPR Program That Runs in Multiple Windows

To create a GPR program that runs in more than one window you must follow these steps:

1. Call the pad_$create_window one time for *each* window that you want the program to run in. For example, if you want your program to run in three window pads, than you must call pad_$create_window three times.

2. Call the gpr_$init routine one time for each window that you want GPR graphics to run in. Set the display mode to gpr_$direct in each gpr_$init call.

The pad_$create_window call is a PAD call and is described in the *Programming with General System Calls* manual.

7.4.2 Monitoring Input Events

When a GPR program monitors input events, it is important that the program be able to determine which window the event occurred in. It is not possible to do so by getting the position from a gpr_$locator or gpr_$locator_update event, because these event types return a position *relative* to the origin of the pad. However, it is possible to determine which window the event occurred in by examining the event_data returned by a gpr_$entered_window event.

The event_data returned will be a single character that represents the id of the window in which the event occurred. By default, the id character of every window is 'A', which makes discrimination between windows impossible. However, you can call the gpr_$set_window_id routine to establish a nondefault character for each window. For example, one window could be 'B', a second window could be 'C', etc. Therefore, if you enter the second window, the event_data returned will be 'C' and your program will know that it is in the second window.

There is an alternative way for determining the window in which an event occurred. In the alternative way, you still call gpr_$set_window_id to establish unique characters for each window. However, you do not necessarily have to enable gpr_$entered_window events. You can enable any kind of event. After the event occurs, you merely call gpr_$inq_window_id and it returns the identification character.

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.

The following program illustrates how to monitor input events in programs that run in multiple windows:

```
Program gpr_in_multiple_windows;
{This program creates a GPR program that runs in one shell and two windows.
 It primarily demonstrates the pad_$create_window, gpr_$event_wait, and
 gpr_$set_window_id calls (though several other GPR and PAD calls are also
 used).  In this program, all output from writeln statements will be printed
 in stdout (which is probably the shell that you invoke the program from).
 We create two windows by calling pad_$create_window twice.  We also
 call gpr_$init twice, once for each window, so that direct mode graphics can
 run in both windows.  The user will move the cursor into one of the two
 windows.  The program will draw a line into the chosen window.  We used the
 gpr_$set_window_id call to mark the two windows with a unique identification
 character.  When the user moves the cursor into one of the two windows, the
 gpr_$event_wait routine will return the identification character of the
 appropriate window.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/pad.ins.pas';

VAR
   unobs          : boolean;
   delete_display : boolean := false;
   event_type     : gpr_$event_t;
   key_set        : gpr_$keyset_t := [ ];
   event_data     : char;
   position       : gpr_$position_t;
   display_bm1, display_bm2  : gpr_$bitmap_desc_t;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{***************************************************************************}
Procedure ASSIGN_SOME_STUFF_TO_BITMAPS(graphics_strid : ios_$id_t);
BEGIN
   gpr_$enable_input(gpr_$entered_window, key_set, status);{enable one event.}
   pad_$set_auto_close(graphics_strid,1,true,status);
   gpr_$set_obscured_opt(gpr_$pop_if_obs,status);
   gpr_$set_auto_refresh(true,status);
END;
{***************************************************************************}
Procedure CREATE_TWO_WINDOWS; {This procedure creates and initializes two
                               windows for the display of graphics.}
VAR
   mode :  static gpr_$display_mode_t := gpr_$direct;
   graphics_strid1, graphics_strid2   : ios_$id_t;
   window1, window2                   : pad_$window_desc_t;
   display_bm1_size, display_bm2_size : gpr_$offset_t;
   hi_plane                           : static gpr_$rgb_plane_t := 1;
BEGIN
{Create two windows.}
   window1.top := 0;     window1.left := 0;
   window1.width := 400; window1.height := 400;
   window2.top := 500;   window2.left := 500;
   window2.width := 200; window2.height := 200;
   pad_$create_window('',0,pad_$transcript,1,window1,graphics_strid1,status);
   pad_$create_window('',0,pad_$transcript,1,window2,graphics_strid2,status);

{Initialize graphics in both windows and give each window a distinct char id.}
   display_bm1_size.x_size := 400;    display_bm1_size.y_size := 400;
   display_bm2_size.x_size := 200;    display_bm2_size.y_size := 200;
   gpr_$init(mode,graphics_strid1,display_bm1_size,hi_plane,dis-
play_bm1,status);
   gpr_$set_window_id('1', status);
   ASSIGN_SOME_STUFF_TO_BITMAPS(graphics_strid1);

   gpr_$init(mode,graphics_strid2,display_bm2_size,
             hi_plane,display_bm2,status);
   gpr_$set_window_id('2', status);
   ASSIGN_SOME_STUFF_TO_BITMAPS(graphics_strid2);
END;
{****************************************************************************}
Procedure DRAW_A_LINE_IN_WINDOW(IN window_that_cursor_is_in : char);
BEGIN
   if window_that_cursor_is_in = '1'
      then gpr_$set_bitmap(display_bm1, status)
      else gpr_$set_bitmap(display_bm2, status);

   discard(gpr_$acquire_display(status));
   gpr_$line(100, 100, status);
   gpr_$release_display(status);
END;
{****************************************************************************}
BEGIN
   CREATE_TWO_WINDOWS;
   writeln('Two empty windows just appeared.  Move the cursor into one');
   writeln('of them.  The program will draw a line in the selected window.');

   discard(gpr_$event_wait(event_type, event_data, position, status));
   DRAW_A_LINE_IN_WINDOW(event_data);

{Pause 2 seconds; then terminate.}
   pause(2);
   gpr_$terminate(false, status);
END.
```

7.5 Adjusting the Input Stream

The 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 and direct modes, input comes directly from the keyboard.

Chapter 8

The Cursor

This chapter explains how to control the cursor. In it, we describe the routines listed below:

gpr_$inq_cursor

gpr_$set_cursor_active

gpr_$set_cursor_pattern

gpr_$set_cursor_position

gpr_$set_cursor_origin

8.1 Overview of the Cursor

The cursor marks your position on the screen. Many aspects of cursors are under the GPR programmer's control. In this section, we examine how the DM controls the cursor outside of GPR applications, and how you can control the cursor within GPR applications.

Forget about GPR programming for a moment and try a little experiment. Place the cursor in some window that is not running a GPR program. The cursor is that blinking rectangle. Or is it? Move the cursor around with a locator deyice (e.g., a mouse or touchpad). Notice that the cursor changes from a blinking rectangle to a nonblinking arrow. Stop moving the locator device. Notice how the cursor changes back to the blinking rectangle. The preceding experiment illustrates the two cursors controlled by the Display Manager, namely, the **blinking cursor** and the **nonblinking cursor**.

You can control the blinking cursor in your GPR program, but you cannot (directly) control the nonblinking cursor. The DM controls the rate at which the blinking cursor blinks, and there is no way that you can adjust this rate. We explain how to use the blinking cursor in the remainder of this chapter.

8.2 Display Mode and Cursor Control

In borrow and direct modes, your GPR program has complete control over the cursor. From these two modes, you can call any of the five cursor routines listed at the top of this page.

In direct mode, the program-defined cursor pattern and origin are in effect only when the cursor is within the window that the graphics program is running in. As the user moves the cursor between the program's window and other windows on the screen, the system automatically changes the cursor pattern.

In frame mode, the cursor is controlled by the Display Manager and is always displayed. Therefore, in frame mode, the only cursor control routine you can call is gpr_$set_cursor_position. Furthermore, you can call gpr_$set_cursor_position only when the cursor lies within the frame.

8.3 How to Control The Cursor In Your GPR Program

Use the following calls to control the cursor in your GPR program:

gpr_$set_cursor_active	Specifies whether or not to display the current cursor pattern.
gpr_$set_cursor_pattern	Sets the current cursor pattern equal to the contents of the specified bitmap.
gpr_$set_cursor_position	Sets a position on the screen for display of the cursor.

By default, the cursor is not displayed in a direct mode or borrow mode GPR application. To display the cursor, call gpr_$set_cursor_active. This call displays the cursor pattern at the cursor position.

The default cursor pattern in a GPR program is a rectangle. The size of this default cursor pattern depends on the standard font the DM uses on the target node, and will therefore vary between nodes. If you don't like the default cursor pattern, you can create your own by storing a cursor pattern in a small bitmap and then calling gpr_$set_cursor_pattern to establish this bitmap as the new cursor pattern. The bitmap can be no larger than 16x16 pixels, and no more than one plane thick. Programs running in frame mode cannot call gpr_$set_cursor_pattern.

By default, the starting cursor position is (0,0). If you do not enable gpr_$locator or gpr_$locator_update event types, then the system will automatically update the cursor position as you move the locator device. (In Section 8.5, we examine the influence of gpr_$locator and gpr_$locator_update on cursor position.) At any point during the execution of the program, you can call gpr_$set_cursor_position to change the cursor position. Programs running in frame mode can call gpr_$set_cursor_position.

> NOTE: Please note the distinction between current position (described in Chapter 3) and cursor position. For example, in order to draw a figure that begins at the cursor position, you must first set the current position equal to the cursor position with the gpr_$move routine.

The following program creates a nondefault cursor pattern:

```
Program nondefault_cursor_example;
{This program creates a nondefault cursor pattern.  It demonstrates the
 gpr_$set_cursor_position, gpr_$set_cursor_pattern, and gpr_$set_cursor_active
 routines.  The program draws a pattern in a main memory bitmap, and then
 establishes this bitmap as the current cursor pattern.  It then activates
 this cursor and gives it a starting position of 200,200.  The cursor will
 automatically follow the motion of the locator device.  When you move the
 locator device, you'll see the default nonblinking cursor pattern.  When
 you stop moving the locator device, you'll see the nondefault cursor pattern
 (which blinks).
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   cursor_bitmap_descriptor : gpr_$bitmap_desc_t;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{****************************************************************************}

Procedure CREATE_CURSOR_PATTERN;
VAR
   size_of_bitmap : static gpr_$offset_t := [16,16];
   attributes_descriptor : gpr_$attribute_desc_t;

BEGIN
{Allocate a small main memory bitmap.}
   gpr_$allocate_attribute_block(attributes_descriptor, status);
   gpr_$allocate_bitmap(size_of_bitmap, hi_plane, attributes_descriptor,
                        cursor_bitmap_descriptor, status);

{Draw an arrow pattern inside the main bitmap.}
   gpr_$set_bitmap(cursor_bitmap_descriptor, status);
   gpr_$move(8, 15, status);
   gpr_$line(8, 0, status);
   gpr_$line(1, 7, status);
   gpr_$line(15,7, status);
   gpr_$line(8, 0, status);
END;
{****************************************************************************}

Procedure INITIALIZE_CURSOR;
VAR
   cursor_active   : static boolean        := true;
   cursor_position : static gpr_$position_t := [200,200];
   cursor_origin   : static gpr_$position_t := [8,0];
BEGIN
{Make the main memory bitmap into the current cursor pattern.}
   gpr_$set_cursor_pattern(cursor_bitmap_descriptor, status);

{Establish 200,200 as the starting cursor position.}
   gpr_$set_cursor_position(cursor_position, status);

{Make the cursor visible.}
   gpr_$set_cursor_active(cursor_active, status);

{Sets position 8,0 as the cursor origin.}
```

```
   gpr_$set_cursor_origin(cursor_origin, status);
END;
{****************************************************************************}

BEGIN
   init(gpr_$borrow);

   create_cursor_pattern;
   initialize_cursor;

{Pause for 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

NOTE: When the cursor is active, the cursor pattern shares display memory with whatever image is currently stored in display memory. Therefore, programs that operate in borrow or direct mode can potentially interfere with the cursor pattern and/or cause the cursor to interfere with a bitmap pattern. To avoid this problem, you should disable the cursor before performing output procedures to any area of the display in which the cursor could be located. To disable the cursor, call gpr_$set_cursor_active.

8.4 Cursor Origin

The gpr_$set_cursor_origin routine designates one of the cursor's pixels as the cursor origin. Thereafter, when you move the cursor with a locator device, the pixel designated as the cursor origin moves to the screen coordinate designated as the cursor position. Conversely, when you use the cursor to point at something, the actual location of the cursor (as returned by gpr_$event_wait or gpr_$cond_event_wait) is equal to the current cursor position offset by the cursor origin.

For example, in the cursor program, we used gpr_$set_cursor_origin to set the cursor origin to coordinates 8,0 as shown in Figure 8–1. If the cursor position is set to (200,200), it means that the pixel representing the cursor origin will be displayed at position (200,200).

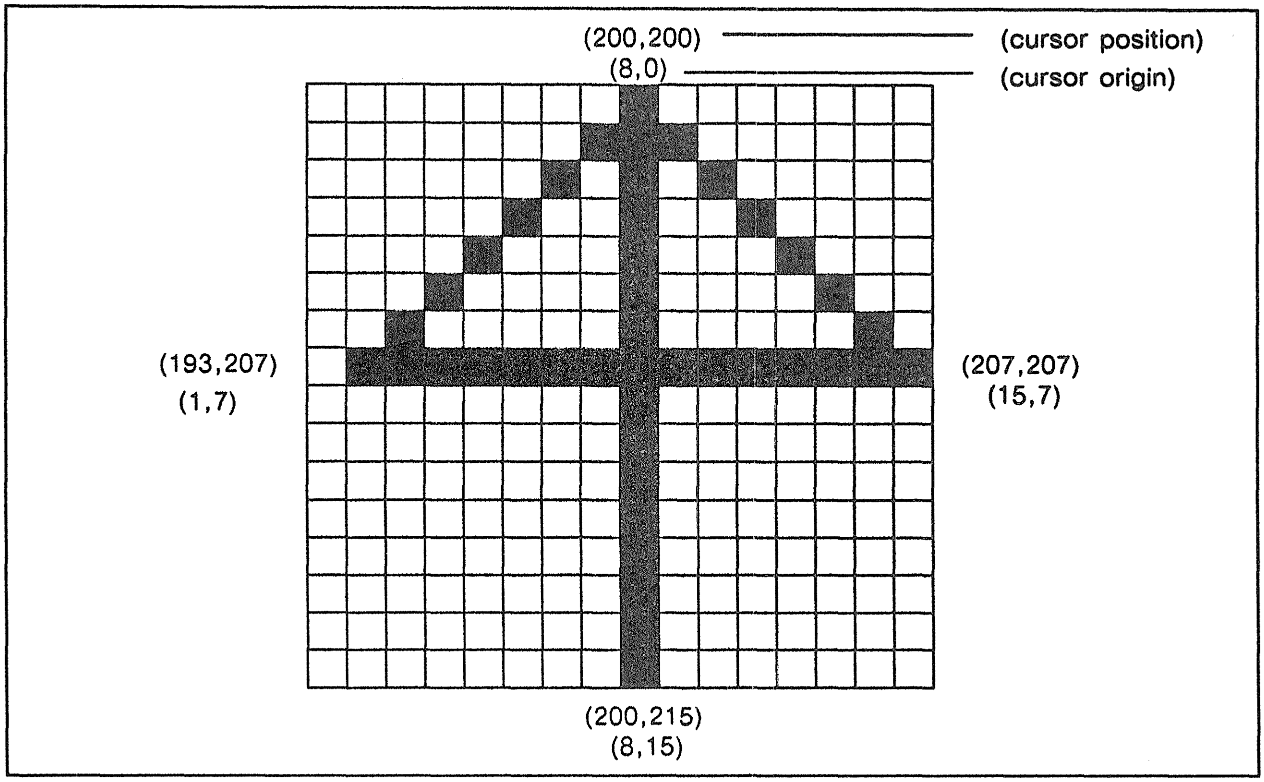


Figure 8-1. Relationship of Cursor Position and Cursor Origin

8.5 Inquiring About the Cursor

You can use the gpr_$inq_cursor routine to return the current cursor position, current cursor origin, and bitmap containing the current cursor pattern.

8.6 Tracking the Cursor

If you do not enable gpr_$locator or gpr_$locator_update, the system will automatically move the nonblinking cursor as the user moves the mouse (or some other locator device).

If you enable gpr_$locator_update, then the system will also automatically move the nonblinking cursor as the user moves the mouse. When the user stops moving the mouse, the system will display the blinking cursor.

If you enable gpr_$locator, then the system will *not* automatically move the cursor. Instead, you must control the motion of the cursor by calling gpr_$event_wait and gpr_$set_cursor_position. The gpr_$event_wait call will return a position which you can pass as input to gpr_$set_cursor_position. This method permits you to display a nondefault cursor pattern whenever you move a locator (instead of only when the locator is stopped).

```
Program tracking_the_cursor;
{This program shows how you can display a nondefault cursor wherever the
 mouse tracks to.  It uses a combination of cursor and event calls to
 demonstrate this feature.  Enter <CTRL-Q>  to exit from the program.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   cursor_bitmap_descriptor : gpr_$bitmap_desc_t;
   event_type : gpr_$event_t;
   mouse_position, cursor_position : gpr_$position_t;
   event_data : char;
   key_set : gpr_$keyset_t := [];

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
{****************************************************************************}
Procedure CREATE_CURSOR_PATTERN;
VAR
   size_of_bitmap        : static gpr_$offset_t := [16,16];
   attributes_descriptor : gpr_$attribute_desc_t;
   center                : static gpr_$position_t := [8,8];
BEGIN
   gpr_$allocate_attribute_block(attributes_descriptor, status);
   gpr_$allocate_bitmap(size_of_bitmap, hi_plane, attributes_descriptor,
                        cursor_bitmap_descriptor, status);

{Draw the cursor pattern (a small filled circle) inside the main bitmap.}
   gpr_$set_bitmap(cursor_bitmap_descriptor, status);
   gpr_$circle_filled(center, 6, status);
END;
{****************************************************************************}
Procedure INITIALIZE_CURSOR;
VAR
   cursor_origin   : static gpr_$position_t := [8,1];
BEGIN
   gpr_$set_cursor_active(false, status);
   gpr_$set_cursor_origin(cursor_origin, status);
   gpr_$set_cursor_pattern(cursor_bitmap_descriptor, status);
   gpr_$set_cursor_active(true, status);
END;
{****************************************************************************}
BEGIN
   init(gpr_$borrow);

   create_cursor_pattern;
   initialize_cursor;

{Activate the locator.}
   event_type := gpr_$locator;
   gpr_$enable_input (event_type, key_set, status);

   repeat
     discard(gpr_$event_wait(event_type, event_data, mouse_position, status));
     if event_type = gpr_$locator
         then begin
                 cursor_position := mouse_position;
                 gpr_$set_cursor_position(cursor_position, status);
                 cursor_position.x_coord := cursor_position.x_coord + 25;
                 gpr_$circle(cursor_position, 1, status);
                 check('drawing circle');
              end;
   until false;

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

Chapter 9

Clip Windows and Plane Masks

This chapter explains how to establish a clip window. A clip window is a kind of bitmap mask. We also explain how to establish a mask on one or more entire planes. This chapter demonstrates the calls shown below:

gpr_$set_clipping_active

gpr_$set_clip_window

gpr_$set_plane_mask_32

gpr_$inq_constraints

9.1 What Is a Clip Window?

A clip window is a rectangular region within a bitmap. Only the bits within that rectangle can be modified. Bits outside the clip window cannot be modified. For example, suppose you call gpr_$line to draw a line from point 1 to point 2. If both point 1 and point 2 are inside the clip window, then the entire line will be drawn. However, if one or both of the points are outside the clip window, then only a fraction (or none) of the line will be drawn. Any part of the line that falls outside the clip window will not be drawn.

Figure 9-1 illustrates the effect of a clip window.

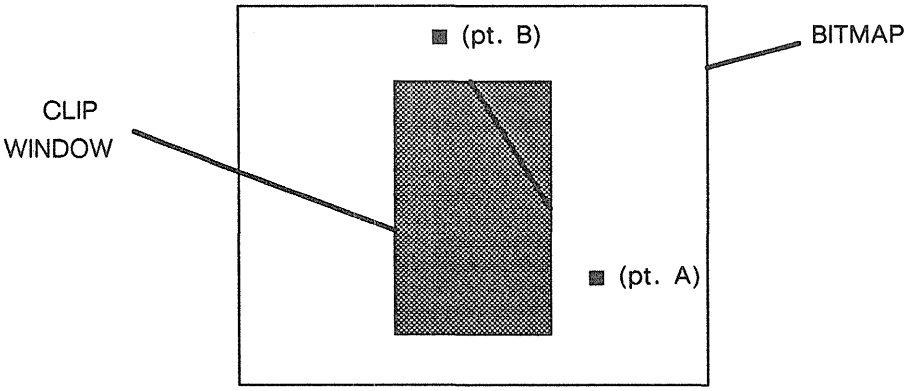


Figure 9-1. A clipping window masks out a portion of the bitmap.

9.2 Default Clip Window Size

The default size of a clip window in borrow and frame mode is the size of the bitmap. The default size of a clip window in direct mode is the size of the window. The size of the clip window is stored in the bitmap's attribute block. If the program reassigns the attribute block from one bitmap to a smaller bitmap, the system automatically reduces the clip window's size to the size of the smaller bitmap. Besides reassigning the attribute block, the only other way to change the default clip window is to call gpr_$set_clip_window.

9.3 Enabling and Disabling the Clip Window

In borrow and frame mode, clipping is disabled by default. In direct mode, it is enabled by default.

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

With clipping disabled, you are allowed access to the entire bitmap, but some GPR routines, such as gpr_$triangle, will return an error status if any of the specified coordinate values lie outside bitmap limits. Other routines, such as gpr_$line, will perform as if clipping were enabled but the clip window covered the entire bitmap. (See the descriptions of the individual routines in the *Domain Graphics Primitives Resource Calls Reference* manual.)

9.4 How to Establish a Nondefault Clip Window

To establish and activate a nondefault clip window, follow these two steps:

1. Call gpr_$set_clip_window to define the region of the clipping window within the current bitmap.

2. Call gpr_$set_clipping_active to enable the clipping window. (Though, you don't have to call this routine if clipping is already enabled.

If you ever decide to disable the clipping window, just call gpr_$set_clipping_active a second time, and set the first argument to false.

The following program demonstrates clipping. Executing this program produces the image shown in Figure 9-2. Notice that the part of the circle that falls outside of the clip window is not drawn.

```
Program clipping;

{  This program demonstrates how to create and activate a clipping window,
   and how a clipping window affects a drawing.  This program creates and
   activates a 200x200 clipping window.  Inside the clipping window,
   we draw a filled circle.  The parts of the circle outside the clipping
   window are not drawn.  If you experiment with the radius size, this
   effect will become more apparent.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   clip_window : gpr_$window_t;
   center      : gpr_$position_t := [500, 500];
   radius      : integer16;

%include 'my_include_file.pas';{contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

   clip_window.window_base.x_coord := 400;
   clip_window.window_base.y_coord := 400;
   clip_window.window_size.x_size  := 200;
   clip_window.window_size.y_size  := 200;
   gpr_$set_clip_window(clip_window, status);  { Create clipping window. }
   gpr_$set_clipping_active(true, status);     { Activate clipping window. }

   radius := 116;
   gpr_$circle_filled(center, radius, status); { Draw filled circle }

{Pause 5 seconds; then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

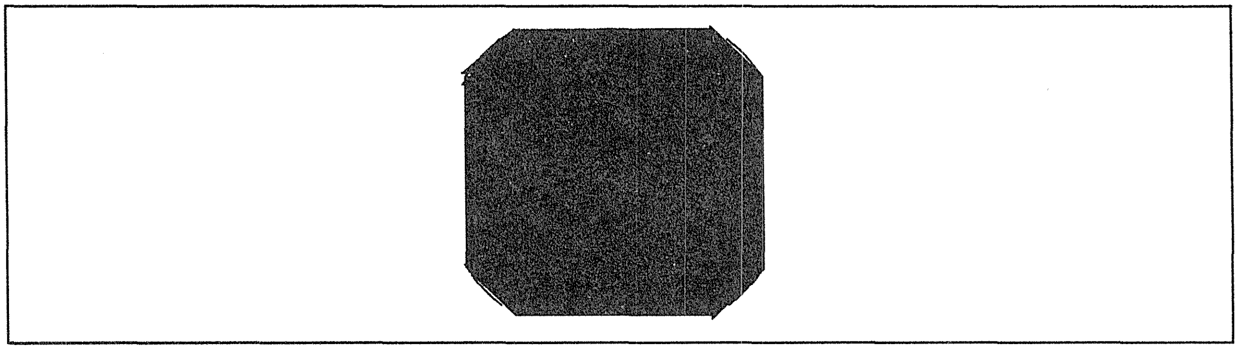


Figure 9-2. A clipped circle

9.5 Multiple Clip Windows

Information about clipping is stored in a bitmap's attribute block. The attribute block has space to store the dimensions of only one clip window. In other words, you cannot simultaneously enable multiple clip windows in the same attribute block. (Though, see Section 10.2 of Chapter 10 for an interesting application that constantly changes the clip window.) You can store a different clip window in different attribute blocks of the same bitmap.

The attribute block also records whether or not the clip window is to be enabled or disabled. You do not have to re-enable the clip window every time you call gpr_$set_clip_window.

9.6 Clip Windows in Double-Buffer Bitmaps

In general, you can use clip windows with double buffered bitmaps just as you would with other bitmaps. Note that changing the clip window size in one bitmap will change the size of the clip window in the other bitmap. When you call gpr_$select_display_buffer, the system switches only the region within the current clip window. This allows multiple regions within a bitmap to be switched independently.

9.7 Setting Plane Masks

A plane mask specifies which planes can have their bits altered. For example, consider an eight-plane node on which a mask has been placed on planes 5 and 7. Therefore, only the bits in planes 5 and 7 can be modified; bits in planes 0, 1, 2, 3, 4, and 6 cannot be modified.

The gpr_$set_plane_mask_32 routine establishes a plane mask. The following program demonstrates its use:

```
Program masking;
{ This program creates a plane mask on a color node.  It demonstrates the
  gpr_$set_plane_mask_32 routine.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   center : gpr_$position_t := [500, 500];
   radius : integer16 := 200;
   mask   : gpr_$mask_32_t := [0,1,2];

%include 'my_include_file.pas';{contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

{We set the fill value to 15, which corresponds to each bit in planes 0
 through 3 being set to a 1.}
   gpr_$set_fill_value(15, status);

{However, we set a plane mask on planes 0, 1, and 2, meaning that plane
 3 cannot be altered.}
   gpr_$set_plane_mask_32(mask, status);

{Therefore, although the fill value is 15, the effective fill value is
 actually only 7 (0111). The circle will be filled with whatever color
 corresponds to index 7.}
   gpr_$circle_filled(center, radius, status); { Draw filled circle }

{ Pause 5 seconds; then terminate. }
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

Note the difference between a plane mask and a clip window. A plane mask restricts the modification of bits that fall outside the specified plane number (i.e., a restriction in the z dimension). By contrast, clip windows restrict the modification of bits that fall outside of a particular region within any plane (i.e., a restriction in the x,y plane). You can use clip windows and plane masks together to restrict elements in three dimensions.

On a 24-plane, true-color bitmap you might consider using plane masks to mask one or more of the primary colors. For example, the green component of the color is stored in planes 8 through 15, so you could mask these planes and keep the green component constant while adjusting the red or blue component.

9.8 Inquiring About Plane Masks and Clip Windows

Use the gpr_$inq_constraints routine to return both the size of the clip window and the plane mask for the current bitmap.

Chapter 10

Direct Mode

As we mentioned in Chapter 2, direct mode gives you the performance of borrow mode without some of borrow mode's disadvantages (namely, Display Manager features are inaccessible from borrow mode). In addition, you can run only one borrow mode program at a time, but you can run several direct mode GPR programs simultaneously.

Throughout this manual we have mentioned certain differences between direct mode and borrow mode, but in this chapter we detail those topics relevant to direct mode programming only. In particular, we explain

- How to acquire and release the display.
- How to control what happens when all or part of the display window is obscured by another window.
- How to find out which part of a window is obscured by other windows.
- How to refresh a window that was obscured by another window.
- How to use high-level I/O commands (such as READ and WRITE) in a direct mode program.

The following calls are described in this chapter:

gpr_$acquire_display	gpr_$release_display
gpr_$force_release	gpr_$set_auto_refresh
gpr_$inq_refresh_entry	gpr_$set_obscured_opt
gpr_$inq_vis_list	gpr_$set_refresh_entry

10.1 Acquiring and Releasing the Display

The Domain system will only write figures and text to a window pad that has **acquired the display**. Conversely, if a window pad has not acquired the display, then your program cannot write figures or text to it.

The Domain system uses acquiring to ensure the integrity of data written to display memory. Acquiring the display is read/write lock mechanism on display memory. In order to control the competition between processes writing to different window pads, *only one process at a time may acquire the display.*

Your GPR program can acquire the display by calling the gpr_$acquire_display routine. When your window pad acquires the display, the system cannot write to other window pads. Therefore, you should try to acquire the display for the smallest amount of time possible. To **release the display**, you must call either gpr_$release_display or gpr_$force_release.

GPR maintains an acquire/release counter. The counter starts at zero when you initialize a direct mode program. Every time you call gpr_$acquire_display, the system increments the counter. Every time you call gpr_$release_display, the system decrements the counter. If the decremented counter is not equal to zero, then the display will not be released. In other words, the system will not release the display unless you have called gpr_$release_display and gpr_$acquire_display an equal number of times. For this reason, most programmers try to pair gpr_$acquire_display and gpr_$release_display calls. The following sample program demonstrates a simple acquire/release pair.

```
program acquire_release;
{This program acquires and releases the display 5000 times.  It demonstrates
 the gpr_$acquire_display and gpr_$release_display routines.  To prove that
 the display is really being released, move to another window at some point
 during this program's execution and try to type something.  If the display
 is really being released, the typed letters will appear.  This program will
 only run in an unobscured window.
}

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

CONST
   radius = 4;

VAR
   x, y, count : integer16;
   center      : gpr_$position_t;
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
    init(gpr_$direct);

    for count := 5 to 5000 do begin
{We do not need to have the display acquired to do these calculations.}
       x := (count * count) MOD display_characteristics.x_window_size;
       y := (count * x) MOD  display_characteristics.y_window_size;

       discard(gpr_$acquire_display(status)); {Acquire the display.}
         gpr_$line(x, y, status);                    {Draw a line.}
         check('Drawing line');
         center.x_coord := x; center.y_coord := y;
         gpr_$circle_filled(center, radius, status); {Draw circle.}
         check('Drawing circle');
       gpr_$release_display(status);          {Release the display.}
    end;

   gpr_$terminate(false, status);   {Terminate program.}
END.
```

Sometimes, it is inconvenient to pair a gpr_$release_display call with every gpr_$acquire_display call. For this reason, GPR also supports a gpr_$force_release call. This call automatically sets the acquire/release counter to zero and releases the display.

10.1.1 Acquire Time Out

If a direct mode program acquires the display for too long without releasing it, the system issues a fault which terminates the program. The system issues the fault because it is important that other processes not be locked out for too long if a direct mode program forgets to release the display. The **acquire time-out** period is the time that the system will wait for an acquired program to release the display. By default, the acquire time-out period is 60 seconds. However, you can call the gpr_$set_acq_time_out routine to raise or lower the acquire time-out period. You should not set the acquire time-out period below 15 seconds.

Note that exceeding the acquire time-out period is not, by itself, a sufficient condition for signaling a fault. The system will not signal the fault unless another process (which could be the Display Manager) needs to acquire the display. For example, if you move the cursor outside the window pad and exceed the acquire time-out period, then the system will signal a fault. However, if you don't enter any input events and no other process requires the display, then it is possible that an acquired program could execute for a very long time.

When your program calls gpr_$event_wait, the system implicitly releases the display and then implicitly reacquires the display when the event occurs. Therefore, a common programming technique is to set up your program as follows:

```
gpr_$acquire_display
 .
 .
 .
     BEGIN A LOOP
     .   DRAW GRAPHICS
         gpr_$event_wait
     .   DRAW GRAPHICS
     END A LOOP
 .
 .
 .
gpr_$release_display
```

If you do set up your program as shown above then it is highly improbable that the program will exceed the acquire time-out period (since the display is implicitly released every time that gpr_$event_wait is called). By contrast, the system does not implicitly release and reacquire the display when you call gpr_$cond_event_wait. Therefore, if your program used gpr_$cond_event_wait within the loop instead of gpr_$event_wait, then your program runs more of a risk of exceeding the acquire time-out period.

> NOTE: Your program should not call ec2_$wait (or any other wait call except gpr_$event_wait) when the display is acquired.

10.2 If the Window Becomes Obscured

Direct mode programs run in windows, of course, and windows on the Domain system can be pushed and popped on top of other windows. Therefore, the window from which your direct mode program is running may become partially or totally **obscured** (i.e., covered) by other windows.

By default, if your direct mode program tries to acquire the display when the window is partially or totally obscured, the system will return an error. However, you can call the gpr_$set_obscured_opt routine to force a nondefault behavior instead. The first argument to gpr_$set_obscured_opt tells the system what to do when the window to be acquired is obscured. For example, if the first argument is gpr_$pop_if_obs, then the system will pop the window on acquisition so that the window is no longer obscured.

If the first argument to gpr_$set_obscured_opt is gpr_$ok_if_obs, then the system acquires the display even though the window is obscured. This option presents several programming challenges and opportunities. The worry here is that the option does not prevent your program from overwriting other windows. The opportunity here is that a well-written program can draw figures and write text in the parts of the window that are not obscured. In the next section, we explain how to do just that.

10.2.1 Drawing in the Nonobscured Parts of a Partially Obscured Window

Suppose that you want to draw figures or write text in the portions of a window that are not obscured. In this case, it would be very helpful to know which portions of the window are not obscured. GPR provides the gpr_$inq_vis_list routine for this purpose.

The gpr_$inq_vis_list routine returns a list of rectangles comprising the total nonobscured portion of the window. Figure 10-1 illustrates how gpr_$inq_vis_list might view the nonobscured portions of a particular window which is obscured by two other windows.

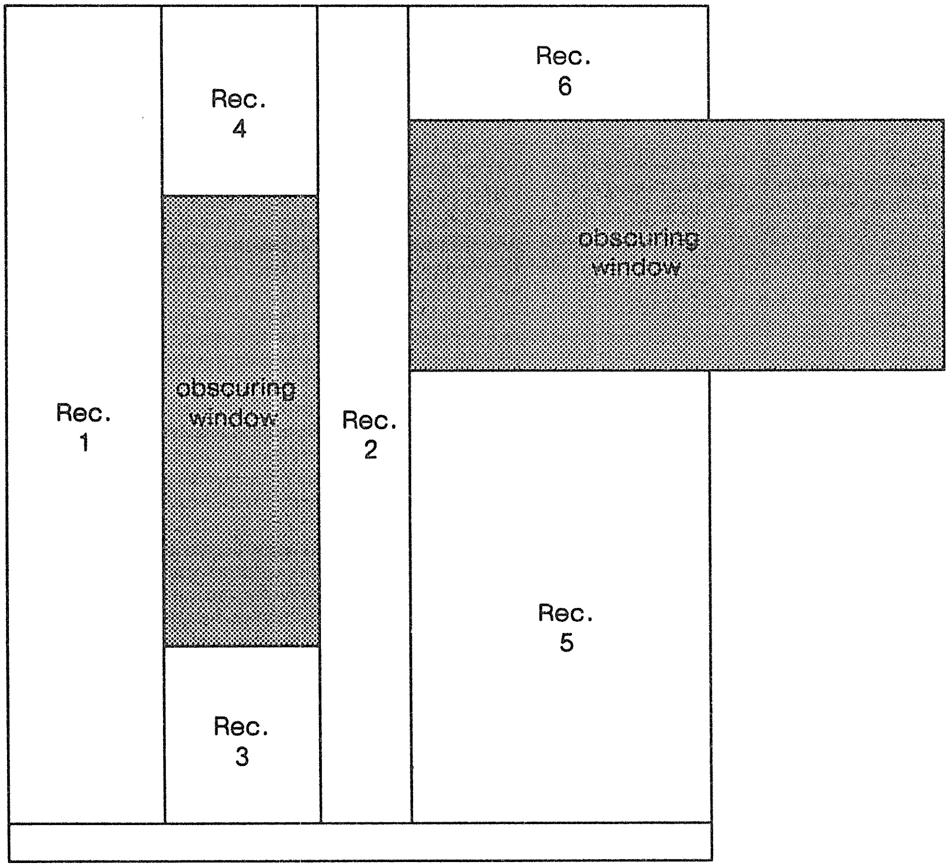


Figure 10-1. The gpr_$inq_vis_list Returns Six Rectangles Composing the Nonobscured Portions of This Window.

The rectangles returned are in gpr_$window_t format which, by no coincidence, is the same format that gpr_$set_clip_window requires for input arguments. For example, given the situation shown in Figure 10-1, you would call gpr_$set_clip_window six times. After every time you call gpr_$set_clip_window, you would do a draw or write operation. The following example demonstrates this technique:

```
Program writing_in_obscured_windows;
{This program draws lines in the nonobscured parts of the window that the
 program runs in.  It demonstrates the gpr_$set_obscured_opt and
 gpr_inq_vis_list routines.  After starting this routine, you should obscure
 portions of the window by growing or popping other windows on top of the
 window running the program.
}
%include ´/sys/ins/base.ins.pas´;
%include ´/sys/ins/gpr.ins.pas´;

CONST
   max_slots = 32;   {Set an arbitrary limit of 32 rectangles.}
VAR
   old_x, old_y, new_x, new_y, count, n : integer16;
   what_to_do_if_obscured : gpr_$obscured_opt_t := gpr_$ok_if_obs;
   slots_available : integer16 := max_slots;
   slots_total     : integer16;
   vis_list        : array[1..max_slots] of gpr_$window_t;

%include ´my_include_file.pas´;{Contains the init, check, and pause routines.}

BEGIN
    init(gpr_$direct);
    gpr_$set_clipping_active(true, status);

    gpr_$set_obscured_opt(what_to_do_if_obscured, status);

    for count := 1 to 3000 do begin  {This loop draws up to 3000 lines.}
       new_x := (count * count) MOD 600;
       new_y := (count * new_x) MOD 610;

       discard(gpr_$acquire_display(status));

{Find the visible slots every time display is acquired.}
         gpr_$inq_vis_list(slots_available, slots_total, vis_list, status);


{This loop draws a line from the old coordinates to the new coordinates
 slots_total times.  There is no harm (other than lost time) in drawing
 a line over the same coordinates.}
         for n := 1 to slots_total do begin
            gpr_$set_clip_window(vis_list[n], status);
            gpr_$move(old_x, old_y, status);
            gpr_$line(new_x, new_y, status);
         end;

         old_x := new_x;  old_y := new_y;
       gpr_$release_display(status);
    end;

{Terminate program.}
   gpr_$terminate(false, status);
END.
```

The preceding program works when windows are pushed on top of the running program. However, if you pop the window containing the running program, the system will seem to erase everything that had been previously written into the bitmap. The way to prevent this erasure is with a refresh operation.

10.3 Refreshing a Window

This section explains how to **refresh** a window. A refresh operation controls the way that a program redraws a figure when the user pops a figure that was previously obscured (partially or totally) by another window. There are two ways to accomplish a window refresh operation.

First, the program can have the Display Manager automatically refresh a popped window. To accomplish this, call the gpr_$set_auto_refresh routine. This routine will signal the Display Manager to automatically redraw the contents of the window whenever the window grows or is popped. The Display Manager only redraws what was in the window before it was obscured or had grown. For example, if only a portion of a drawing is displayed in a window because the window was too small, only that portion of the drawing will be redrawn if the window has grown. For this reason, gpr_$set_auto_refresh is most useful to handle redrawing when windows get popped. In addition, gpr_$set_auto_refresh should only be used for static data.

> NOTE: Your program should not call the gpr_$set_auto_refresh routine if you specified the gpr_$ok_if_obs option to gpr_$set_obscured_opt. Conversely, it is good programming practice to specify the gpr_$block_if_obs option to gpr_$set_obscured_opt if you intend to enable auto refresh.

The second way to refresh a window is to write your own refresh procedure that the system will automatically call whenever a refresh operation is necessary. This technique allows your application program to call the actual procedures that created the drawing; therefore, your entire drawing is redrawn. Writing your own refresh procedure is a good technique for displaying dynamic data. To perform this second technique your program must call the gpr_$set_refresh_entry routine. The first argument to this routine is a pointer to a procedure (i.e., a function returning void in C or a subroutine in FORTRAN). If the window needs to be refreshed, the system will automatically call this procedure. The second argument to gpr_$set_refresh_entry is also a pointer to a procedure; this procedure refreshes hidden display memory.

> NOTE: In Pascal and FORTRAN, the refresh procedure must be stored in a separately–compiled object file. In other words, the refresh procedures *cannot* be stored in the same source file that contains the call to gpr_$set_refresh_entry. This restriction has to do with the way that the compilers implement pointers to routines. In C, the refresh procedure *can* be stored in the same file as the file that contains the call to gpr_$set_refresh_entry.

The following two pieces of source code must be stored in different files, compiled separately, and bound together. The two pieces demonstrate how to write your own refresh procedure:

```
PROGRAM refresh_example;
{This program contains a refresh procedure.  It demonstrates the
 gpr_$set_refresh_entry routine.  The program draws a simple design.  You
 should obscure a portion of the window the program is running in and then
 pop the window so that the program will require a refresh.  The refresh
 will redraw the simple design.  To leave the program, type the letter Q.}
%include ´/sys/ins/base.ins.pas´;
%include ´/sys/ins/gpr.ins.pas´;
VAR
   unobscured : boolean;
   ev_pos   : gpr_$position_t;
   ev_type  : gpr_$event_t;
   ev_char  : char;
   keys     : gpr_$keyset_t :=  [´q´,´Q´];

PROCEDURE draw(IN unobscured : boolean;
               IN position_change : boolean); EXTERN;
%include ´my_include_file.pas´;{Contains the init, check, and pause routines.}
BEGIN
    init(gpr_$direct);

{Draw a picture.}
    draw(true, false);

{Establish the refresh procedure.  If window needs to be refreshed as
 the result of a pop, the draw procedure will automatically be called.
}
    gpr_$set_refresh_entry (addr(draw), nil, status);
    check(´setting the refresh entry procedure.´);

{Enter the letter Q to exit the program.}
    gpr_$enable_input (gpr_$keystroke, keys, status);
    discard (gpr_$event_wait (ev_type, ev_char, ev_pos, status));

    gpr_$terminate(false, status);{Terminate.}
END.


MODULE draw;
{This module must be bound with caller_of_refresh_procedure.  It contains the
 procedure that the system automatically calls when a refresh is required.}
%include ´/sys/ins/base.ins.pas´;       {required insert file}
%include ´/sys/ins/gpr.ins.pas´;        {required insert file}
%include ´my_include_file.pas´;

PROCEDURE draw;
BEGIN
{Draws a box bisected by a line.}
    discard(gpr_$acquire_display(status));
    gpr_$draw_box (200, 200, 600, 600, status);
    gpr_$move(200,200, status);
    gpr_$line(600, 600, status);
    gpr_$release_display(status);
END;
```

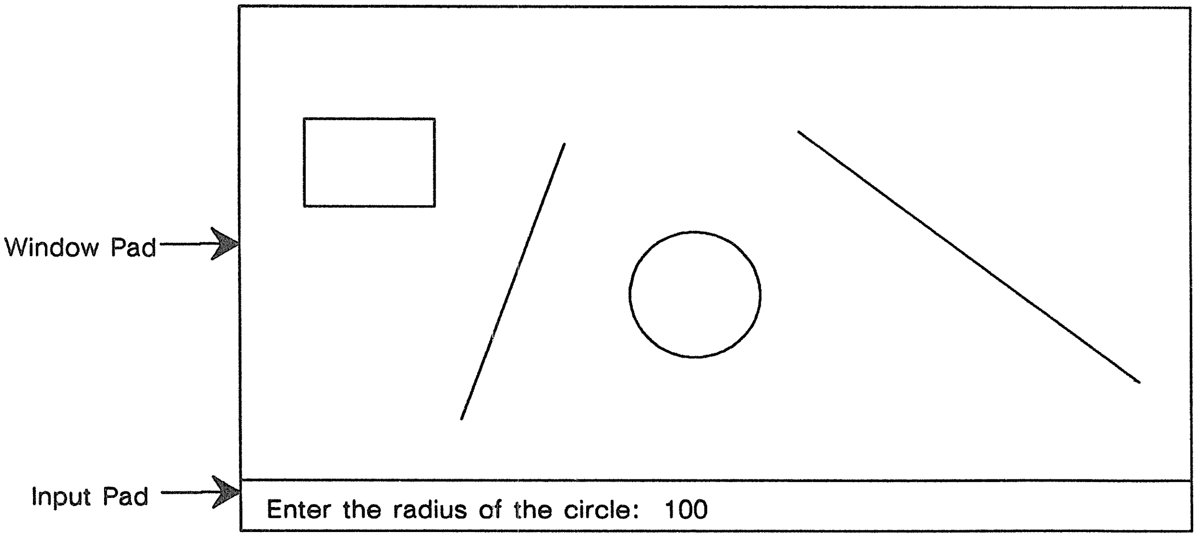
10.3.1 Refreshing a Window in a Double-Buffer Application

In general, refreshing a double-buffer application works the same way that refreshing any bitmap works. Note, however, that you should always re-establish the buffers before refreshing.

For example, suppose you are running a double-buffer program in a window. If you grow the window, the system will automatically make the primary bitmap visible (even if it was invisible when you grew the window). Therefore, the refresh procedure should always specify the buffer bitmap as the current bitmap. With the buffer bitmap current, redraw the image, and then toggle the bitmaps (with gpr_$select_display_buffer) to complete the refresh procedure.

10.4 How to Use High-Level I/O Commands

In a direct mode program, graphics are displayed in a window pad which usually appears above an input pad as shown in Figure 10-2. In Chapter 7, we described how input events in the window pad could be monitored with the gpr_$event_wait and gpr_$cond_event_wait routines. In this section, we describe how you can also enter input in the input pad.



```
Enter the radius of the circle:  100
```

Figure 10-2. The Input Pad and the Window Pad in a Direct Mode Program

You can generate prompts and read input from the input pad by using high-level I/O commands in your program such as READ and WRITE (scanf and printf). If you do use high-level I/O commands, you should note the following:

- The display cannot be acquired when you issue the high-level I/O command.
- The input pad is only one line long. Therefore, only the most recent line can be displayed. However, when you terminate the program, the transcript pad will contain a record of all prompts and user input.
- If you write a string with a high-level output command, the string will not appear in the input pad unless it is followed by a high-level input command.

The following simple program demonstrates the use of high-level I/O commands in direct mode programs:

```
PROGRAM high_level_input
{This program prompts the user for the coordinates of a line, and then draws
 a line based on the user's input.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/pad.ins.pas';

VAR
   x_start, y_start, x_stop, y_stop : integer16;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$direct);
   gpr_$set_cursor_active(true,status);

   write('Enter x coord of line start: '); readln(x_start);
   write('Enter y coord of line start: '); readln(y_start);
   write('Enter x coord of line stop: ');  readln(x_stop);
   write('Enter y coord of line stop: ');  readln(y_stop);

   discard(gpr_$acquire_display(status));
   gpr_$move(x_start, y_start, status);
   gpr_$line(x_stop, y_stop, status);
   gpr_$release_display(status);

 {Pause 5 seconds, then terminate.}
   pause(5.0);
   gpr_$terminate(false,status);
 END.
```

Chapter 11

Raster Operations

This chapter explains raster operations. Raster operations affect BLT routines, fill routines, and draw routines by specifying how source bits interact with destination bits. Graphics programmers rely on raster operations for a task such as specifying the overlap color when two images intersect. This chapter demonstrates the calls shown below:

gpr_$inq_raster_op_prim_set	gpr_$rop_prim_set
gpr_$inq_raster_ops	gpr_$set_raster_op

11.1 What is a Raster Operation?

A raster operation specifies how pixel values are determined in each plane of a destination bitmap for BLT, drawing, and fill 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. Instead, the raster operation specifies how pixel values are determined when BLTs, drawing, and fill routines are performed. Table 11-1 describes the 16 raster operations in words, and Table 11-2 lists the truth tables for all the raster operations.

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 fill routines, there is no source bitmap. However, you might imagine the source bitmap as being the image that is about to be drawn. You can assume that the value of each bit in the source bitmap depends on the current draw value or fill value.

Text routines are not affected by the current raster operation.

In summary, raster operations specify the way that the image currently displayed combines with the image that is about to overwrite it.

Table 11-1. 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 11-2. 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
0	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

11.2 Setting the Raster Operation

The default raster operation for BLT, fill, and draw operations is 3. This raster operation writes the source bits to the destination bits. In other words, the system will ignore whatever was stored in the destination bits. (That is, the old image gets overwritten by the new value.)

To specify a nondefault raster operation, you call the gpr_$set_raster_op routine. Note that this routine only changes the raster operation in one plane. Therefore, if your target node has n planes, you *may* want to call gpr_$set_raster_op n times. Note also that the raster operation is stored in the bitmap's attribute block. So if your program uses several different attribute blocks, you should make sure that the correct attribute block is current before calling gpr_$set_raster_op.

We provide two examples to demonstrate raster operations. Both programs work in four-plane or eight-plane mode only. The first program demonstrates the effect of raster operations on draw routines, in particular, gpr_$line. Here it is:

```
Program intersecting_lines;
{This program draws two intersecting lines to demonstrate raster operations.
 It uses the gpr_$set_raster_op and gpr_$inq_raster_ops routines.
 Assuming that the default color map is loaded, the program will draw one
 red line and one green line.  Because we set the raster operation to 7,
 the intersection of the two lines will be drawn in blue.  This program
 should be run on a color node.}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
VAR
   raster_op       : gpr_$raster_op_t;
   raster_op_array : gpr_$raster_op_array_t;
   plane           : gpr_$rgb_plane_t;
%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
   init(gpr_$borrow);

   gpr_$set_draw_width(20, status);
   gpr_$set_draw_value(1, status);   {Draws a red line.}
   gpr_$move(0, 200, status);
   gpr_$line(400, 200, status);

   raster_op := 6;
   for plane := 0 to hi_plane do
      gpr_$set_raster_op(plane, raster_op, status);

   gpr_$set_draw_value(2, status);
   gpr_$move(200, 0, status);
   gpr_$line(200, 400, status);   {Draws a blue line. }

   gpr_$inq_raster_ops(raster_op_array, status);
   for plane := 0 to hi_plane do
     writeln('raster op for plane ', plane:-1,' = ',
             raster_op_array[plane]:-1);

 {Pause 5 seconds; then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
 END.
```

In the preceding example, we set the raster operation in every plane to 6. Raster operation 6 performs an *exclusive or*. Given the draw value of 1 for the first line and the draw value of 2 for the second line, Table 11-3 shows how the intersecting points in the two lines will combine to form a draw value of 3.

Table 11-3. Total Plane Values in Program intersecting_lines

	Plane 3	Plane 2	Plane 1	Plane 0	
Bit values in destination planes	0	0	0	1	= 1
Bit values in source planes	0	0	1	0	= 2
Resulting bit values	0	0	1	1	= 3

The following example demonstrates the effect of raster operations on BLTs:

```
Program raster_ops_in_blts;
{This program demonstrates how you can use raster operations with BLTs.
 It uses the gpr_$set_raster_op routine.  Assuming that the default color
 map is loaded, the program will draw one blue rectangle and one yellow
 rectangle.  Then it will BLT the blue rectangle to the area covered by
 the yellow rectangle.  Because the raster operation is 1 for each plane,
 the BLT will produce a red rectangle.  This program should be run on a
 color node.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

CONST
   left_of_source = 0;            left_of_dest = 400;
   top_of_source = 0;             top_of_dest =  400;
   size_x_of_source = 200;        size_x_of_dest = 200;
   size_y_of_source = 200;        size_y_of_dest = 200;

VAR
   raster_op        : gpr_$raster_op_t;
   plane            : gpr_$rgb_plane_t;
   source_window    : gpr_$window_t;
   destination_origin : gpr_$position_t := [left_of_dest, top_of_dest];
   rectangle1       : gpr_$window_t := [[left_of_source, top_of_source],
                                         [size_x_of_source, size_y_of_source]];
   rectangle2       : gpr_$window_t := [[left_of_dest, top_of_dest],
                                         [size_x_of_dest, size_y_of_dest]];

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

   gpr_$set_fill_value(3, status);
   gpr_$rectangle(rectangle1, status);  {Draw a blue rectangle.}
   gpr_$set_fill_value(5, status);
   gpr_$rectangle(rectangle2, status);  {Draw a yellow rectangle.}
   pause(2.0);

   source_window := rectangle1;

   raster_op := 1;  {Destinaton := Source AND Destination}
   for plane := 0 to hi_plane do
      gpr_$set_raster_op(plane, raster_op, status);

{BLT the blue rectangle to the area covered by the yellow rectangle.}
   gpr_$pixel_blt(display_bitmap, source_window, destination_origin, status);

{Pause 5 seconds; then terminate.}
   pause(5.0);
   gpr_$terminate(false, status);
END.
```

In the preceding example, we set the raster operation in every plane to 2. Raster operation 2 performs an *and*. Given the fill value of 3 for the destination bitmap and the fill value of 5 for the source bitmap, Table 11-4 shows how the intersecting points in the two lines will combine to form a fill value of 1.

Table 11-4. Total Plane Values in Program raster_ops_in_blts

	Plane 3	Plane 2	Plane 1	Plane 0	
Bit values in destination planes	0	0	1	1	= 3
Bit values in source planes	0	1	0	1	= 5
Resulting bit values	0	0	0	1	= 1

11.3 Restricting Raster Ops to Particular GPR Categories

GPR permits you to specify different raster operations for different kinds of GPR routines. For example, you can set the raster operation to be 5 for fills but 12 for BLTs. The call that makes this possible is gpr_$raster_op_prim_set. The first argument to this call is the category of GPR routine that the next raster operation will apply to chosen from among the following:

- Draw routines
- Fill routines
- BLT routines

The following program demonstrates how to set different raster operations for the three different categories:

```
Program raster_op_categories;
{This program demonstrates how to set different raster operations for
 three different categories of GPR routines.  It demonstrates the
 gpr_$raster_op_prim_set and gpr_$set_raster_op routines.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';

VAR
   raster_op            : gpr_$raster_op_t;
   plane_to_set_rop_on  : gpr_$rgb_plane_t := 0;
   prim_set             : gpr_$rop_prim_set_t;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}

BEGIN
   init(gpr_$borrow);

 {Set the raster operation to 5 for all line drawing routines. }
   prim_set := [gpr_$rop_line];  raster_op := 5;
   gpr_$raster_op_prim_set(prim_set, status);
   gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status);

 {Set the raster operation to 7 for all fill routines. }
   prim_set := [gpr_$rop_fill];  raster_op := 7;
   gpr_$raster_op_prim_set(prim_set, status);
   gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status);

 {Set the raster operation to 13 for all blt routines. }
   prim_set := [gpr_$rop_blt];   raster_op := 13;
   gpr_$raster_op_prim_set(prim_set, status);
   gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status);

   gpr_$terminate(false, status);
 END.
```

11.4 Inquiring About Raster Operations

GPR supports the following two routines to inquire about the current raster operations:

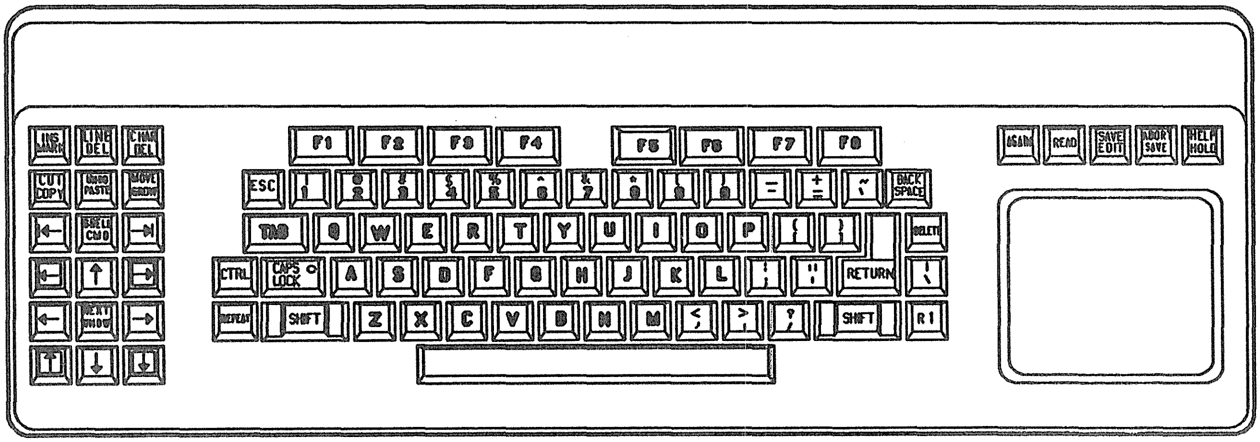
gpr_$inq_raster_op_prim_set Returns the primitive(s) which will be affected by the next gpr_$set_raster_op call, or the primitive(s) for which gpr_$inq_raster_op will return the current raster operation.

gpr_$inq_raster_ops Returns the raster operation for the category of primitives most recently specified with gpr_$raster_op_prim_set.

Appendix A

Keyboard Charts

The following chart and figure give the 8-bit ASCII values generated for the DOMAIN low-profile keyboard. This chart includes 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*.



LOW-PROFILE KEYBOARD

Figure A-1. Low-Profile Keyboard

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

	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

Appendix B

C Programs

This appendix contains all the programming examples used in the manual translated into C.

NOTE: See the *Programming With General System Calls* manual for details on simulating Pascal data types in C.

The Include File Used By All Sample C Programs

```
#include "//cascade7/sys/ins/error.ins.c"
#include "//cascade7/sys/ins/time.ins.c"

status_$t            status;
gpr_$bitmap_desc_t   display_bitmap;
gpr_$offset_t        display_bitmap_size;
gpr_$rgb_plane_t     hi_plane;
gpr_$disp_char_t     display_characteristics;


void check(messagex)
char *messagex;
{
   if (status.all)
   {   error_$print (status);
       printf("Error occurred while %s.\n", messagex);
   }
}


void pause(t)
float t;
{
time_$clock_t  time;

   time.high16 = 0;
   time.low32  = 250000 * t;

   time_$wait (time_$relative, time, status);
   check("pausing");
}


void init(mode)
gpr_$display_mode_t   mode;
{
static short int         unit = 1;
static short int         disp_len = sizeof(gpr_$disp_char_t);
       short int         disp_len_returned;
       short int         unobscured;

   gpr_$inq_disp_characteristics(mode, unit, disp_len,
                              display_characteristics,
                              disp_len_returned, status);
   check("in init after inquiring");

   display_bitmap_size.x_size = display_characteristics.x_window_size;
   display_bitmap_size.y_size = display_characteristics.y_window_size;
   hi_plane                   = display_characteristics.n_planes - 1;

   gpr_$init(mode, unit, display_bitmap_size,
             hi_plane, display_bitmap, status);
   check("in init after initializing");
}
```

A Simple GPR Program To Get Started (From Chapter 2)

```
/* Name of Program -- getting_started_with_gpr */
/* This program draws a line in borrow mode from the upper left corner of
   the screen to the lower right corner of the screen.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, & pause functions.*/

main()
{
   init(gpr_$borrow);

   gpr_$line(display_bitmap_size.x_size, display_bitmap_size.y_size, status);
   check("drawing line");

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program To Draw a Simple Line (From Chapter 3)

```
/*Name of Program -- simple_lines */
/*This program demonstrates how to use the gpr_$line call,
 and how to change the current position with the gpr_$move call. */

#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c";/*Contains the init, check, and pause routines.*/

main()
{
   init(gpr_$borrow);

/*Draw a line from the coordinate origin (0,0) to the endpoint (300,300) */
   gpr_$line(300, 300, status);

/*The coordinate origin is now set at (300,300).  Use the gpr_$move call to
  change the coordinate origin to (100,500). */
   gpr_$move(100, 500, status);

/*Draw a line from the coordinate origin (100,500) to the endpoint
(500,500).*/
   gpr_$line(500, 500, status);

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Draw Connected Lines (From Chapter 3)

```
/* Name of Program -- connected_lines */
/*This program draws several connected lines.  It demonstrates the
 gpr_$polyline call.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/


main()
{
static gpr_$coordinate_array_t   array_of_x_coordinates =  {200, 300, 400};
static gpr_$coordinate_array_t   array_of_y_coordinates =  {300, 400, 200};
static short int                 number_of_end_points   =  3;

   init(gpr_$borrow);

/*Establish the current position at (0,300).  If we do not call gpr_$move, the
  current position will be (0,0).*/
   gpr_$move(0, 300, status);

/*Draw three connected lines.  Notice that each endpoint becomes the
startpoint
 for the next line.*/
   gpr_$polyline(array_of_x_coordinates, array_of_y_coordinates,
                 number_of_end_points, status);

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Draw Disconnected Lines (From Chapter 3)

```
/* Name of Program -- disconnected_lines */
/*This program draws three disconnected lines.  It demonstrates the
 gpr_$multiline call.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

 gpr_$coordinate_array_t  array_of_x_coordinates = {100,400,100,400,100,400};
 gpr_$coordinate_array_t  array_of_y_coordinates = {100,100,200,200,300,300};
 short int  number_of_points =  6;

#include "my_include_file.c"/*Contains the init, check, and pause routines.*/

main()
{
   init(gpr_$borrow);

/*Draw three disconnected lines.  Line1 runs from (100,100) to (400, 100);
  Line2 runs from (100,200) to (400,200); Line3 runs from (100,300) to
  (400,300).  Notice that we don't have to use gpr_$move to establish the
  current position.*/
   gpr_$multiline(array_of_x_coordinates, array_of_y_coordinates,
                 number_of_points, status);

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Draw Two Circles (From Chapter 3)

```
/*Name of Program -- circles_example */
/*This program draws two circles.  It demonstrates the gpr_$circle and
  gpr_$circle_filled calls.
*/

#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

  gpr_$position_t center;
  short int       radius;
main()
{
   init(gpr_$borrow);

/*Draw an unfilled circle.*/
   center.x_coord =  200;
   center.y_coord =  200;
   radius         =  100;
   gpr_$circle(center, radius, status);

/*Draw a filled circle.*/
   center.x_coord =  400;
   center.y_coord =  400;
   radius         =  100;
   gpr_$circle_filled(center, radius, status);

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Draw Two Arcs (From Chapter 3)

```
/* Name of Program -- arcs_example */
/*This program draws two arcs.  It demonstrates the gpr_$arc_c2p and
 gpr_$arc_3p routines.*/

#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

 gpr_$position_t       center;
 gpr_$position_t       p2;
 gpr_$arc_direction_t  direction;
 gpr_$arc_option_t     option;
 gpr_$position_t       point2, point3;

main()
{
   init(gpr_$borrow);

                /****Demonstration of gpr_$arc_3p ****/
/*The gpr_$arc_3p call draws an arc through any three noncolinear points.
 The three points are the current position, point2, and point3.
 The system draws the arc from the current position through
 point2 and completes the arc at point3.
*/
   gpr_$move(200, 200, status);                  /*set the current position.*/
   point2.x_coord = 300;  point2.y_coord = 300;  /*set point2 of the arc.*/
   point3.x_coord = 200;  point3.y_coord = 400;  /*set point3 of the arc.*/
   gpr_$arc_3p(point2, point3, status);          /*draw the arc.*/


                /****Demonstration of gpr_$arc_c2p ****/
/*The gpr_$arc_c2p call draws an arc between two points.
 The radius of the arc is the distance from the current position to center.
 The system starts the arc at the current position and revolves it in a
 clockwise or counterclockwise direction.  The end point of the arc lies
 on an imaginary ray beginning at center and passing through position p2.
 The 'option' parameter is only meaningful if the current position
 is both the start and end point of the arc. In this case, 'option'
 tells the routine whether to draw a full circle or to draw nothing at all.
*/
   gpr_$move(600, 400, status);                /*set the current position.*/
   center.x_coord = 600; center.y_coord = 600; /*set "center"*/
   p2.x_coord     = 500; p2.y_coord     = 600; /*set "p2"*/
   direction =  gpr_$arc_ccw;                /*draw the arc counterclockwise*/
   option    =  gpr_$arc_draw_full;            /*ignored in this case.*/
   gpr_$arc_c2p(center, p2, direction, option, status);
   check("drawing arc_c2p");

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Draw a Spline (From Chapter 3)

```
/* Name of Program -- spline_example */
/*This program draws a cubic spline as a function of x.
 It demonstrates the gpr_$spline_cubic_x routine.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
  gpr_$coordinate_t  array_of_x_pts[5] =  {100,200,300,400,500};
  gpr_$coordinate_t  array_of_y_pts[5] =  { 20, 80,180,320,500};
  short int             npoints        =  5;

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

main()
{
   init(gpr_$borrow);

/*Draw spline as a function of x.  Starting position of spline will be
 the current position of [0,0] */
   gpr_$spline_cubic_x(array_of_x_pts, array_of_y_pts, npoints, status);
   check("drawing spline");

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Draw a Filled Triangle, Rectangle, and Trapezoid (From Chapter 3)

```
/* Name of Program -- triangle_rectangle_trapezoids */
/*This program draws a triangle, rectangle, and trapezoid.  It demonstrates
 the gpr_$triangle, gpr_$rectangle, and gpr_$trapezoid calls.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

  gpr_$position_t triangle_vertex1, triangle_vertex2, triangle_vertex3;
  gpr_$window_t   rectangle;
  gpr_$trap_t     trapezoid;

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow);

/*Draw a filled triangle.*/
   triangle_vertex1.x_coord =  100;  triangle_vertex1.y_coord =  100;
   triangle_vertex2.x_coord =  300;  triangle_vertex2.y_coord =  100;
   triangle_vertex3.x_coord =  200;  triangle_vertex3.y_coord =  200;
   gpr_$triangle(triangle_vertex1, triangle_vertex2, triangle_vertex3,status);

/*Draw a filled rectangle.*/
   rectangle.window_base.x_coord = 100; rectangle.window_base.y_coord = 300;
   rectangle.window_size.x_size  = 200; rectangle.window_size.y_size  = 300;
   gpr_$rectangle(rectangle, status);

/*Draw a filled trapezoid.  In GPR, a trapezoid is a four-sided polygon
  with parallel bottom and top sides.*/
   trapezoid.top.x_coord_l = 300; /*x coordinate of the top left point.*/
   trapezoid.top.x_coord_r = 500; /*x coordinate of the top right point.*/
   trapezoid.top.y_coord   = 200; /*y coordinate of the top line segment.*/
   trapezoid.bot.x_coord_l = 400; /*x coordinate of the bottom left point.*/
   trapezoid.bot.x_coord_r = 650; /*x coordinate of the bottom right point.*/
   trapezoid.bot.y_coord   = 500; /*y coordinate of the bottom line segment.*/
   gpr_$trapezoid(trapezoid, status);

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Draw a Filled Polygon (From Chapter 3)

```
/* Name of Program -- polygons */
/*The program draws a five-sided polygon.  It demonstrates the
gpr_$start_pgon,
 gpr_$pgon_polyline, and gpr_$close_fill_pgon routines.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

 gpr_$coordinate_array_t    array_of_x_coords =  {100,600,600,350};
 gpr_$coordinate_array_t    array_of_y_coords =  {600,600,300,100};
 short int                  number_of_points_in_array =  4;

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow);

/*Set the starting point of the polygon.*/
   gpr_$start_pgon(100, 300, status);

/*Set the other four points of the polygon.*/
   gpr_$pgon_polyline(array_of_x_coords, array_of_y_coords,
                      number_of_points_in_array, status);
   check("calling pgon_polyline");

/*Connect the five points and fill it with the current fill color.*/
   gpr_$close_fill_pgon(status);
   check("calling close_fill_pgon");

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate((short)0, status);
}
```

A Program To Create and Use a Nondefault Tile Pattern (From Chapter 3)

```
/* Name of Program -- tile_pattern */
/*This program creates a nondefault tile pattern.  It demonstrates the
 gpr_$set_fill_value, gpr_$set_fill_background_value, and
 gpr_$set_fill_pattern routines.  You must run this program on a color node.*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

#define  fill_value       3
#define  background_value 5

 gpr_$position_t        center = {300, 300};
 short int              radius =  300, x, y, scale = 1;
 gpr_$offset_t          size_of_tile_pattern =  {32,32};
 gpr_$attribute_desc_t  attribute_block_descriptor;
 gpr_$bitmap_desc_t     bitmap_desc_of_tile_pattern;
 gpr_$rgb_plane_t       hi_plane_of_mmb =  0  ;
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow);

/*Generate a 32x32 main memory bitmap to hold the tile pattern.  It is
 essential that the hi_plane value be 0.  If it is not 0, then the calls
 to set the fill value and fill background value will have no effect.*/
   gpr_$allocate_attribute_block(attribute_block_descriptor, status);
   gpr_$allocate_bitmap(size_of_tile_pattern, hi_plane_of_mmb,
                        attribute_block_descriptor,
                        bitmap_desc_of_tile_pattern, status);
   gpr_$set_bitmap(bitmap_desc_of_tile_pattern, status);

/*Set every 16th bit in the bitmap.*/
   for (x = 0; x <= 31; x++)
       {for (y = 0; y <= 31; y++)
          {if ((x % 4 == 0) && (y % 4 == 0))
             {gpr_$move(x, y, status);
              gpr_$line(x, y, status);
             }
          }
      }

/*Make the display bitmap current and then set 3 attributes in its attribute
 block.  Every 16th bit will be painted with the fill value; the other 15 bits
 will be painted with the background fill value.*/
   gpr_$set_bitmap(display_bitmap, status);
   gpr_$set_fill_value(fill_value, status);
   gpr_$set_fill_background_value(background_value, status);
   gpr_$set_fill_pattern(bitmap_desc_of_tile_pattern, scale, status);

/*Draw a filled circle using the current tile pattern.*/
   gpr_$circle_filled(center, radius, status);
   pause(5.0); /*Pause for 5 seconds, then terminate.*/
   gpr_$terminate((short)0, status);
}
```

A Program To Draw Thick and Dashed Lines (From Chapter 3)

```
/* Name of Program -- widths_and_patterns */
/*This program draws one thick line and two dashed lines.  It demonstrates how
 to use gpr_$set_draw_width, gpr_$set_line_pattern, and gpr_$set_draw_pattern.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

 short int            width_of_line_in_pixels =  11;
 short int            repeat_count, length, scale;
 gpr_$line_pattern_t  pattern;
 gpr_$linestyle_t     style;

#include "my_include_file.c"/*Contains the init, check, and pause routines.*/
main()
{
  init(gpr_$borrow);

/*The following sequence draws a line 11-pixels thick spanning x from 0 to 300

 and y from 245 to 255 inclusive.
*/
  gpr_$set_draw_width(width_of_line_in_pixels, status);
  gpr_$move(  0, 250, status);
  gpr_$line(300, 250, status);

/*The following sequence creates a dashed line.  The dashed line pattern
 repeats itself every 150 pixels (which is equal to the repeat count times
 the length).  The pattern consists of 100 pixels set to the draw value,
 followed by 50 pixels set to the background value.  We specified this pattern

 by setting the length equal to 3 which instructed gpr_$set_line_pattern to
 evaluate only the three most significant bits in pattern.  The three most
 significant bits are set to '110'.  When each bit is magnified by the repeat
 count, the pattern becomes fifty 1's, fifty 1's, and fifty 0's.
*/
  repeat_count =  50;
  pattern[0] =  0xC000; /* 1100000000000000 in binary */
  length =  3;
  gpr_$set_line_pattern(repeat_count, pattern, length, status);
  gpr_$set_draw_width(1, status);
  gpr_$move(  0, 450, status);
  gpr_$line(500, 450, status);

/*The gpr_$set_linestyle call is an alternative to the gpr_$set_line_pattern
 call.  It is simpler to use, but is consequently less powerful.  The
 following sequence creates a pattern that repeats itself every 100 pixels,
 consisting of 50 drawn pixels followed by 50 background pixels.*/
  style =  gpr_$dotted;
  scale =  50;
  gpr_$set_linestyle(style, scale, status);
  gpr_$move(0, 650, status);
  gpr_$line(650, 650, status);

  pause(5.0); /*Pause for 5 seconds, then terminate.*/
  gpr_$terminate((short)0, status);
}
```

A Program To Write a Simple String to the Display (From Chapter 4)

```
/* Name of Program -- simple_text_example */
/*This program writes a simple string to the display.  It demonstrates the
 gpr_$load_font_file, gpr_$set_text_font, and gpr_$text routines.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

 name_$pname_t pathname_of_font = {"f9x15"};
 short int     pathname_length = 5, font_id, string_length = 27;
 char          string_to_write[]  =  {"Hello from Apollo computer."};

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow);

/*Load a font.*/
   gpr_$load_font_file(pathname_of_font, pathname_length, font_id, status);
   check("loading a load_font_file");
   gpr_$set_text_font(font_id, status);
   check("setting the text font");

/*Write the string so that it begins at position 100,100 */
   gpr_$move(100,100,status);
   gpr_$text(string_to_write, string_length, status);

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Uses More Than One Text Font (From Chapter 4)

```
/* Name of Program -- three_fonts */
/*This program writes a string to the display in three different fonts.  It
 demonstrates the gpr_$load_font_file, gpr_$set_text_font, gpr_$text, and
 gpr_$inq_text_extent routines.  The program uses the gpr_$inq_text_extent
 call to measure the string.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

  gpr_$offset_t  size_in_pixels;
  short int      count =  0;
  short enum FONT_SIZES {large, medium, small} pick = large;
  short int      font_id[3];
  char     output_string[] = {"The rain in Spain falls mainly on the plain."};
  short int      string_length = sizeof(output_string);
main()
{
   init(gpr_$borrow);

/*Load three fonts into the font storage area of display memory.*/
   gpr_$load_font_file("f9x15", (short)5, font_id[(short)large],  status);
   check("loading large font");
   gpr_$load_font_file("f7x13", (short)5, font_id[(short)medium], status);
   check("loading medium font");
   gpr_$load_font_file("f5x7",  (short)4, font_id[(short)small],  status);
   check("loading small font");

   do
/*Establish one of the three fonts as the current font.*/
     {gpr_$set_text_font(font_id[(short)pick], status);
      gpr_$move(100, (short)(++count * 100), status);

/*Write the string to the display in the current font.*/
      gpr_$text(output_string, string_length, status);

/*Calculate how much space (in pixels) the string requires.*/
      gpr_$inq_text_extent(output_string, string_length, size_in_pixels,
                           status);

/*Write the space information to stdout. */
      printf("The string requires %d pixels in the x dimension.\n",
              size_in_pixels.x_size);
      ((short)pick)++;
     }
   while(pick <= small)

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Writes Text in Different Directions (From Chapter 4)

```
/* Name of Program -- text_direction */
/* This program writes a simple string to the display in each of the four
   possible printing directions.  It demonstrates gpr_$set_text_path.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

 short int          font_id;
 gpr_$direction_t   direction;
 char               str[4][7]   = {" UP   ", " DOWN ", " LEFT ", " RIGHT"};
 short              count       = 0;

main()
{
      init(gpr_$borrow);

/*Load a font and make it current.*/
      gpr_$load_font_file("f9x15", 5, font_id, status);
      check("loading font");
      gpr_$set_text_font(font_id, status);

/*The following sequence loops through each of the four possible print
 directions, printing one string in each direction.  Note that the starting
 position for each string is (500,400)
*/
      for (direction = gpr_$up; direction <= gpr_$right; ((short)direction)++)

         {gpr_$move(500, 400, status);
          gpr_$set_text_path(direction, status);
          gpr_$text(str[count++], 7, status);
         }

/*Pause for 5 seconds, then terminate.*/
      pause(5.0);
      gpr_$terminate(false, status);
}
```

A Program That Modifies a Text Font's Characteristics (From Chapter 4)

```
/* Name of Program -- modifiable_fonts */
/*This program creates a modifiable copy of a font and then changes the gap
 between characters and the width after every period.  It demonstrates the
 gpr_$replicate_font, gpr_$set_character_width, and
 gpr_$set_horizontal_spacing routines.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

   name_$pname_t   pathname_of_font =  {"f9x15"};
   short int       pathname_length  =  5;
   short int       font_id, replicated_font_id;
   char    string_to_write[]  =  {"Hi there.Good morning.My name is Apollo."};
   short int       string_length =  sizeof(string_to_write);
   char            character_to_change;
   short int       new_width_in_pixels;
   short int       horizontal_spacing_in_pixels;

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow);

/*Prepare a font, make a copy of the font, and then make the copy of the
 font current.*/
   gpr_$load_font_file(pathname_of_font, pathname_length, font_id, status);
   gpr_$replicate_font(font_id, replicated_font_id, status);
   gpr_$set_text_font(replicated_font_id, status);

/*Tighten the gap between characters by one pixel. This will cut off the right
 edge of wide characters.*/
   horizontal_spacing_in_pixels = -1;
   gpr_$set_horizontal_spacing(replicated_font_id,
                               horizontal_spacing_in_pixels, status);

/*But, leave a gap of 45 pixels after every period.*/
   character_to_change =  ´.´;
   new_width_in_pixels =  45;
   gpr_$set_character_width(replicated_font_id, character_to_change,
                            new_width_in_pixels, status);

/*Write the string.*/
   gpr_$move(100,100,status);
   gpr_$text(string_to_write, string_length, status);

/*Pause for 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program To Create a Hidden Memory Bitmap (From Chapter 5)

```
/* Name of Program -- hidden_memory_bitmaps */
/*This program creates one hidden display memory bitmap, writes a circle to
 it, and then blts it to display memory so that the circle becomes visible.
 It demonstrates the gpr_$allocate_hdm_bitmap, gpr_$allocate_bitmap,
 gpr_$set_bitmap, and gpr_$pixel_blt routines.  */
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
   gpr_$offset_t         size_of_hdm_bitmap;
   gpr_$attribute_desc_t attribute_block_descriptor;
   gpr_$bitmap_desc_t    hdm_bitmap_descriptor, bitmap_descriptor;
   gpr_$position_t       center_of_circle = {25,25};
   short int             radius_of_circle =  20;
   gpr_$window_t         source_window;
   gpr_$position_t       dest_origin;
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
  init(gpr_$borrow);

/*The following sequence creates a 224 by 224 HDM bitmap.  Notice how
 the attribute_block_descriptor returned by gpr_$allocate_attribute_block
 is used as an input parameter in gpr_$allocate_hdm_bitmap.  The value for
 hi_plane comes from "my_include_file.c"
*/
  gpr_$allocate_attribute_block(attribute_block_descriptor, status);
  size_of_hdm_bitmap.x_size =  224;  size_of_hdm_bitmap.y_size =  224;
  gpr_$allocate_hdm_bitmap(size_of_hdm_bitmap, hi_plane,
            attribute_block_descriptor, hdm_bitmap_descriptor, status);
  check("allocating hdm bitmap");

/*The current bitmap is now the display bitmap. Therefore, if we call a
 drawing or text routine, all data will be written to the display
 bitmap.  However, we want to draw a filled circle in the HDM bitmap.
 Before drawing the filled circle, we must make the HDM bitmap current
 by calling the gpr_$set_bitmap routine.  Since the gpr_$allocate_hdm_bitmap
 routine does not clear the data stored in hidden memory, we must
 clear it with the gpr_$clear routine.*/
  gpr_$set_bitmap(hdm_bitmap_descriptor, status);
  gpr_$clear((long)0, status);
  gpr_$circle_filled(center_of_circle, radius_of_circle, status);

/*We now blt the entire HDM bitmap to a portion of display memory.
 The gpr_$pixel_blt command blts the specified pixels from the HDM
 bitmap to the current bitmap.  Therefore, before we call gpr_$pixel_blt,
 we must set the current bitmap back to the display bitmap.*/
  source_window.window_base.x_coord = 0;  /*Here, we specify the section*/
  source_window.window_base.y_coord = 0;  /*of the HDM bitmap          */
  source_window.window_size.x_size  = 224;/*that we want to blt.       */
  source_window.window_size.y_size  = 224;
  dest_origin.x_coord =  400;  /*Here, we specify where in display */
  dest_origin.y_coord =  400;  /*memory the system will blt to.    */
  gpr_$set_bitmap(display_bitmap, status);/*Make the display bitmap current.*/
  gpr_$pixel_blt(hdm_bitmap_descriptor,source_window,dest_origin, status);
   pause(5.0);   /*Pause for 5 seconds, then terminate.*/
  gpr_$terminate(false, status);
}
```

A Program That Toggles Between Frame 0 and Frame 1 (From Chapter 5)

```
/* Name of Program -- toggling_display_frames */
/*This program toggles between frame 0 and frame 1.  It demonstrates
 gpr_$set_bitmap_dimensions and gpr_$select_color_frame.  This program will
 only run on a DN550, DN560, DN600, or DN660.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

   gpr_$offset_t  new_size_of_bitmap;
   short int      pictures, frame, x;

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

main()
{
     init(gpr_$borrow);

/*The gpr_$init call will not permit you to specify a borrow mode display
  bitmap larger than 1024 x 1024.  Because we want to create a 1024 x 2048
  bitmap, we must call gpr_$set_bitmap_dimensions after we call gpr_$init.
*/
     new_size_of_bitmap.x_size =  1024;  new_size_of_bitmap.y_size =  2048;
     gpr_$set_bitmap_dimensions(display_bitmap, new_size_of_bitmap, hi_plane,
                                status);
     check("resetting dimensions");
/*It is a good idea to clear the entire display since HDM probably contains
  fonts,icons, etc.*/
     gpr_$clear((long)0, status);

/*We will now draw horizontal lines in frame 0 and vertical lines in frame 1,
 and toggle the two frames at .1 second intervals to examine these
 fast-changing images.*/
     for (pictures = 1; pictures <= 200; pictures++)
        {if (frame = (pictures % 2))
           {gpr_$move((short)0,  pictures, status);
            gpr_$line((short)200,pictures, status);
           }
        else
           {x = pictures + 300;
            gpr_$move(x, (short)1024, status);
            gpr_$line(x, (short)1224, status);
           }
        gpr_$select_color_frame(frame, status);
        pause(0.1);
        }

 /*Pause 5 seconds, then terminate.*/
     pause(5.0);
     gpr_$terminate(false, status);
 }
```

A Program That Creates a Main Memory Bitmap (From Chapter 5)

```
/* Program main_memory_bitmaps; */
/*This program creates one main memory bitmap, writes a circle in it, and blts
 the circle to display memory where it can become visible on the screen.
 It demonstrates gpr_$allocate_attribute_block, gpr_$allocate_bitmap,
 gpr_$set_bitmap, and gpr_$pixel_blt.  */
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

 gpr_$offset_t         size_of_main_memory_bitmap;
 gpr_$attribute_desc_t attribute_block_descriptor;
 gpr_$bitmap_desc_t    main_memory_bitmap_descriptor;
 gpr_$position_t       center_of_circle =  {25,25}  ;
 short                 radius_of_circle =  20  ;
 gpr_$window_t         source_window;
 gpr_$position_t       destination_origin;
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
    init(gpr_$borrow);
/*The following sequence creates a 50 by 50 main memory bitmap.  Notice how
 the attribute_block_descriptor returned by gpr_$allocate_attribute_block
 is used as an input parameter in gpr_$allocate_bitmap.  The value for
 hi_plane comes from ´my_include_file.pas´
*/
    gpr_$allocate_attribute_block(attribute_block_descriptor, status);
    size_of_main_memory_bitmap.x_size =  50;
    size_of_main_memory_bitmap.y_size =  50;
    gpr_$allocate_bitmap(size_of_main_memory_bitmap, hi_plane,
                         attribute_block_descriptor,
                         main_memory_bitmap_descriptor, status);

/*The current bitmap is now the display bitmap. Therefore, if we call a figure
 drawing or text writing routine, all data will be written to the display
 bitmap.  However, we want to draw a filled circle in the main memory bitmap.
 Before drawing the filled circle, we must make the main memory bitmap current
 by calling the gpr_$set_bitmap routine.
*/
    gpr_$set_bitmap(main_memory_bitmap_descriptor, status);
    gpr_$circle_filled(center_of_circle, radius_of_circle, status);

/*We now blt the entire main memory bitmap to a portion of display memory.
 The gpr_$pixel_blt command blts the specified pixels from the main memory
 bitmap to the current bitmap.  Therefore, before we call gpr_$pixel_blt,
 we must set the current bitmap back to the display bitmap.*/
    source_window.window_base.x_coord =    0; /*Here, we specify the section*/
    source_window.window_base.y_coord =    0; /*of the main memory bitmap   */
    source_window.window_size.x_size  =  100; /*that we want to blt.        */
    source_window.window_size.y_size  =  100;
    destination_origin.x_coord = 400; /*Here, we specify where in display */
    destination_origin.y_coord = 400; /*memory the system will blt to.    */
    gpr_$set_bitmap(display_bitmap,status);/*Make the display bitmap current*/
    gpr_$pixel_blt(main_memory_bitmap_descriptor, source_window,
                   destination_origin, status);
    pause(5.0); /*Pause for 5 seconds, then terminate.*/
    gpr_$terminate(false, status);
}
```

A Program That Creates a Pixel Oriented External File Bitmap (From Chapter 5)

```
/* Name of Program -- pixel_oriented_bitmaps */
/*This program creates a pixel-oriented bitmap for a 4- or 8-plane color node.
  You should run this program twice.  The first time, select the ´create´
  option so that the program will create a file containing a bitmap.  The
  second time, select the ´display´ option so that the program will display
  the contents of this file on your screen.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, pause, and check routines.*/
#define  x_size_of_bitmap           200
#define  y_size_of_bitmap           200
#define  number_of_bytes_per_pixel    1
typedef  char            color_range;

 char                    choice;
 char                    pathname_of_bitmap[256], *pstr;
 gpr_$attribute_desc_t   attribs;
 short int               bytes_per_line;
 color_range             memory[0x7FFFFF], *storage;
 gpr_$bitmap_desc_t      disk_bitmap;
/*******************************************************************************/
void access_external_bitmap(access)
gpr_$access_mode_t  access;
{
 short int          pathname_len;
 gpr_$version_t     version;
 gpr_$offset_t      size;
 short int          groups;
 gpr_$bmf_group_header_array_t  header;
 unsigned char      created;

   for (pstr = pathname_of_bitmap; *++pstr; pathname_len++); /*pathname_len*/
   version.major =  1;   version.minor =  1;                       /*version*/
   size.x_size = x_size_of_bitmap; size.y_size = y_size_of_bitmap;   /*size*/
   groups      =  1;                                                /*groups*/
   header[0].n_sects =  1;                                          /*header*/
   header[0].pixel_size =  hi_plane + 1;
   header[0].allocated_size =  8;
   header[0].bytes_per_line =  0;
   header[0].bytes_per_sect =  0;
   header[0].storage_offset =  0;

   gpr_$open_bitmap_file(access,pathname_of_bitmap,pathname_len,version,size,
                    groups, header, attribs, disk_bitmap, created, status);
   check("opening an external file bitmap");
   printf("allocated_size = %hd\n", header[0].allocated_size);
   bytes_per_line          = header[0].bytes_per_line;
   storage                 = header[0].storage_offset;
}
/*******************************************************************************/
void set_1_pixel_in_external_bitmap(x, y, color_index)
short         x, y;
color_range   color_index;
{
 long     i, offset;
/*Convert a two-dimensional pixel position to a one-dimensional offset from
 the beginning of the bitmap.*/
   offset =  (y * bytes_per_line) + (x * number_of_bytes_per_pixel);

/*Now, load the color index for this pixel into the bitmap.*/
   *(storage + offset) =  color_index;
}
/****************************************************************************/
void draw_figure() /*This simple routine calculates the color value
for every pixel in the bitmap, but it does not actually store any values in
the bitmap.  The set_1_pixel_in_external bitmap routine does that.*/
{
 short           x,y,q;
 color_range     color_index;

   for (x = 0; x <= 200; x++)
      {for (y = 1; y <= 200; y++)
          {color_index =  (y % 16);
           set_1_pixel_in_external_bitmap (x, y, color_index);
          }
      }
}
/****************************************************************************/
void display_the_bitmap()
{
 static gpr_$position_t  dest_origin   =  {400,400};
 static gpr_$window_t    source_window =  {{0,0},{x_size_of_bitmap,
                                                  y_size_of_bitmap}};
   access_external_bitmap(gpr_$readonly);

/*BLT the external file bitmap from disk to display memory.*/
   gpr_$set_bitmap(display_bitmap, status);
   gpr_$pixel_blt(disk_bitmap, source_window, dest_origin, status);
}
/****************************************************************************/
main()
{
   printf("Do you want to create a bitmap file or display a bitmap file?\n");
   printf("(Enter ´c´ or ´d´) -- ");
   scanf("%c", &choice);
   if (choice == ´c´)
     printf("Be patient; it will take some time to create the bitmap.\n");
   printf("What is the pathname of the file. -- ");
   scanf("%s", pathname_of_bitmap);
   init(gpr_$borrow);
   if (choice == ´c´)
     { gpr_$allocate_attribute_block(attribs,status);
       access_external_bitmap(gpr_$create);
       draw_figure();
     }
   else
     { display_the_bitmap();
       pause(5.0);
     }
   gpr_$terminate(false, status);/*Terminate the graphics package.*/
}
```

A Program That Demonstrates Double-Buffer Techniques (From Chapter 5)

```
/* Name of Program -- double_buffer_example */
/*This program demonstrates double buffering techniques. It uses the
  gpr_$allocate_buffer, gpr_$set_bitmap, gpr_$inq_visible_buffer,
  gpr_$select_display_buffer routines.  This program can only be run on a
  machine that supports double buffering.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, pause, and check routines.*/


 gpr_$bitmap_desc_t    visible_bitmap, invisible_bitmap;
 gpr_$bitmap_desc_t    display_bitmap, buffer_bitmap;
/****************************************************************************/
void create_a_buffer_bitmap()
/*This function allocates a buffer bitmap and then clears its contents.*/
{
   gpr_$allocate_buffer(display_bitmap, buffer_bitmap, status);
   check("allocating a buffer bitmap");
   gpr_$set_bitmap(buffer_bitmap, status);
   gpr_$clear((long)0, status);
}
/****************************************************************************/
void which_bitmap_is_visible()
{
   gpr_$inq_visible_buffer(visible_bitmap, status);
   if (visible_bitmap == display_bitmap)
     invisible_bitmap =  buffer_bitmap;
   else
     invisible_bitmap =  display_bitmap;
}
/****************************************************************************/
void      draw_pattern_in_invisible_bitmap(count)
short int count;
/*This function draws one circle to the bitmap that is currently invisible.*/
{
       gpr_$position_t       center;
static short int             radius = 80;

    which_bitmap_is_visible();
/*Make the invisible bitmap current so that we can draw to it:*/
    gpr_$set_bitmap(invisible_bitmap, status);

/*Draw a filled circle in the current bitmap.*/
    center.x_coord =  radius + (count * 12);
    if (count % 2)
      center.y_coord = 100;
    else
      center.y_coord = 400;

    gpr_$set_fill_value( (long)((count % 8) + 1), status);
    gpr_$circle_filled(center, radius, status);
}
/****************************************************************************/
void make_invisible_bitmap_visible(options)
gpr_$double_buffer_option_t  options;
```

```
{
gpr_$bitmap_desc_t display_desc;/*the bitmap you want displayed*/
gpr_$bitmap_desc_t option_desc; /*the bitmap you want to make invisible*/
linteger      option_value = 0; /*ignored unless options = gpr_$clear_buffer*/

   which_bitmap_is_visible();
   display_desc = invisible_bitmap;
   option_desc  = visible_bitmap;

/*Make invisible bitmap visible, and make visible bitmap invisible. */
   gpr_$select_display_buffer(display_desc, option_desc, option_value,
                              options, status);
}
/****************************************************************************/
void load_font()
{
short int    fontid;
    gpr_$load_font_file("/sys/dm/fonts/f7x13", 19, fontid, status);
    check("loading the font file");
    gpr_$set_text_font( fontid, status);
    gpr_$set_text_value((long)7, status );
}
/****************************************************************************/
void print_text_in_invisible_bitmap(number)
short int  number;
{
#define  print_text_at_x  350
#define  print_text_at_y  250
char                string[15], number_as_a_string[3];

   which_bitmap_is_visible();
   if (invisible_bitmap == display_bitmap)
     strcpy(string,"DISPLAY BITMAP");
   else
     strcpy(string,"BUFFER BITMAP ");

   gpr_$move(print_text_at_x, print_text_at_y, status);
   gpr_$text(string, 15, status);

   number_as_a_string[0] =  (number / 10) + 48;
   number_as_a_string[1] =  (number % 10) + 48;
   gpr_$move(print_text_at_x, print_text_at_y + 20, status);
   gpr_$text(number_as_a_string, 2, status);
}
/****************************************************************************/
main()
{
#define images_to_display 50
short int                    counter, w;
float                        time_to_pause_between_images;
gpr_$double_buffer_option_t  what_to_do_to_invisible_bitmap ;

    printf("Enter the pause time (in seconds) between images -- \n");
    scanf("%f", &time_to_pause_between_images);

    printf("Enter 0 for gpr_$clear_buffer,\n");
    printf("      1 for gpr_$undisturbed_buffer,\n");
    printf("   or 2 for gpr_$copy_buffer -- \n");
    scanf("%hd", &w);
    what_to_do_to_invisible_bitmap = (gpr_$double_buffer_option_t)w;

    init(gpr_$borrow);        /*create the primary bitmap.*/
    create_a_buffer_bitmap(); /*create the buffer bitmap.*/
    load_font();

    for (counter = 1; counter <= images_to_display; counter++)
       {draw_pattern_in_invisible_bitmap(counter);
        print_text_in_invisible_bitmap(counter);
        make_invisible_bitmap_visible(what_to_do_to_invisible_bitmap); /*and
        make the visible bitmap invisible.*/
        pause(time_to_pause_between_images);
       }

/*Pause for 5 seconds; then terminate.*/
    pause(5.0);   /*So user can examine the final image.*/
    gpr_$terminate(false, status);
}
```

A Program That Demonstrates The Three Different Kinds of Block Transfers (BLTs) (From Chapter 5)

```
/* Name of Program -- blts */
/*This program contrasts the three BLT calls.  It demonstrates the
  gpr_$bit_blt, gpr_$pixel_blt, and gpr_$additive_blt calls.  This program
  must be run on a color node.*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

main(){
static gpr_$position_t           center              =  {50,50};
static short int                 radius              =  40;
static gpr_$pixel_value_t        fill_value          =  6;
static gpr_$offset_t             main_mem_bm_size    =  {100, 100};
       gpr_$attribute_desc_t     attrib_desc;
       gpr_$bitmap_desc_t        main_mem_bitmap;
       gpr_$bitmap_desc_t        source_bitmap_desc;
static gpr_$window_t             source_window       =  {{0, 0},{100, 100}};
       gpr_$position_t           destination_origin ;
       gpr_$rgb_plane_t          source_plane       ;
       gpr_$rgb_plane_t          destination_plane  ;
static gpr_$offset_t             size                =  {500,500};

   init(gpr_$borrow);

/*Create a main memory bitmap.*/
   gpr_$allocate_attribute_block(attrib_desc, status);
   gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc,
                        main_mem_bitmap, status);

/*Draw a filled circle in the main memory bitmap.*/
   gpr_$set_bitmap(main_mem_bitmap, status);
   gpr_$set_fill_value(fill_value, status);
   gpr_$circle_filled(center, radius, status);

   source_bitmap_desc =  main_mem_bitmap;
   gpr_$set_bitmap(display_bitmap, status);

/*BLT the circle from main memory to display memory three different ways.*/
/*First, BLT the contents of every plane in the main memory bitmap to every
  plane in the display memory bitmap.
*/
   destination_origin.x_coord =  400; destination_origin.y_coord =  0;
   gpr_$pixel_blt(source_bitmap_desc, source_window, destination_origin,
                  status);
   check("pixel blt");

/*Second, BLT the contents of plane 1 in the main memory bitmap to plane 3
  of the display memory bitmap.
*/
   source_plane =  1;
   destination_plane =  3;
   destination_origin.x_coord =  400; destination_origin.y_coord =  200;
   gpr_$bit_blt(source_bitmap_desc, source_window, source_plane,
                destination_origin, destination_plane, status);
   check("bit blt");
/*Third, BLT the contents of plane 0 in the main memory bitmap to every plane
  of the display memory bitmap.  Since plane 0 contains all zeros, the
  circle will probably be displayed in black and may therefore be invisible.
*/
   source_plane =  0;
   destination_origin.x_coord =  400; destination_origin.y_coord =  400;
   gpr_$additive_blt(source_bitmap_desc, source_window, source_plane,
                     destination_origin, status);
   check("additive blt");

/*Pause 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Sets and Employs Two Attribute Blocks for the Same Bitmap (From Chapter 5)

```
/* Name of Program -- attribute_blocks */
/*This program allocates two different attribute blocks and assigns different
 attributes to them.  It demonstrates the gpr_$allocate_attribute_block and
 gpr_$set_attribute_block routines.  If you run this program on a monochrome
 display, you'll get one set of attributes, and if you run it on a color
 display, you'll get a different set of attributes.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
  gpr_$attribute_desc_t  attrib_desc1, attrib_desc2 ;
  gpr_$bitmap_desc_t     main_mem_bitmap;
  gpr_$offset_t          main_mem_bm_size   =  {400, 400};
  gpr_$window_t          source_window      =  {{0, 0},{400, 400}};
  gpr_$position_t        destination_origin =  {100, 100};
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow);

/*Allocate two attribute blocks.  Both attribute blocks will begin with the
 default attributes.*/
   gpr_$allocate_attribute_block(attrib_desc1, status);
   gpr_$allocate_attribute_block(attrib_desc2, status);

/*Allocate a main memory bitmap and make it current.  The third argument could
 have been attrib_desc1 or attrib_desc2; it doesn't matter. */
   gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc1,
                        main_mem_bitmap, status);
   gpr_$set_bitmap(main_mem_bitmap, status);

/*Define nondefault attributes for a monochrome node.*/
   if (hi_plane == 0)
     {
  /*Associate the attributes in attrib_desc1 with the main memory bitmap.*/
              gpr_$set_attribute_block(attrib_desc1, status);
  /*Assign one nondefault attribute value to this attribute block.*/
              gpr_$set_draw_width((short)20, status);
     }
/*Define nondefault attributes for a color node.*/
   else {
  /*Associate the attributes in attrib_desc2 with the main memory bitmap.*/
              gpr_$set_attribute_block(attrib_desc2, status);
  /*Assign two nondefault attribute values to this attribute block.*/
              gpr_$set_draw_width((short)4, status);
              gpr_$set_draw_value((long)3, status);
        }

/*Draw a line in the main memory bitmap from 0,0 to 300,300*/
   gpr_$line(300, 300, status);

/*BLT the main memory bitmap to the screen.*/
   gpr_$set_bitmap(display_bitmap, status);
   gpr_$pixel_blt(main_mem_bitmap, source_window, destination_origin, status);
   pause(5.0); /*Pause 5 seconds, then terminate.*/
   gpr_$terminate(false, status);
}
```

A Program That Converts a Color Bitmap to an Array, Manipulates the Array, and Converts the Array Back to a Bitmap (From Chapter 5)

```
/* Name of Program -- read_write_pixels */
/*This program demonstrates how to manipulate a bitmap with the
 gpr_$read_pixel and gpr_$write_pixel routines.  This program will work
 only on a color node.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

   long int         fill_color = 2, new_fill_color = 3, new_background_color = 1;
   short int           i,j;
   gpr_$position_t  center                 =  {50,50};
   short int        radius                 =  40;
   gpr_$window_t    source_window          =  { {0,0},   {100,100}};
   gpr_$window_t    destination_window     =  { {0,200},{100,100}};
   long int         pixel_array[100][100];

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

main()
{
   init(gpr_$borrow);

/*Draw a filled circle on the screen.*/
   gpr_$set_fill_value(fill_color, status);
   gpr_$circle_filled(center, radius, status);

/*Convert a 100x100xhi_plane portion of the display bitmap into an 100x100
  array.  Note that the gpr_$read_pixels all does not change the existing
  bitmap in anyway.*/
   gpr_$read_pixels(source_window, pixel_array, status);
   check("reading pixels");

/*Examine each element of the array.  Give each element a new fill color
  and a new background color. */
   for (i=0; i <= 99; i++)
       {for (j=0; j <= 99; j++)
           {if (pixel_array[i][j] == fill_color)
              pixel_array[i][j] = new_fill_color;
            else
              pixel_array[i][j] = new_background_color;
           }
       }
/*Now convert the modified array back to a bitmap and put it in the designated
  region of the display bitmap (where it will be displayed). */
   gpr_$write_pixels(pixel_array, destination_window, status);
   check("writing pixels");
/*Pause 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Manipulates a Main Memory Bitmap Through a Bitmap Pointer (From Chapter 5)

```
/* Name of Program -- inq_bm_ptrs_in_main_mem */
/*This program demonstrates how to access the virtual addresses of a main
 memory bitmap.  It illustrates the gpr_$inq_bitmap_pointer routine.  This
 program allocates a main memory bitmap and then sets the value of every pixel
 associated with this bitmap to color index 5.  It then BLTs the bitmap to
 the screen.  This program must be run on a color node.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#define  width_of_bitmap    480  /*This number should be divisible by 32.*/
#define  height_of_bitmap   320
#define  array_size         (width_of_bitmap / 32)
  gpr_$offset_t   main_mem_bm_size   =  {width_of_bitmap, height_of_bitmap};
  gpr_$window_t   window             =  {{0,0},
                                        {width_of_bitmap, height_of_bitmap}};
  gpr_$rgb_plane_t      plane;
  gpr_$bitmap_desc_t    main_mem_bitmap;
  short int             stor_line_width, y, x;
  gpr_$attribute_desc_t attrib_desc;
  long int              bitmap_setup[array_size], *storage_ptr;
  gpr_$position_t       dest_origin   =  {10,10};
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow);

/*Create a main memory bitmap.*/
   gpr_$allocate_attribute_block(attrib_desc, status);
   gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc,
                        main_mem_bitmap, status);

/*Set every pixel in this bitmap to color index 5.  We do this by setting
  every bit in plane 0 and plane 2 to '1', and by setting every bit in the
  other planes to '0'.
*/
  gpr_$inq_bitmap_pointer(main_mem_bitmap,storage_ptr,stor_line_width,status);
  for (plane = 0; plane <= hi_plane; plane++) /* Dimension z. */
     {for (y = 0; y < height_of_bitmap; y++)  /* Dimension y. */
         {for (x = 0; x < array_size; x++)    /* Dimension x. */
             {if ((plane == 0) || (plane == 2))
               *storage_ptr++ =  0xFFFFFFFF;  /*Set all 32 bits to "1"*/
             else
               *storage_ptr++ =  0x00000000;  /*Set all 32 bits to "0"*/
             }
/*Sometimes the main memory bitmap contains a lot of blank space at the end of
  every scan line, so we have to adjust the storage_ptr accordingly. */
          if ((x*2) < stor_line_width)
             storage_ptr += ((stor_line_width - (x*2)) / 2);
          }
     }
/*BLT the main memory bitmap to the screen.*/
  gpr_$pixel_blt(main_mem_bitmap, window, dest_origin, status);
  pause(5.0); /*Pause 5 seconds, then terminate.*/
  gpr_$terminate(true,status);
}
```

A Program That Manipulates a Display Memory Bitmap Through a Bitmap Pointer (From Chapter 5)

```
/* program inq_bm_ptrs_in_disp_mem; */
/* This program demonstrates how to access the virtual address of a display
 * memory bitmap.  It illustrates the gpr_$inq_bitmap_pointer,
 * gpr_$enable_direct_access, and gpr_$remap_color_memory routines.  */
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c"/* Contains the init, check, and pause routines.*/
#define width_of_rectangle 480
#define height_of_rectangle 320
#define array_size width_of_rectangle/32
main()
{
  gpr_$rgb_plane_t         plane;
  short                    storage_line_width, scan_line, c;
  int                      *storage_ptr_save, *storage_ptr;
  static gpr_$window_t     rectangle =  { {0,0},
                                          {width_of_rectangle,
                                           height_of_rectangle}};
  init(gpr_$borrow);

/* Display a rectangle filled with color index 15.*/
  gpr_$set_fill_value(15, status);
  gpr_$rectangle(rectangle, status);
  pause(1.0);

  gpr_$inq_bitmap_pointer(display_bitmap, storage_ptr_save,
                          storage_line_width, status);
  for (plane = 0; plane <= 3; plane++)
  {
/*The first time through the loop, the system will remap color memory so that
 *the starting virtual address of plane 0 will be equal to storage_ptr_save.
 *The next time through the loop, the system will remap color memory so that
 *the starting virtual address of plane 1 will be equal to storage_ptr_save,
 *and so on with planes 2 and 3.  We must reestablish the value of storage_ptr
 *each time through the loop.
*/
     gpr_$remap_color_memory(plane, status);
     storage_ptr = storage_ptr_save;

/* It is good programming practice to call gpr_$enable_direct_access before
 * writing directly to the bitmap (i.e., before changing bits without using
 * GPR calls).*/
     gpr_$enable_direct_access(status);

     for (scan_line = 0; scan_line < height_of_rectangle; scan_line++)
     {
        for (c = 0; c < width_of_rectangle/32; c++)
          *(storage_ptr + c) =  0x00F0F0F0;
        storage_ptr =  storage_ptr + storage_line_width/2;
     }
     pause(1.0);
   }
  pause(2.0); /* Pause 2 seconds, then terminate.*/
   gpr_$terminate(true,status);
}
```

A Program That Manipulates a Hidden Display Memory Bitmap Through a Bitmap Pointer (From Chapter 5)

```
/* PROGRAM inq_bm_ptrs_in_hdm; */
/* This program demonstrates how to access the virtual addresses of a HDM
 * bitmap.  It illustrates the gpr_$inq_bitmap_pointer routine.  This program
 * allocates a HDM and then sets the value of every pixel associated with
 * this bitmap to color index 5 (which is yellow on the default color map).
 * It then BLTs the HDM bitmap to the screen.  This program must be run on
 * a color node.
*/

#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

#define WIDTH_OF_BITMAP 128
#define HEIGHT_OF_BITMAP 64
#define ARRAY_SIZE WIDTH_OF_BITMAP/32

main()
{
  static gpr_$window_t   source_window = {{0,0},{128,64}};
  short                  stor_line_width, index, plane, scan;
  gpr_$attribute_desc_t  attrib_desc;
  static gpr_$offset_t   bitm_size = {1024,800};
static gpr_$offset_t  size_of_hdm_bitmap = {WIDTH_OF_BITMAP,HEIGHT_OF_BITMAP};
  gpr_$bitmap_desc_t     hdm_bitmap;
  int                    *hdm_ptr_save;
  int                    *hdm_ptr;
  static gpr_$position_t dest_origin = {10,10};

  init(gpr_$borrow);

/* Create hidden display memory bitmap. */
  gpr_$allocate_attribute_block(attrib_desc, status);
  gpr_$allocate_hdm_bitmap(size_of_hdm_bitmap, hi_plane, attrib_desc,
                           hdm_bitmap, status);

  gpr_$inq_bitmap_pointer(hdm_bitmap, hdm_ptr_save, stor_line_width, status);

  for (plane = 0; plane <= hi_plane; plane++)
  {
    gpr_$remap_color_memory(plane, status);
    hdm_ptr = hdm_ptr_save;
    gpr_$enable_direct_access( status );
    for (scan = 0; scan < HEIGHT_OF_BITMAP; scan++)
    {
      for (index = 0; index < ARRAY_SIZE; index++)
      {
      /* Set every bit in plane 0 and plane 2 to '1'.  Set every bit in all
other
       * planes to '0'
       */
        if (plane == 0 || plane == 2)
          *(hdm_ptr+index) = 0xFFFFFFFF;
        else
          *(hdm_ptr+index) = 0x00000000;
      }
      hdm_ptr = hdm_ptr + (stor_line_width/2);
    }
  }

/* BLT the HDM bitmap to the screen. */
   gpr_$pixel_blt(hdm_bitmap, source_window, dest_origin, status);

/*Pause 5 seconds, then terminate.*/
  pause(5.0);
  gpr_$terminate(true,status);
}
```

A Program That Draws a Line, Fills a Triangle, and Writes Some Text in Different Colors (From Chapter 6)

```
/* Program setting_colors; */
/*This program controls the colors used to draw a line, fill a triangle, and
 print text.  It demonstrates the gpr_$set_draw_value, gpr_$set_fill_value,
 gpr_$set_text_value, and gpr_$set_text_background_value.  Note that this
 program does not establish a new color map; therefore, the colors displayed
 on your screen will depend on whatever is stored in the current color map.*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
/****************************************************************************/
void draw_a_colored_line()
{
   gpr_$set_draw_value(1L, status); /* Draw a line with color index 1. */
   gpr_$line(90, 150, status);
}
/****************************************************************************/
void fill_a_triangle_with_color()
{
static gpr_$position_t  v[3] = {{100,100},{100,300},{300,100}};

  gpr_$set_fill_value(2L, status); /* Fill a triangle with color index 2. */
  gpr_$triangle(v[0], v[1], v[2], status);
}
/****************************************************************************/
void write_text_in_color()
{
short int    font_id;
   gpr_$load_font_file("/sys/dm/fonts/f9x15", 19, font_id, status);
   check("loading the font");
   gpr_$set_text_font(font_id, status);
   gpr_$set_text_value(3L, status);  /*Write text with color index 3.*/

/*The default text background index is 0.*/
   gpr_$move(20, 320, status);
   gpr_$text("Greetings from the written world", 32, status);

/*But we can establish a nondefault text background index.*/
   gpr_$set_text_background_value(7L,status);
   gpr_$move(20, 350, status);
   gpr_$text("Greetings from the written world", 32, status);
}
/****************************************************************************/
main()
{
   init(gpr_$borrow);
   draw_a_colored_line();
   fill_a_triangle_with_color();
   write_text_in_color();

/*Pause 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Alters the Color Map (From Chapter 6)

```
/* Name of Program -- color_circles */
/*This program demonstrates how to build a new color map. It demonstrates
 the gpr_$set_color_map call.*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
/****************************************************************************/
long int  color_entry(red, green, blue)
short red, green, blue;
{
    return( (red < 16) + (green < 8) + blue );
}
/****************************************************************************/
void create_color_map()
{
 static gpr_$pixel_value_t  start_index = 0;
 static short int           number_of_entries = 8;
        gpr_$color_vector_t color_map;

    color_map[0] =  color_entry (0,0,0);       /*color--black*/
    color_map[1] =  color_entry (0,0,255);     /*color--dark blue*/
    color_map[2] =  color_entry (50,50,255);
    color_map[3] =  color_entry (75,75,255);
    color_map[4] =  color_entry (100,100,255);
    color_map[5] =  color_entry (150,150,255);
    color_map[6] =  color_entry (200,200,255); /*color--light blue*/
    color_map[7] =  color_entry (255,255,255); /*color--white*/

/*Now that the array color_map contains the correct values, establish it
 as the real system color map.*/
    gpr_$set_color_map(start_index,number_of_entries, color_map, status);
}
/****************************************************************************/
void draw_concentric_circles() /*Draw 7 concentric circles - each a lighter*/
{                              /*shade of blue. The last circle is white.*/
 static gpr_$position_t    center =  {300,300};
 static short int          radius =  300;
        gpr_$pixel_value_t index;

   for (index = 1; index <= 7; index++)
      {gpr_$set_fill_value(index,status);
       gpr_$circle_filled(center,radius,status);
       radius -= 40;
       printf("%d,%d\n", index, radius);
      }
}
/****************************************************************************/
main()
{
   init(gpr_$borrow);
   create_color_map();
   draw_concentric_circles();
   pause(5.0);  /*Pause 5 seconds, then terminate.*/
   gpr_$terminate(false, status);
}
```

A Program That Saves the Original Color Map, Loads a New Color Map, and Then Restores the Original Color Map (From Chapter 6)

```
/* Name of Program -- restore_color_table */
/*This program demonstrates how to save the current color map, create a new
 color map, and then restore the original color map.  It demontrates the
 gpr_$inq_color_map and gpr_$set_color_map routines.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

 gpr_$pixel_value_t  start_index = 0;
 short int           number_of_entries = 16, n;
 gpr_$color_vector_t old_color_map, new_color_map;

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
/***************************************************************************/
void draw_circles()  /*Draw 16 circles to illustrate the current contents*/
{                    /*of the color map.*/
 gpr_$position_t     center;
 gpr_$pixel_value_t  n;
    for (n = 0; n <= 15; n++)
       {gpr_$set_fill_value(n, status);
        check("Setting fill value");
        center.x_coord =  50;  center.y_coord =  25*n;
        gpr_$acquire_display(status);
        gpr_$circle_filled(center, 15, status);
        check("Filling circle");
        gpr_$release_display(status);
       }
}
/***************************************************************************/
void save_old_color_map()  /*Save the color map the system was using. */
{
    gpr_$inq_color_map(start_index, number_of_entries, old_color_map, status);
    check("inquiring about color map");
}
/***************************************************************************/
void create_new_color_map() /*Load a new color map.*/
{
    for (n = 2; n <= 15; n++)
      new_color_map[n] =  (15*n < 16) + (255 < 8) + (255-(15*n));

    gpr_$acquire_display(status);
    gpr_$set_color_map(start_index, number_of_entries, new_color_map, status);
    check("Setting color map");
    gpr_$release_display(status);
}
/***************************************************************************/
void restore_old_color_map()  /*Reload the color map the system was using*/
{
    gpr_$acquire_display(status);
    gpr_$set_color_map(start_index, number_of_entries, old_color_map, status);
    check("restoring color map");
    gpr_$release_display(status);
}
```

```
/****************************************************************************/
main()
{
   init(gpr_$direct);

   save_old_color_map();     draw_circles();   pause(2.0);
   create_new_color_map();                     pause(2.0);
   restore_old_color_map();                    pause(2.0);

/*Terminate.*/
   gpr_$terminate(false, status);
}
```

A Program That Creates a Plane Oriented Bitmap Containing a Special Color Map (From Chapter 6)

```
/* Name of Program -- plane_oriented_bitmap */
/* There's a bug in this program, and I don't know what it is. */
/*This program creates a pixel-oriented bitmap for a 4- or 8-plane color node.
  You should run this program twice.  The first time, select the 'create'
  option so that the program will create a file containing a bitmap.  The
  second time, select the 'display' option so that the program will display
  the contents of this file on your screen.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, pause, and check routines.*/

#define  x_size_of_bitmap   200
#define  y_size_of_bitmap   200
#define  n_of_colors        (short)16
#define  draw_value         (long)15
typedef  short              color_range;

 char                    choice;
 char                    pathname_of_bitmap[256], *pstr;
 gpr_$attribute_desc_t   attribs;
 short int               bytes_per_line;
 color_range             memory[0x7FFFFF], *storage;
 gpr_$bitmap_desc_t      disk_bitmap;
/****************************************************************************/
void access_external_bitmap(access)
gpr_$access_mode_t  access;
{
 short int          pathname_len;
 gpr_$version_t     version;
 gpr_$offset_t      size;
 short int          groups;
 gpr_$bmf_group_header_array_t  header;
 unsigned char      created;

   for (pstr = pathname_of_bitmap; *++pstr; pathname_len++); /*pathname_len*/
   version.major =  1;   version.minor =  1;                      /*version*/
   size.x_size = x_size_of_bitmap; size.y_size = y_size_of_bitmap;   /*size*/
   groups      =  1;                                               /*groups*/
   header[0].n_sects =  hi_plane + 1;                              /*header*/
   header[0].pixel_size =      1;
   header[0].allocated_size =  0;
   header[0].bytes_per_line =  0;
   header[0].bytes_per_sect =  0;
   header[0].storage_offset =  0;

   gpr_$open_bitmap_file(access,pathname_of_bitmap, pathname_len,version,size,
                       groups, header, attribs, disk_bitmap, created, status);
   check("opening an external file bitmap");
   printf("allocated_size = %hd\n", header[0].allocated_size);
}
/****************************************************************************/
void draw_figure_in_file_bitmap()  /*Draw a line in the file bitmap.*/
/*You can use the GPR line calls to draw lines in a plane oriented bitmap;
  however, these calls won't work in a pixel oriented bitmap.*/
```

```
{
   gpr_$set_bitmap(disk_bitmap, status);
   gpr_$set_draw_value(draw_value, status);
   gpr_$line(150, 150, status);
}
/*****************************************************************************/
void create_bitmap_file_color_map()
{
        short int    counter;
 static color_range  blue_component = 0, red_component = 255, green_component;
        gpr_$pixel_value_t  color_map[n_of_colors];

   color_map[0] = 0;  /* Keep slot 0 black */
   for (counter = 1; counter <= 15; counter++)
      {green_component = counter*16;
       color_map[counter] = ((red_component < 16) + (green_component < 8) +
                             (blue_component));
      }
/*We now establish the array as the color map for the external file bitmap.*/
   gpr_$set_bitmap_file_color_map(disk_bitmap,(long)0,(short)n_of_colors,
                                  color_map,status);
   check("setting the bitmap file color map");
}
/*****************************************************************************/
void display_the_bitmap()
{
       gpr_$pixel_value_t color_map[256];
static gpr_$window_t      source_window =  {{0,0},{x_size_of_bitmap,
                                                   y_size_of_bitmap}};
static gpr_$position_t    dest_origin   =  {400,400};

   access_external_bitmap(gpr_$readonly);

/*Load the external file color map into array variable color_map.*/
   gpr_$inq_bitmap_file_color_map(disk_bitmap, (long)0, (short)n_of_colors,
                                  color_map, status);
   check("inquiring about bitmap file color map");

/*Load the colors from array variable color_map into the system color map.*/
   gpr_$set_color_map((long)0, (short)n_of_colors, color_map, status);
   check("setting color map");

/*BLT the external file bitmap to display memory*/
   gpr_$set_bitmap(display_bitmap, status);
   gpr_$pixel_blt(disk_bitmap, source_window, dest_origin, status);
   check("blting from disk to display");
}
/*****************************************************************************/
main()
{
   printf("Do you want to create a bitmap file or display a bitmap file?\n");
   printf("(Enter ´c´ or ´d´) -- ");
   scanf("%c", &choice);
   if (choice == ´c´)
     printf("Be patient; it will take some time to create the bitmap.\n");
   printf("What is the pathname of the file. -- ");
   scanf("%s", pathname_of_bitmap);
   init(gpr_$borrow);

   if (choice == 'c')
     { gpr_$allocate_attribute_block(attribs,status);
       access_external_bitmap(gpr_$create);
       create_bitmap_file_color_map();
       draw_figure_in_file_bitmap();
     }
   else
     { display_the_bitmap();
       pause(5.0);
     }
/*Terminate the graphics package.*/
   gpr_$terminate(false, status);
}
```

A Program That Demonstrates 24-Plane True-Color Programming (From Chapter 6)

```
/* Name of Program -- lines_in_true_color */
/*This program demonstrates true-color programming.  It draws 1020 lines, each
 line having a different color.  Notice that we never call gpr_$set_color_map.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

#define  left_end_of_line     0
#define  right_end_of_line 1000

 gpr_$pixel_value_t      color_total;
 short     r, y_coord_of_line, red_component, green_component, blue_component;

#include "my_include_file.c" /*contains the init, check, and pause routines.*/
main()
{
   init(gpr_$borrow_rgb);
   green_component =  255;

/*This program draws 1020 lines, each line a different color.*/
   for (r =  1; r <= 4; r++)
      { red_component =  (r * 64) - 1;  /*red_component = 63,127,191,255*/
        for (blue_component = 1; blue_component <= 255; blue_component++)
           {
/*Create a unique draw value.*/
              color_total = (red_component << 16) + (green_component << 8) +
                             blue_component;
              gpr_$set_draw_value(color_total, status);

/*Draw the line with this draw value.*/
              y_coord_of_line += 1;
              gpr_$move(left_end_of_line, y_coord_of_line, status);
              gpr_$line(right_end_of_line, y_coord_of_line, status);
           }
      }

/*Pause 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Demonstrates Color Zoom (From Chapter 6)

```
/* Name of Program -- zooming */
/*This program demonstrates how to magnify (zoom) the display bitmap.
 It uses the gpr_$color_zoom call.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"

 short int        radius = 50, x_mag_factor, y_mag_factor;
 gpr_$position_t  center =  {100,100};

#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
main()
{
   printf("Enter the magnification factor in the X dimension -- ");
   scanf("%hd", &x_mag_factor);
   printf("Enter the magnification factor in the Y dimension -- ");
   scanf("%hd", &y_mag_factor);

   init(gpr_$borrow);

   gpr_$circle_filled(center, radius, status);
   pause(1.0);

   if (x_mag_factor > display_characteristics.x_zoom_max)
     printf("X magnification factor is too high for this node.\n");
   else if (y_mag_factor > display_characteristics.y_zoom_max)
     printf("Y magnification factor is too high for this node.\n");
   else if ((y_mag_factor == 1) && (x_mag_factor != 1))
     {printf("You cannot set the Y mag factor to 1 if the \n");
      printf("X mag factor is greater than 1.\n");
     }
   else
     {gpr_$color_zoom(x_mag_factor, y_mag_factor, status);
      check("zooming");
      pause(5.0);
     }

/*Terminate.*/
   gpr_$terminate(false, status);
}
```

A Program That Enables Two Event Types and Waits for One of Them To Occur (From Chapter 7)

```
/* Name of Program -- simple_events */
/*This program enables two kinds of events.  It then waits for one of those
 events to occur and reports which one occurred.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "/sys/ins/kbd.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

#define SETSIZE    (short)256
/****************************************************************************/
void enable_events()  /*This function enables the three mouse buttons and
                        all lowercase letters.*/
{
 gpr_$event_t   event_type;
 gpr_$keyset_t  mouse_buttons, key_set;
 char           lowercase_letter;

    event_type =  gpr_$keystroke;
    lib_$init_set(key_set, SETSIZE);
    for (lowercase_letter = ′a′; lowercase_letter <= ′z′; lowercase_letter++)
       lib_$add_to_set(key_set, SETSIZE, (short)lowercase_letter);
    gpr_$enable_input(event_type, key_set, status);
    check("enabling input 2");

    event_type =  gpr_$buttons;
    lib_$init_set(mouse_buttons, SETSIZE);
    lib_$add_to_set(mouse_buttons, SETSIZE, KBD_$M1D);
    lib_$add_to_set(mouse_buttons, SETSIZE, KBD_$M2D);
    lib_$add_to_set(mouse_buttons, SETSIZE, KBD_$M3D);
    gpr_$enable_input(event_type, mouse_buttons, status);
    check("enabling input 1");
}
/****************************************************************************/
main()
{
 gpr_$event_t     event_type;
 char             event_data;
 gpr_$position_t  position;

    init(gpr_$borrow);
    enable_events();

/*Wait for the user to type a lowercase letter or to depress a mouse button.*/
    gpr_$event_wait(event_type, event_data, position, status);
    check("waiting for an event");

    gpr_$terminate(false, status);

/*Report on the event that occurred.*/
    if (event_type == gpr_$keystroke)
      printf("You typed the letter %c\n",event_data);
    else
      printf("You hit a mouse button.\n");
}
```

A Program That Leaves a Line Trail (From Chapter 7)

```
/* Name of Program -- locator_events */
/*This program enables and reports locator events.  It uses the
  gpr_$enable_input and gpr_$event_wait routines.  The program allows you to
  sketch lines.  To begin, depress the left-most mouse button, then move the
  cursor.  To end, release the left-most mouse button.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "/sys/ins/kbd.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/
#define SETSIZE    (short)256
gpr_$event_t      event_type;
gpr_$keyset_t     mouse_buttons, key_set;
char              event_data;
gpr_$position_t  position;
/****************************************************************************/
void enable_events()
/*This procedure enables the tracking of the mouse and the left-most
  mouse button.*/
{
    event_type = gpr_$locator_update;
    gpr_$enable_input(event_type, key_set, status);

    event_type =  gpr_$buttons;
    lib_$init_set(mouse_buttons, SETSIZE);
    lib_$add_to_set(mouse_buttons, SETSIZE, KBD_$M1D);
    lib_$add_to_set(mouse_buttons, SETSIZE, KBD_$M1U);
    gpr_$enable_input(event_type, mouse_buttons, status);
}
/****************************************************************************/
main()
{
static short int   sketch_started = 0;

    init(gpr_$borrow);
    enable_events();

    do
      {gpr_$event_wait(event_type, event_data, position, status);
/* If user presses down M1 */
       if ((event_type == gpr_$buttons) && (event_data == KBD_$M1D))
         {gpr_$move(position.x_coord, position.y_coord, status);
          sketch_started = 1;
         }
/* If user moves mouse while M1 is depressed */
       else if ((event_type == gpr_$locator_update) && (sketch_started))
         gpr_$line(position.x_coord, position.y_coord, status);
/* If user releases M1 */
       else if ((event_type == gpr_$buttons) && (event_data == KBD_$M1U))
         break;
      }
    while (1);

    pause(5.0);/*Pause 5 seconds; then terminate.*/
    gpr_$terminate(false, status);
}
```

A Program That Monitors Input Events in Multiple Windows (From Chapter 7)

```
/* Name of Program -- gpr_in_multiple_windows */
/*This program creates a GPR program that runs in one shell and two windows.
 It primarily demonstrates the pad_$create_window, gpr_$event_wait, and
 gpr_$set_window_id calls (though several other GPR and PAD calls are also
 used).  In this program, all output from printf statements will be printed
 in stdout (which is probably the shell that you invoke the program from).
 We create two windows by calling pad_$create_window twice.  We also
 call gpr_$init twice, once for each window, so that direct mode graphics can
 run in both windows.  The user will move the cursor into one of the two
 windows.  The program will draw a line into the chosen window.  We used the
 gpr_$set_window_id call to mark the two windows with a unique identification
 character.  When the user moves the cursor into one of the two windows, the
 gpr_$event_wait routine will return the identification character of the
 appropriate window.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "/sys/ins/pad.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

 char                unobs;
 char                delete_display  =  false;
 gpr_$event_t        event_type;
 gpr_$keyset_t       key_set;
 char                event_data;
 gpr_$position_t     position;
 gpr_$bitmap_desc_t  display_bm1, display_bm2;
 static char         window_id1 = '1', window_id2 = '2';
/****************************************************************************/
void assign_some_stuff_to_bitmaps(graphics_strid)
ios_$id_t  graphics_strid;
{
   gpr_$enable_input(gpr_$entered_window, key_set, status);
   pad_$set_auto_close(graphics_strid,1,true,status);
   gpr_$set_obscured_opt(gpr_$pop_if_obs,status);
   gpr_$set_auto_refresh(true,status);
}
/****************************************************************************/
void create_two_windows() /*This procedure creates and initializes two
                            windows for the display of graphics.*/
{
 static gpr_$display_mode_t  mode = gpr_$direct;
        ios_$id_t            graphics_strid1, graphics_strid2;
        pad_$window_desc_t   window1, window2;
        gpr_$offset_t        display_bm1_size, display_bm2_size;
 static gpr_$rgb_plane_t     hi_plane =  1;
/*Create two windows.*/
   window1.top    =  0;     window1.left    =  0;
   window1.width =  400;    window1.height =  400;
   window2.top    =  500;   window2.left    =  500;
   window2.width =  200;    window2.height =  200;
   pad_$create_window("",0,pad_$transcript,1,window1,graphics_strid1,status);
   pad_$create_window("",0,pad_$transcript,1,window2,graphics_strid2,status);

/*Initialize graphics in both windows and give each window a distinct char
id.*/
```

```
   display_bm1_size.x_size =  400;    display_bm1_size.y_size =  400;
   display_bm2_size.x_size =  200;    display_bm2_size.y_size =  200;
   gpr_$init(mode,graphics_strid1,display_bm1_size,hi_plane,dis-
play_bm1,status);
   gpr_$set_window_id(window_id1, status);
   check("setting first window id");
   assign_some_stuff_to_bitmaps(graphics_strid1);

   gpr_$init(mode,graphics_strid2,display_bm2_size,
             hi_plane,display_bm2,status);
   gpr_$set_window_id(window_id2, status);
   check("setting second window id");
   assign_some_stuff_to_bitmaps(graphics_strid2);
}
/*****************************************************************************/
void draw_a_line_in_window(window_that_cursor_is_in)
char window_that_cursor_is_in;
{
   if (window_that_cursor_is_in == window_id1)
     gpr_$set_bitmap(display_bm1, status);
   else
     gpr_$set_bitmap(display_bm2, status);

   gpr_$acquire_display(status);
   gpr_$line(100, 100, status);
   gpr_$release_display(status);
}
/*****************************************************************************/
main()
{
   create_two_windows();
   printf("Two empty windows just appeared. Move the cursor into one\n");
   printf("of them. The program will draw a line in the selected window.\n");

   gpr_$event_wait(event_type, event_data, position, status);
   draw_a_line_in_window(event_data);

/*Pause 2 seconds; then terminate.*/
   pause(2.0);
   gpr_$terminate(false, status);
}
```

A Program That Creates a Nondefault Cursor Pattern (From Chapter 8)

```
/* Name of Program -- nondefault_cursor_example */
/*This program creates a nondefault cursor pattern.  It demonstrates the
 gpr_$set_cursor_position, gpr_$set_cursor_pattern, and gpr_$set_cursor_active
 routines.  The program draws a pattern in a main memory bitmap, and then
 establishes this bitmap as the current cursor pattern.  It then activates
 this cursor and gives it a starting position of 200,200.  The cursor will
 automatically follow the motion of the locator device.  When you move the
 locator device, you'll see the default nonblinking cursor pattern.  When
 you stop moving the locator device, you'll see the nondefault cursor pattern
 (which blinks).
*/

#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

gpr_$bitmap_desc_t     cursor_bitmap_descriptor;
/****************************************************************************/

void create_cursor_pattern()
{
 static gpr_$offset_t          size_of_bitmap        =  {16,16};
        gpr_$attribute_desc_t  attributes_descriptor;

/*Allocate a small main memory bitmap.*/
   gpr_$allocate_attribute_block(attributes_descriptor, status);
   gpr_$allocate_bitmap(size_of_bitmap, hi_plane, attributes_descriptor,
                        cursor_bitmap_descriptor, status);

/*Draw an arrow pattern inside the main bitmap.*/
   gpr_$set_bitmap(cursor_bitmap_descriptor, status);
   gpr_$move(8, 15, status);
   gpr_$line(8, 0, status);
   gpr_$line(1, 7, status);
   gpr_$line(15,7, status);
   gpr_$line(8, 0, status);
}
/****************************************************************************/
void activate_cursor()
{
 static unsigned char    cursor_active   =  true;
 static gpr_$position_t  cursor_position =  {200,200};
 static gpr_$position_t  cursor_origin   =  {8,0};
/*Make the main memory bitmap into the current cursor pattern.*/
   gpr_$set_cursor_pattern(cursor_bitmap_descriptor, status);

/*Establish 200,200 as the starting current cursor position.*/
   gpr_$set_cursor_position(cursor_position, status);

/*Make the cursor visible.*/
   gpr_$set_cursor_active(cursor_active, status);

/*Sets position 8,0 as the cursor origin.*/
   gpr_$set_cursor_origin(cursor_origin, status);
}
```

```
/****************************************************************************/
main()
{
   init(gpr_$borrow);

   create_cursor_pattern();
   activate_cursor();

/*Let the user move the cursor around for 10 seconds, then terminate.*/
   pause(10.0);
   gpr_$terminate(false, status);
}
```

A Program That Tracks a Nondefault Cursor as the Mouse Moves (From Chapter 8)

```
/* Name of Program -- tracking_the_cursor */
/* This program shows how you can display a nondefault cursor wherever the
   mouse tracks to.  It uses a combination of cursor and event calls to
   demonstrate this feature.  Enter <CTRL-Q> to exit from the program.*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

gpr_$bitmap_desc_t  cursor_bitmap_descriptor;
/****************************************************************************/
void create_cursor_pattern()
{
static gpr_$offset_t           size_of_bitmap = {16,16};
       gpr_$attribute_desc_t   attributes_descriptor;
static gpr_$position_t         center         = {8,8};

   gpr_$allocate_attribute_block(attributes_descriptor, status);
   gpr_$allocate_bitmap(size_of_bitmap, hi_plane, attributes_descriptor,
                        cursor_bitmap_descriptor, status);

/*Draw the cursor pattern (a small filled circle) inside the main bitmap.*/
   gpr_$set_bitmap(cursor_bitmap_descriptor, status);
   gpr_$circle_filled(center, (short)6, status);
}
/****************************************************************************/
void initialize_cursor()
{
static gpr_$position_t   cursor_origin = {8,1};

   gpr_$set_cursor_active(false, status);
   gpr_$set_cursor_origin(cursor_origin, status);
   gpr_$set_cursor_pattern(cursor_bitmap_descriptor, status);
   gpr_$set_cursor_active(true, status);
}
/****************************************************************************/
main()
{
gpr_$event_t        event_type;
gpr_$position_t     mouse_position, cursor_position;
char                event_data;
gpr_$keyset_t       key_set;

   init(gpr_$borrow);

   create_cursor_pattern();
   initialize_cursor();

/*Activate the locator.*/
   event_type = gpr_$locator;
   gpr_$enable_input (event_type, key_set, status);

   while(1)
     {gpr_$event_wait(event_type, event_data, mouse_position, status);
      if (event_type == gpr_$locator)
        {cursor_position = mouse_position;
         gpr_$set_cursor_position(cursor_position, status);
        cursor_position.x_coord = cursor_position.x_coord + 25;
        gpr_$circle(cursor_position, 1, status);
        check("drawing circle");
       }
    }

/*Terminate the graphics package.*/
   gpr_$terminate(false, status);
}
```

A Program That Demonstrates Clipping (From Chapter 9)

```
/* Name of Program -- clipping */

/* This program demonstrates how to create and activate a clipping window,
   and how a clipping window affects a drawing.  This program creates and
   activates a 200x200 clipping window.  Inside the clipping window,
   we draw a filled circle.  The parts of the circle outside the clipping
   window are not drawn.  If you experiment with the radius size, this
   effect will become more apparent.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*contains the init, check, and pause routines.*/

main()
{
 static gpr_$window_t    clip_window;
 static gpr_$position_t  center      =  {500, 500};
        short int        radius      = 116;

   init(gpr_$borrow);

   clip_window.window_base.x_coord =  400;
   clip_window.window_base.y_coord =  400;
   clip_window.window_size.x_size  =  200;
   clip_window.window_size.y_size  =  200;
   gpr_$set_clip_window(clip_window, status);  /* Create clipping window. */
   gpr_$set_clipping_active(true, status);     /* Activate clipping window. */

   gpr_$circle_filled(center, radius, status); /* Draw filled circle */

/*Pause 5 seconds; then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Sets a Plane Mask (From Chapter 9)

```
/* Name of Program -- masking */
/* This program creates a plane mask on a color node.  It demonstrates the
  gpr_$set_plane_mask_32 routine.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*contains the init, check, and pause routines.*/

main()
{
 static gpr_$position_t   center = {500, 500};
 static short int         radius =  200;
        long int          mask;

   init(gpr_$borrow);

/*We set the fill value to 15, which corresponds to each bit in planes 0
 through 3 being set to a 1.*/
   gpr_$set_fill_value(15L, status);

/*Let us put a plane mask on planes 0, 1, and 2, meaning that they are
  the only planes that can be altered.  */
   mask = (1L < 0) | (1L < 1) | (1L < 2);
   gpr_$set_plane_mask_32(mask, status);

/*Therefore, although the fill value is 15 (1111), the effective fill value is

  actually only 7 (0111). The circle will be filled with whatever color
  corresponds to index 7.*/
   gpr_$circle_filled(center, radius, status); /* Draw filled circle */

/* Pause 5 seconds; then terminate. */
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Direct Mode Program That Acquires and Releases the Display (From Chapter 10)

```
/*program acquire_release; */
/*This program acquires and releases the display 5000 times.  It demonstrates
 the gpr_$acquire_display and gpr_$release_display routines.  To prove that
 the display is really being released, move to another window at some point
 during this program's execution and try to type something.  If the display
 is really being released, the typed letters will appear.  This program will
 only run in an unobscured window.
*/

#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c"/*Contains the init, check, and pause routines.*/

main()
{
short int         x, y, count;
gpr_$position_t   center;
#define radius 4

  init(gpr_$direct);

  for (count =  1; count <= 5000; count++)
     {
/*We do not need to have the display acquired to do these calculations.*/
    x = (count * count) % display_characteristics.x_window_size;
    y = (count * x)     % display_characteristics.y_window_size;

    gpr_$acquire_display(status);           /*Acquire the display.*/
         gpr_$line(x, y, status);                   /*Draw a line.*/
         check("drawing line");
         center.x_coord = x; center.y_coord = y;
         gpr_$circle_filled(center, radius, status); /*Draw circle.*/
         check("drawing circle");
    gpr_$release_display(status);            /*Release the display.*/
  }

 /*Terminate program.*/
   gpr_$terminate(false, status);
 }
```

A Direct Mode Program That Draws in the Unobscured Regions of a Window (From Chapter 10)

```
/* Name of Program -- writing_in_obscured_windows */
/*This program draws lines in the nonobscured parts of the window that the
 program runs in.  It demonstrates the gpr_$set_obscured_opt and
 gpr_inq_vis_list routines.  After starting this routine, you should obscure
 portions of the window by growing or popping other windows on top of the
 window running the program.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

#define max_slots 32   /*Set an arbitrary limit of 32 rectangles.*/

main()
{
  short                  old_x, old_y, new_x, new_y, count, n;
  gpr_$obscured_opt_t  what_to_do_if_obscured = gpr_$ok_if_obs;
  short                  slots_available =  max_slots, slots_total;
  gpr_$window_t          vis_list[max_slots];

  init(gpr_$direct);
  gpr_$set_clipping_active(true, status);

  gpr_$set_obscured_opt(what_to_do_if_obscured, status);

  for (count =  1; count <=  3000; count ++)  /*This loop draws 3000 lines.*/
     {new_x =  (count * count) % 600;
      new_y =  (count * new_x) % 610;

/*Find the visible slots every time display is acquired.*/
      gpr_$acquire_display(status);
      gpr_$inq_vis_list(slots_available, slots_total, vis_list, status);
      if (!(count % 100))
        printf("%hd\n", slots_total);

/*This loop draws a line from the old coordinates to the new coordinates
 slots_total times.  There is no harm (other than lost time) in drawing
 a line over the same coordinates.*/
      for (n = 0; n < slots_total; n++)
         {gpr_$set_clip_window(vis_list[n], status);
          gpr_$move(old_x, old_y, status);
          gpr_$line(new_x, new_y, status);
         }
      old_x =  new_x;  old_y =  new_y;
      gpr_$release_display(status);
     }

/*Terminate program.*/
   gpr_$terminate(false, status);
}
```

A Direct Mode Program That Demonstrates Window Refresh (From Chapter 10)

```
/* Name of Program -- refresh_example */
/*This program contains a refresh procedure.  It demonstrates the
 gpr_$set_refresh_entry routine.  The program draws a simple design.  You
 should obscure a portion of the window the program is running in and then
 pop the window so that the program will require a refresh.  The refresh
 will redraw the simple design.  To leave the program, type the letter Q.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

extern void draw();
gpr_$rwin_pr_t  pf = draw;

main()
{
 unsigned  char   unobscured;
 gpr_$position_t  ev_pos;
 gpr_$event_t     ev_type;
 char             ev_char;
 gpr_$keyset_t    keys;

    init(gpr_$direct);

/*Draw a picture.*/
    draw(true, false);

/*Establish the refresh function. Once established, if the window needs to be
  refreshed as the result of a pop, GPR will automatically call the draw
  function.
*/
    gpr_$set_refresh_entry (pf,(long)0, status);
    check("setting the refresh entry procedure.");

/*Enter the letter Q to exit the program.*/
    lib_$init_set(keys, 256);
    lib_$add_to_set(keys, 256, 'q');
    lib_$add_to_set(keys, 256, 'Q');
    gpr_$enable_input (gpr_$keystroke, keys, status);
    gpr_$event_wait (ev_type, ev_char, ev_pos, status);

/*Terminate.*/
    gpr_$terminate(false, status);
}
```

A Direct Mode Program That Accepts High-Level Input From The Input Pad (From Chapter 10)

```
/* Name of Program -- high_level_input */
/*This program prompts the user for the coordinates of a line, and then draws
  a line based on the user´s input.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c"/*Contains the init, check, and pause routines.*/

main()
{
short int   x_start, y_start, x_stop, y_stop;

   init(gpr_$direct);
   gpr_$set_cursor_active(true,status);

   printf("Enter x coord of line start: "); scanf("%hd", &x_start);
   printf("Enter y coord of line start: "); scanf("%hd", &y_start);
   printf("Enter x coord of line stop:  "); scanf("%hd", &x_stop);
   printf("Enter y coord of line stop:  "); scanf("%hd", &y_stop);

   gpr_$acquire_display(status);
   gpr_$move(x_start, y_start, status);
   gpr_$line(x_stop, y_stop, status);
   gpr_$release_display(status);

/*Pause 5 seconds, then terminate.*/
   pause(5.0);
   gpr_$terminate(false,status);
}
```

A Program That Sets the Raster Operation for a Draw Operation (From Chapter 11)

```
/* Name of Program -- intersecting_lines */
/*This program draws two intersecting lines to demonstrate raster operations.
 It uses the gpr_$set_raster_op and gpr_$inq_raster_ops routines.
 Assuming that the default color map is loaded, the program will draw one
 red line and one green line.  Because we set the raster operation to 7,
 the intersection of the two lines will be drawn in blue.  This program should
 be run on a color node.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

main()
{
 gpr_$raster_op_t        raster_op;
 gpr_$raster_op_array_t  raster_op_array;
 gpr_$rgb_plane_t        plane;

   init(gpr_$borrow);

   gpr_$set_draw_width(20, status);
   gpr_$set_draw_value((long)1, status);  /*Draws a red line.*/
   gpr_$move(0, 200, status);
   gpr_$line(400, 200, status);

   raster_op = 6;
   for (plane = 0; plane <= hi_plane; plane++)
      gpr_$set_raster_op(plane, raster_op, status);

   gpr_$set_draw_value((long)2, status);
   gpr_$move(200, 0, status);
   gpr_$line(200, 400, status);   /*Draws a blue line. */

   gpr_$inq_raster_ops(raster_op_array, status);
   for (plane = 0; plane <= hi_plane; plane++)
      printf("Raster op for plane %d = %d\n", plane, raster_op_array[plane]);

 /*Pause 5 seconds; then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
 }
```

A Program That Sets the Raster Operation for a BLT (From Chapter 11)

```
/* Name of Program -- raster_ops_in_blts */
/*This program demonstrates how you can use raster operations with BLTs.
 It uses the gpr_$set_raster_op routine.  Assuming that the default color
 map is loaded, the program will draw one blue rectangle and one yellow
 rectangle.  Then it will BLT the blue rectangle to the area covered by
 the yellow rectangle.  Because the raster operation is 1 for each plane,
 the BLT will produce a red rectangle.  This program should be run on a
 color node.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

#define  left_of_source      0
#define  left_of_dest      400
#define  top_of_source       0
#define  top_of_dest       400
#define  size_x_of_source  200
#define  size_x_of_dest    200
#define  size_y_of_source  200
#define  size_y_of_dest    200

 gpr_$raster_op_t  raster_op;
 gpr_$rgb_plane_t  plane;
 gpr_$window_t     source_window;
 gpr_$position_t   destination_origin = {left_of_dest, top_of_dest};
 gpr_$window_t     rectangle1      =  {{left_of_source, top_of_source},
                                       {size_x_of_source, size_y_of_source}};
 gpr_$window_t     rectangle2      =  {{left_of_dest, top_of_dest},
                                       {size_x_of_dest, size_y_of_dest}};
main()
{
   init(gpr_$borrow);

   gpr_$set_fill_value((long)3, status);
   gpr_$rectangle(rectangle1, status);  /*Draw a blue rectangle.*/
   gpr_$set_fill_value((long)5, status);
   gpr_$rectangle(rectangle2, status);  /*Draw a yellow rectangle.*/
   pause(2.0);

   source_window =  rectangle1;

   raster_op =  1;  /*Destinaton =  Source AND Destination*/
   for (plane = 0; plane <= hi_plane; plane++)
      gpr_$set_raster_op(plane, raster_op, status);

/*BLT the blue rectangle to the area covered by the yellow rectangle.*/
   gpr_$pixel_blt(display_bitmap, source_window, destination_origin, status);

/*Pause 5 seconds; then terminate.*/
   pause(5.0);
   gpr_$terminate(false, status);
}
```

A Program That Restricts Raster Operations to Particular Categories (From Chapter 11)

```
/* Name of Program -- raster_op_categories */
/*This program demonstrates how to set different raster operations for
 three different categories of GPR routines.  It demonstrates the
 gpr_$raster_op_prim_set and gpr_$set_raster_op routines.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "my_include_file.c" /*Contains the init, check, and pause routines.*/

main()
{
        gpr_$raster_op_t            raster_op;
 static gpr_$rgb_plane_t            plane_to_set_rop_on = 0;
        int                         prim_set;

   init(gpr_$borrow);

/*Set the raster operation to 5 for all line drawing routines. */
   prim_set =  1 < (short)gpr_$rop_line;
   raster_op =  5;
   gpr_$raster_op_prim_set(prim_set, status);
   gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status);

/*Set the raster operation to 7 for all fill routines. */
   prim_set =  1 < (short)gpr_$rop_fill;
   raster_op =  7;
   gpr_$raster_op_prim_set(prim_set, status);
   gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status);

/*Set the raster operation to 13 for all blt routines. */
   prim_set =  1 < (short)gpr_$rop_blt;
   raster_op =  13;
   gpr_$raster_op_prim_set(prim_set, status);
   gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status);

   gpr_$terminate(false, status);
}
```

A Program That Sets the Decomposition Technique (From Appendix E)

```
/*Name of Program -- set_the_decomposition_technique */
/*This program sets the decomposition technique to gpr_$non_overlapping_tris
  on certain node types.  It demonstrates the gpr_$pgon_decomp_technique
  routine.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "my_include_file.c"

gpr_$rop_prim_set_elems_t  prim_set;/*The set of primitives that raster*/
                                     /*operations will affect.*/
short int                  setsize = 16;

main()
{
  init(gpr_$borrow);

/* Set the decomposition technique on certain node types.*/
  switch (display_characteristics.controller_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("establish the pgon decomp technique");
        }

/*Establish the set of raster operations for lines and fills.*/
    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);
    check("establishing the raster op prim set");

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

/*Terminate the program.*/
    gpr_$terminate(false,status);
}
```

A Program That Fills a Polygon Using the Triangle Technique (From Appendix E)

```
/* Program name is -- triangle_technique */
/*This program demonstrates the triangle technique.  It illustrates the
  gpr_$set_triangle_fill_criteria and gpr_$multitriangle routines.
*/
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gpr.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "my_include_file.c"

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;
short int                       setsize = 16;

main()
{
    init(gpr_$direct);

/* Set the decomposition technique on certain node types.*/
  switch (display_characteristics.controller_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("establish the pgon decomp technique");
        }

    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("establishing the raster op prim set");

    window.window_base.x_coord = 200;
    window.window_base.y_coord = 200;
    window.window_size.x_size = 1;
    window.window_size.y_size = 1;

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

    gpr_$read_pixels(window, pixel_array, 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);

    list_size = 30;
    gpr_$close_return_pgon_tri(list_size, t_list, n_triangles, status);

/* Draw the triangles with a parity fill. */
    gpr_$multitriangle(t_list, n_triangles, status);

/* Keep image displayed on screen for 5 seconds, then clear screen. */
    pause(5.0);
    gpr_$clear(pixel_array[0], status);

    winding_set.wind_type = gpr_$nonzero;
    gpr_$set_triangle_fill_criteria(winding_set,status);

/* Draw the triangles with a nonzero fill. */
    gpr_$multitriangle(t_list, n_triangles, status);

/* Keep image displayed on screen for 5 seconds, then clear screen. */
    pause(5.0);
    gpr_$clear(pixel_array[0], status);

    winding_set.wind_type = gpr_$specific;
    winding_set.winding_no = 2;
    gpr_$set_triangle_fill_criteria(winding_set,status);

/* Draw the triangles with a specific winding number fill. */
    gpr_$multitriangle(t_list, n_triangles, status);

/*Pause 5 seconds, release display, terminate program.*/
    pause(5.0);
    gpr_$release_display(status);
    gpr_$terminate(false, status);
}
```

Appendix C

FORTRAN Programs

This appendix contains all the programming examples used in the manual translated into FORTRAN.

NOTE: See the *Programming With General System Calls* manual for details on simulating Pascal data types in FORTRAN.

The File That Must Be Bound With All Sample FORTRAN Programs

```
C*************************************************************
      subroutine CHECK(messagex)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/error.ins.ftn'
%include 'my_common_block.ftn'

      character*(*)  messagex

      if (status .ne. 0) then
         call error_$print (status)
         print 10, messagex
      endif
10    format('error occurred while ', A)
      END
C*************************************************************
      subroutine PAUSE(t)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/time.ins.ftn'
      integer*4     t

      integer*2  time(3)
      time(1) = 0
      time(2) = 4 * t
      time(3) = 0

      call time_$wait (time_$relative, time, status)
      call check('In Procedure PAUSE.')
      END
```

```
C*************************************************************
      subroutine INIT(mode)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      integer*2  mode
      integer*2  unit, disp_len, disp_len_returned
      logical    unobscured

      unit = 1
      disp_len = 62
      call gpr_$inq_disp_characteristics(mode, unit, disp_len,
     +       display_characteristics, disp_len_returned, status)
      call check('inquiring about the display characteristics')

      display_bitmap_size(1) = display_characteristics(5)
      display_bitmap_size(2) = display_characteristics(6)
      hi_plane               = display_characteristics(15) - 1
      print 20, hi_plane
20    FORMAT ('hi_plane = ', I2)

      call gpr_$init(mode, unit, display_bitmap_size, hi_plane,
     +               display_bitmap, status)
      call check('calling gpr_$init')
      END
```

The Common Block That Must Be Included In Every Sample Program (From Chapter 2)

```
      integer*4  status
      integer*4  display_bitmap
      integer*2  display_bitmap_size(2)
      integer*2  display_characteristics(31)
      integer*2  hi_plane
      common /group1/ status, display_bitmap, display_bitmap_size,
     +                display_characteristics, hi_plane
```

A Simple GPR Program To Get Started (From Chapter 2)

```
      PROGRAM getting_started_with_gpr
C This program draws a line in borrow mode from the upper left corner of
C the screen to the lower right corner of the score.

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

      call init(gpr_$borrow)

      call gpr_$line(display_characteristics(5),
     +               display_characteristics(6),
     +               status)
      call check('drawing line')

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Draw a Simple Line (From Chapter 3)

```
      Program simple_lines
C This program demonstrates how to use the gpr_$line call,
C and how to change the current position with the gpr_$move call.

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

      call init(gpr_$borrow)

C Draw a line from the coordinate origin (0,0) to the endpoint (300,300).
      call  gpr_$line(int2(300), int2(300), status)

C The current position is now set at (300,300).  Use the gpr_$move call to
C change the current position to (100,500).
      call  gpr_$move(int2(100), int2(500), status)

C Draw a line from the current position (100,500) to the endpoint (500,500).
      call gpr_$line(int2(500), int2(500), status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Draw Connected Lines (From Chapter 3)

```
      Program disconnected_lines
C This program draws several connected lines.  It demonstrates the
C gpr_$polyline call.

%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/gpr.ins.ftn´
%include ´my_common_block.ftn´

      integer*2  array_of_x_coordinates(3), array_of_y_coordinates(3)
      integer*2  number_of_end_points
      data       array_of_x_coordinates/200, 300, 400/
      data       array_of_y_coordinates/300, 400, 200/
      data       number_of_end_points/3/

      call init(gpr_$borrow)

C Establish the current position at (0,300).  If we do not call gpr_$move, the
C current position will be (0,0).}
      call gpr_$move(int2(0), int2(300), status)

C Draw three connected lines.  Notice that each endpoint becomes the
startpoint
C for the next line.
      call gpr_$polyline(array_of_x_coordinates, array_of_y_coordinates,
     +                   number_of_end_points, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Draw Disconnected Lines (From Chapter 3)

```
      Program disconnected_lines
C This program draws three disconnected lines.  It demonstrates the
C gpr_$multiline call.

%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/gpr.ins.ftn´
%include ´my_common_block.ftn´

      integer*2  array_of_x_coordinates(6), array_of_y_coordinates(6)
      integer*2  number_of_points
      data       array_of_x_coordinates/100,400,100,400,100,400/
      data       array_of_y_coordinates/100,100,200,200,300,300/
      data       number_of_points/6/

      call init(gpr_$borrow)

C Draw three disconnected lines.  Line1 runs from (100,100) to (400, 100);
C Line2 runs from (100,200) to (400,200); Line3 runs from (100,300) to
C (400,300).  Notice that we don´t have to use gpr_$move to establish the
C current position.}
      call gpr_$multiline(array_of_x_coordinates,
     +                    array_of_y_coordinates,
     +                    number_of_points, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Draw Two Circles (From Chapter 3)

```
      Program circles_example
C This program draws two circles.  It demonstrates the gpr_$circle and
C gpr_$circle_filled calls.

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

      integer*2  center(2)
      integer*2  radius

      call init(gpr_$borrow)

C Draw an unfilled circle.
      center(1) = 200
      center(2) = 200
      radius    = 100
      call gpr_$circle(center, radius, status)

C Draw a filled circle.
      center(1) = 400
      center(2) = 400
      radius    = 100
      call gpr_$circle_filled(center, radius, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(false, status)
      end
```

A Program To Draw Two Arcs (From Chapter 3)

```
      Program arcs_example
C This program draws two arcs.  It demonstrates the gpr_$arc_c2p and
C gpr_$arc_3p routines.

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

      integer*2  center(2), p2(2), point2(2), point3(2)
      integer*2  direction
      integer*2  option

      call init(gpr_$borrow)

C           *** Demonstration of gpr_$arc_3p ***
C The gpr_$arc_3p call draws an arc through any three noncolinear points.
C The three points are the current position, point2, and point3.
C The system draws the arc from the current position through
C point2 and completes the arc at point3.
      call gpr_$move(int2(200), int2(200), status) {set the current position.}
      point2(1) = 300
      point2(2) = 300
      point3(1) = 200
      point3(2) = 400
      call gpr_$arc_3p(point2, point3, status)  {draw the arc.}
      call check('drawing arc_3p')


C           ***Demonstration of gpr_$arc_c2p ***
C The gpr_$arc_c2p call draws an arc between two points.
C The radius of the arc is the distance from the current position to center.
C The system starts the arc at the current position and revolves it in a
C clockwise or counterclockwise direction.  The end point of the arc lies
C on an imaginary ray beginning at center and passing through position p2.
C The 'option' parameter is only meaningful if the current position
C is both the start and end point of the arc. In this case, 'option'
C tells the routine whether to draw a full circle or to draw nothing at all.
      call gpr_$move(int2(600), int2(400), status) {set the current position.}
      center(1) = 600
      center(2) = 600
      p2(1)     = 500
      p2(2)     = 600
      direction = gpr_$arc_ccw  {draw the arc counterclockwise}
      option    = gpr_$arc_draw_full  {ignored in this case.}
      call gpr_$arc_c2p(center, p2, direction, option, status)
      call check('drawing arc_c2p')

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(false, status)
      end
```

A Program To Draw a Spline (From Chapter 3)

```
      Program spline_example
C This program draws a cubic spline as a function of x.
C It demonstrates the gpr_$spline_cubic_x routine.

%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/gpr.ins.ftn´
%include ´my_common_block.ftn´

      integer*2   array_of_x_pts(5), array_of_y_pts(5), npoints
      data        array_of_x_pts/100,200,300,400,500/
      data        array_of_y_pts/ 20, 80,180,320,500/
      data        npoints/5/

      call init(gpr_$borrow)

C Draw spline as a function of x.  Starting position of spline will be
C the current position of [0,0]
      call gpr_$spline_cubic_x(array_of_x_pts, array_of_y_pts,
     +                         npoints, status)
      call check(´drawing spline´)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(false, status)
      end
```

A Program To Draw a Filled Triangle, Rectangle, and Trapezoid (From Chapter 3)

```
      Program triangle_rectangle_trapezoids
C This program draws a triangle, rectangle, and trapezoid.  It demonstrates
C the gpr_$triangle, gpr_$rectangle, and gpr_$trapezoid calls.

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

      integer*2 triangle_vertex1(2), triangle_vertex2(2)
      integer*2 triangle_vertex3(2)
      integer*2 rectangle(4)
      integer*2 trapezoid(6)

      call init(gpr_$borrow)

C Draw a filled triangle.
      triangle_vertex1(1) = 100
      triangle_vertex1(2) = 100
      triangle_vertex2(1) = 300
      triangle_vertex2(2) = 100
      triangle_vertex3(1) = 200
      triangle_vertex3(2) = 200
      call gpr_$triangle(triangle_vertex1, triangle_vertex2,
     +                   triangle_vertex3, status)

C Draw a filled rectangle.
      rectangle(1) = 100        {x coord of window base}
      rectangle(2) = 300        {y coord of window base}
      rectangle(3) = 200        {x size of window}
      rectangle(4) = 300        {y size of window}
      call gpr_$rectangle(rectangle, status)

C Draw a filled trapezoid.  In GPR, a trapezoid is a four-sided polygon
C with parallel bottom and top sides.
      trapezoid(1) = 300  {x coordinate of the top left point.}
      trapezoid(2) = 500  {x coordinate of the top right point.}
      trapezoid(3) = 200  {y coordinate of the top line segment.}
      trapezoid(4) = 400  {x coordinate of the bottom left point.}
      trapezoid(5) = 650  {x coordinate of the bottom right point.}
      trapezoid(6) = 500  {y coordinate of the bottom line segment.}
      call gpr_$trapezoid(trapezoid, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Draw a Filled Polygon (From Chapter 3)

```
      Program polygons
C The program draws a five-sided polygon.  It demonstrates the
gpr_$start_pgon,
C gpr_$pgon_polyline, and gpr_$close_fill_pgon routines.

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

      integer*2 array_of_x_coords(4), array_of_y_coords(4)
      integer*2 number_of_points_in_array
      data      array_of_x_coords/100,600,600,350/
      data      array_of_y_coords/600,600,300,100/
      data      number_of_points_in_array/4/

      call init(gpr_$borrow)

C Set the starting point of the polygon.
      call gpr_$start_pgon(int2(100), int2(300), status)

C Set the other four points of the polygon.
      call gpr_$pgon_polyline(array_of_x_coords, array_of_y_coords,
     +                       number_of_points_in_array, status)
      call check('calling pgon_polyline')

C Connect the five points and fill it with the current fill color.
      call gpr_$close_fill_pgon(status)
      call check('calling close_fill_pgon')

{Pause for 5 seconds, then terminate.}
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Create and Use a Nondefault Tile Pattern (From Chapter3)

```
       Program tile_pattern
C This program creates a nondefault tile pattern.  It demonstrates the
C gpr_$set_fill_value, gpr_$set_fill_background_value, and
C gpr_$set_fill_pattern routines.  You must run this program on a color node.

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      integer*4    fill_value, background_value
      integer*2    center(2), radius, x, y, scale, q
      integer*2    size_of_tile_pattern(2)
      integer*4    attribute_block_descriptor
      integer*4    bitmap_desc_of_tile_pattern
      integer*2    hi_plane_of_mmb
      data center/300,300/
      data radius/300/
      data scale/1/
      data hi_plane_of_mmb/0/
      data background_value/5/
      data fill_value/3/
      data size_of_tile_pattern/32,32/

      call init(gpr_$borrow)

C Generate a 32x32 main memory bitmap to hold the tile pattern.  It is
C essential that the hi_plane value be 0.  If it is not 0, then the calls
C to set the fill value and fill background value will have no effect.
      call gpr_$allocate_attribute_block(attribute_block_descriptor,
     +                                  status)
      call check('allocating attribute block')
      call gpr_$allocate_bitmap(size_of_tile_pattern, hi_plane_of_mmb,
     +                         attribute_block_descriptor,
     +                         bitmap_desc_of_tile_pattern, status)
      call check('allocating bitmap')
      call gpr_$set_bitmap(bitmap_desc_of_tile_pattern, status)
      call check('setting bitmap')

C Set every 16th bit in the bitmap.
      do x = 0,31
        q1 = MOD(x,4)
        do y = 0,31
          q2 = MOD(y,4)
          if ((q1 .eq. 0) .and. (q2 .eq. 0)) then
            call gpr_$move(x, y, status)
            call gpr_$line(x, y, status)
          end if
        enddo
      enddo
C Make the display bitmap current and then set 3 attributes in its attribute
C block.  Every 16th bit will be painted with the fill value; the other 15
C bits will be painted with the background fill value.
      call gpr_$set_bitmap(display_bitmap, status)
      call gpr_$set_fill_value(fill_value, status)
      call gpr_$set_fill_background_value(background_value, status)
      call gpr_$set_fill_pattern(bitmap_desc_of_tile_pattern, scale,
     +                          status)
```

```
C Draw a filled circle using the current tile pattern.
      call gpr_$circle_filled(center, radius, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Draw Thick and Dashed Lines (From Chapter 3)

```
      Program widths_and_patterns
C This program draws one thick line and two dashed lines.  It demonstrates how
C to use gpr_$set_draw_width, gpr_$set_line_pattern, and gpr_$set_draw_pat-
tern.

%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/gpr.ins.ftn´
%include ´my_common_block.ftn´

      integer*2  width_of_line_in_pixels, repeat_count
      integer*2  pattern(4)
      integer*2  length, style, scale

      call init(gpr_$borrow)

C The following sequence draws a line 11-pixels thick spanning x from 0 to 300
C and y from 245 to 255 inclusive.
      width_of_line_in_pixels = 11
      call gpr_$set_draw_width(width_of_line_in_pixels, status)
      call gpr_$move(  int2(0), int2(250), status)
      call gpr_$line(int2(300), int2(250), status)

C The following sequence creates a dashed line.  The dashed line pattern
C repeats itself every 150 pixels (which is equal to the repeat count times
C the length).  The pattern consists of 100 pixels set to the draw value,
C followed by 50 pixels set to the background value.  We specified this pat-
tern
C by setting the length  equal to 3 which instructed gpr_$set_line_pattern to
C evaluate only the three most significant bits in pattern.  The three most
C significant bits are set to ´110´.  When each bit is magnified by the repeat
C count, the pattern becomes fifty 1´s, fifty 1´s, and fifty 0´s.
      repeat_count = 50
      pattern(1) = 16#C000   { 1100000000000000 in binary }
      length = 3
      call gpr_$set_line_pattern(repeat_count, pattern, length, status)
      call gpr_$set_draw_width(int2(1), status)
      call gpr_$move(  int2(0), int2(450), status)
      call gpr_$line(int2(500), int2(450), status)


C The gpr_$set_linestyle call is an alternative to the gpr_$set_line_pattern
C call.  It is simpler to use, but is consequently less powerful.  The
C following sequence creates a pattern that repeats itself every 100 pixels,
C consisting of 50 drawn pixels followed by 50 background pixels.
      style = gpr_$dotted
      scale = 50
      call gpr_$set_linestyle(style, scale, status)
      call gpr_$move(int2(0), int2(650), status)
      call gpr_$line(int2(650), int2(650), status)

{Pause for 5 seconds, then terminate.}
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Write a Simple String to the Display (From Chapter 4)

```
      Program simple_text_example
C This program writes a simple string to the display.  It demonstrates the
C  gpr_$load_font_file, gpr_$set_text_font, and gpr_$text routines.

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

      character*5  pathname_of_font
      integer*2    pathname_length
      integer*2    font_id
      character*27 string_to_write
      integer*2    string_length
      data         string_to_write /'Hello from Apollo computer.'/
      data         string_length /27/

      call init(gpr_$borrow)

C Load a font.

C Pathname /sys/dm/fonts/f9x15 must exist.
      pathname_of_font = 'f9x15'
      pathname_length  = 5
      call gpr_$load_font_file(pathname_of_font, pathname_length,
     +                         font_id, status)
      call check('loading a load_font_file')
      call gpr_$set_text_font(font_id, status)
      call check('setting the text font')

C Write the string so that it begins at position 100,100
      call gpr_$move(int2(100), int2(100), status)
      call gpr_$text(string_to_write, string_length, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Uses More Than One Text Font (From Chapter 4)

```
      Program three_fonts
C This program writes a string to the display in three different fonts.  It
C demonstrates the gpr_$load_font_file, gpr_$set_text_font, gpr_$text, and
C gpr_$inq_text_extent routines.  The program uses the gpr_$inq_text_extent
C call to measure the string.

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


      character*44   output_string
      parameter      (output_string =
     +               'The rain in Spain falls mainly on the plain.')
      integer*2      string_length
      parameter      (string_length = 44)
      integer*2      font_sizes, font_id(3), pick
      integer*2      size_in_pixels(2)
      integer*2      count
      data count/0/

      call init(gpr_$borrow)

C Load three fonts into the font storage area of display memory.

C Prepare a large, medium, and small font.
      call gpr_$load_font_file('f9x15', int2(5), font_id(1), status)
      call gpr_$load_font_file('f7x13', int2(5), font_id(2), status)
      call gpr_$load_font_file('f5x7',  int2(4), font_id(3), status)

      do pick = 1, 3
C Establish one of the three prepared fonts as the current font.
        call gpr_$set_text_font(font_id(pick), status)
        count = count + 1
        call gpr_$move(int2(100), int2(count * 100), status)
C Write the string to the display in the current font.
        call gpr_$text(output_string, string_length, status)

C Calculate how much space (in pixels) the string requires.
        call gpr_$inq_text_extent(output_string, string_length,
     +                                  size_in_pixels, status)

C Write the space information.  Print sends information to stdout, not to
C the display.
        PRINT*,'Current font uses',size_in_pixels(1),' pixels in x dim.'
      enddo

{Pause for 5 seconds, then terminate.}
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Writes Text in Different Directions (From Chapter 4)

```
      Program text_direction
C This program writes a simple string to the display in each of the four
C possible printing directions.  It demonstrates gpr_$set_text_path.

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

      integer*2    font_id
      integer*2    direction
      character*6 strings(4)
      integer*2    count
      data         strings(1), strings(2), strings(3), strings(4)
     +             /' UP   ', ' DOWN ', ' LEFT ', ' RIGHT'/
      data         count/0/

      call init(gpr_$borrow)

C Load a font and make it current.
      call gpr_$load_font_file('f9x15', int2(5), font_id, status)
      call gpr_$set_text_font(font_id, status)

C The following sequence loops through each of the four possible print
C directions, printing one string in each direction.  Note that the starting
C position for each string is (500,400)

      do direction = gpr_$up, gpr_$right
          call gpr_$move(int2(500), int2(400), status)
          call gpr_$set_text_path(direction, status)
          count = count + 1
          call gpr_$text(strings(count), int2(6), status)
      enddo

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Modifies a Text Font's Characteristics (From Chapter 4)

```
      Program modifiable_fonts
C This program creates a modifiable copy of a font and then changes the gap
C between characters and the width after every period.  It demonstrates the
C gpr_$replicate_font, gpr_$set_character_width, and
C gpr_$set_horizontal_spacing routines.

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


      character*5  pathname_of_font
      integer*2    pathname_length
      integer*2    font_id, replicated_font_id
      character*43 string_to_write
      integer*2    string_length
      character*1  character_to_change
      integer*2    new_width_in_pixels
      integer*2    horizontal_spacing_in_pixels

      data         pathname_of_font /'f9x15'/
      data         pathname_length /5/
      data         string_to_write
     +             /'Hi there.Good morning.My name is Apollo.'/
      data         string_length /43/

      call init(gpr_$borrow)

C Prepare a font, make a copy of the font, and then make the copy of the
C font current.
      call gpr_$load_font_file(pathname_of_font, pathname_length,
     +                         font_id, status)
      call gpr_$replicate_font(font_id, replicated_font_id, status)
      call gpr_$set_text_font(replicated_font_id, status)

C Tighten the gap between characters by one pixel. This will cut off the right
C edge of wide characters.
      horizontal_spacing_in_pixels = -1
      call gpr_$set_horizontal_spacing(replicated_font_id,
     +     horizontal_spacing_in_pixels, status)

C But, leave a gap of 45 pixels after every period.
      character_to_change = '.'
      new_width_in_pixels = 45
      call gpr_$set_character_width(replicated_font_id,
     +     character_to_change, new_width_in_pixels, status)

C Write the string.
      call gpr_$move(int2(100), int2(100), status)
      call gpr_$text(string_to_write, string_length, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program To Create a Hidden Memory Bitmap (From Chapter 5)

```
      Program hidden_memory_bitmaps
C This program creates one hidden display memory bitmap, writes a circle to
C it, and then blts the bitmap to display memory so that the circle becomes
C visible.  It demonstrates the gpr_$allocate_hdm_bitmap,
C  gpr_$allocate_bitmap, gpr_$set_bitmap, and gpr_$pixel_blt routines.

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

      integer*2     size_of_hdm_bitmap(2)
      integer*4     attribute_block_descriptor
      integer*4     hdm_bitmap_descriptor, bitmap_descriptor
      integer*2     center_of_circle(2)
      integer*2     radius_of_circle
      integer*2     source_window(4)
      integer*2     destination_origin(2)

      data          center_of_circle/25,25/
      data          radius_of_circle/20/

      call init(gpr_$borrow)

C The following sequence creates a 224 by 224 HDM bitmap.  Notice how
C the attribute_block_descriptor returned by gpr_$allocate_attribute_block
C is used as an input parameter in gpr_$allocate_hdm_bitmap.  The value for
C hi_plane comes from 'my_include_file.pas'
      call gpr_$allocate_attribute_block(attribute_block_descriptor,
     +                                    status)
      size_of_hdm_bitmap(1) = 224
      size_of_hdm_bitmap(2) = 224
      call gpr_$allocate_hdm_bitmap(size_of_hdm_bitmap, hi_plane,
     +                              attribute_block_descriptor,
     +                              hdm_bitmap_descriptor, status)
      call check('allocating hdm bitmap')

C The current bitmap is now the display bitmap. Therefore, if we call a
C drawing or text routine, all data will be written to the display
C bitmap.  However, we want to draw a filled circle in the HDM bitmap.
C Before drawing the filled circle, we must make the HDM bitmap current
C by calling the gpr_$set_bitmap routine.  Since the
C gpr_$allocate_hdm_bitmap routine does not clear the data stored in
C hidden memory, we must clear it with the gpr_$clear routine.
      call gpr_$set_bitmap(hdm_bitmap_descriptor, status)
      call gpr_$clear(0, status)
      call gpr_$circle_filled(center_of_circle, radius_of_circle,
     +                        status)


C We now blt the entire HDM bitmap to a portion of display memory.
C The gpr_$pixel_blt command blts the specified pixels from the HDM
C bitmap to the current bitmap.  Therefore, before we call
C gpr_$pixel_blt, we must set the current bitmap back to the display
C bitmap.
      source_window(1) = 0  {Here, we specify the section of the HDM}
      source_window(2) = 0   {bitmap that we want to blt.}
      source_window(3) = 224
      source_window(4) = 224
      destination_origin(1) = 400  {Here, we specify where in display }
      destination_origin(2) = 400   {memory the system will blt to. }
C Make the display bitmap current.
      call gpr_$set_bitmap(display_bitmap, status)
      call gpr_$pixel_blt(hdm_bitmap_descriptor, source_window,
     +                    destination_origin, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Toggles Between Frame 0 and Frame 1 (From Chapter 5)

```
      Program toggling_display_frames
C This program toggles between frame 0 and frame 1.  It demonstrates
C gpr_$set_bitmap_dimensions and gpr_$select_color_frame.  This program will
C only run on a DN550, DN560, DN600, or DN660.

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

      integer*2   new_size_of_bitmap(2)
      integer*2   pictures, frame

      call init(gpr_$borrow)

C The gpr_$init call will not permit you to specify a borrow mode display
C bitmap larger than 1024 x 1024.  Because we want to create a 1024 x 2048
C bitmap, we must call gpr_$set_bitmap_dimensions after we call gpr_$init.
C
      new_size_of_bitmap(1) = 1024
      new_size_of_bitmap(2) = 2048
      call gpr_$set_bitmap_dimensions(display_bitmap,new_size_of_bitmap,
     +                                hi_plane, status)
      call check('resetting dimensions')
C It is a good idea to clear the entire display since HDM probably contains
C fonts,icons, etc.
      call gpr_$clear(0, status)

C We will now draw horizontal lines in frame 0 and vertical lines in frame 1,
C and toggle the two frames at 1 second intervals to examine these
C fast-changing images.
C
      do pictures = 1,20
        frame = MOD(pictures,2)
        if (frame .eq. 0) then
          call gpr_$move(int2(0),  pictures, status)
          call gpr_$line(int2(200),pictures, status)
        else
          call gpr_$move(pictures + 300,int2(1024), status)
          call gpr_$line(pictures + 300,int2(1224), status)
        end if
        call gpr_$select_color_frame(frame, status)
        call pause(1)
      end do

C Pause 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Creates a Main Memory Bitmap (From Chapter 5)

```
      Program main_memory_bitmaps
C This program creates one main memory bitmap, writes a circle in it, and blts
C the circle to display memory where it can become visible on the screen.
C It demonstrates gpr_$allocate_attribute_block, gpr_$allocate_bitmap,
C gpr_$set_bitmap, and gpr_$pixel_blt.

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

      integer*2   size_of_main_memory_bitmap(2)
      integer*4   attribute_block_descriptor
      integer*4   main_memory_bitmap_descriptor
      integer*2   source_window(4)
      integer*2   destination_origin(2)
      integer*2   center_of_circle(2)
      integer*2   radius_of_circle
      data        center_of_circle/25,25/
      data        radius_of_circle/20/

      call init(gpr_$borrow)

C The following sequence creates a 50 by 50 main memory bitmap.  Notice how
C the attribute_block_descriptor returned by gpr_$allocate_attribute_block
C is used as an input parameter in gpr_$allocate_bitmap.  The value for
C hi_plane comes from the init subroutine.
C
      call gpr_$allocate_attribute_block(attribute_block_descriptor,
     +                                    status)
      size_of_main_memory_bitmap(1) = 50
      size_of_main_memory_bitmap(2) = 50
      call gpr_$allocate_bitmap(size_of_main_memory_bitmap, hi_plane,
     +                    attribute_block_descriptor,
     +                    main_memory_bitmap_descriptor, status)

C The current bitmap is now the display bitmap. Therefore, if we call a figure
C drawing or text writing routine, all data will be written to the display
C bitmap.  However, we want to draw a filled circle in the main memory bitmap.
C Before drawing the filled circle, we must make the main memory bitmap cur-
rent
C by calling the gpr_$set_bitmap routine.
C
      call gpr_$set_bitmap(main_memory_bitmap_descriptor, status)
      call gpr_$circle_filled(center_of_circle, radius_of_circle,
     +                        status)

C We now blt the entire main memory bitmap to a portion of display memory.
C The gpr_$pixel_blt command blts the specified pixels from the main memory
C bitmap to the current bitmap.  Therefore, before we call gpr_$pixel_blt,
C we must set the current bitmap back to the display bitmap.
C
      source_window(1) = 0  {Here, we specify the section of the HDM}
      source_window(2) = 0   {bitmap that we want to blt.}
      source_window(3) = 100
      source_window(4) = 100
      destination_origin(1) = 400  {Here, we specify where in display }
      destination_origin(2) = 400   {memory the system will blt to. }
C Make the display bitmap current.
      call gpr_$set_bitmap(display_bitmap, status)
      call gpr_$pixel_blt(main_memory_bitmap_descriptor, source_window,
     +                    destination_origin, status)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Creates a Pixel Oriented External File Bitmap (From Chapter 5)

```
      Program pixel_oriented_bitmaps
C This program creates a pixel-oriented bitmap for a 4- or 8-plane color node.
C You should run this program twice.  The first time, select the 'create'
C option so that the program will create a file containing a bitmap.  The
C second time, select the 'display' option so that the program will display
C the contents of this file on your monitor.
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

C These lines should be included in all subroutines
      integer*4     attribs
      integer*2     bytes_per_line
      integer*4     storage
C      character*1   memvec(16#7fffff)
C      pointer       /storage/ memvec
      character*256 pathname_of_bitmap
      integer*4     disk_bitmap

      common /glob_stor/bytes_per_line, storage, attribs, disk_bitmap,
     +                  pathname_of_bitmap
C down to here!!!!

      integer*2 dest_origin(2), source_window(4)
      character*8 my_choice
      integer*2   choice, create, display
      parameter(create  = 0)
      parameter(display = 1)
      data dest_origin   /400,400/
      data source_window /0,0,200,200/

      print *,'Do you want to create a bitmap or display a bitmap?'
      write(*,10)
10    format('(enter ''create'' or ''display'') -- ',$)
      read(*,20) my_choice
20    format(a)
      if (my_choice.eq.'create') then
          choice = create
          print *,'be patient; it will take some time to create the ',
     +            'bitmap.'
      else
          choice = display
      endif
      print *,'what is the pathname of the file you want to ',
     +         my_choice, ' -- '
      read(*,20) pathname_of_bitmap

      call init(gpr_$borrow)

      if (choice.eq.create) then                      {Create the bitmap}
          call gpr_$allocate_attribute_block(attribs,status)
          call check('allocating an attribute block.')
          call access_external_bitmap(gpr_$create)
          call draw_figure
      else                                           {Display the bitmap}
          PRINT *, 'displaying the bitmap'
          call access_external_bitmap(gpr_$readonly)
          call gpr_$set_bitmap(display_bitmap, status) {BLT the bitmap from}
                                                      {disk to display memory.}
          call check('setting bitmap')
          call gpr_$pixel_blt(disk_bitmap, source_window, dest_origin,
     +                        status)
          call check('blting')
          call pause(5)
      endif

      call gpr_$clear(0, status)
      call gpr_$terminate(.false., status)
      call check('terminating the graphics package')
      end
C ****************************************************************************
      SUBROUTINE access_external_bitmap(access)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
      INTEGER*2 ACCESS

c     these lines should be included in all subroutines
      integer*4     attribs
      integer*2     bytes_per_line
      integer*4     storage
C      character*1   memvec(16#7fffff)
C      pointer       /storage/ memvec
      character*256 pathname_of_bitmap
      integer*4     disk_bitmap

      common /glob_stor/ bytes_per_line, storage, attribs, disk_bitmap,
     +                   pathname_of_bitmap
c     down to here!!!!
%include 'my_common_block.ftn'
      INTEGER*2 version(2), size(2), groups, header(8)
      INTEGER*4 BPS,SO
      LOGICAL*4 created
      EQUIVALENCE(BPS, HEADER(5))
      EQUIVALENCE(SO,  HEADER(7))

      version(1)    = 1
      version(2)    = 1
      size(1)       = 200 {X size of bitmap}
      size(2)       = 200 {Y size of bitmap}
      groups        = 1
      header(1)     = 1
      header(2)     = hi_plane + 1
      header(3)     = 8
      header(4)     = 0
      BPS           = 0
      SO            = 0

      CALL gpr_$open_bitmap_file(access,pathname_of_bitmap,
     1                            INT2(LEN(pathname_of_bitmap)),
     1                            version, size, groups,
     2                            header, attribs, disk_bitmap, created,
     3                            status)
      CALL check('opening an external file bitmap')
      PRINT *,'allocated_size = ', header(3)
      bytes_per_line = header(4)
      storage = SO
      END
C *****************************************************************************
      subroutine set_1_pixel_in_external_bitmap(x,y,color_index)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
      integer*2 x, y, color_index
c These lines should be included in all subroutines
      integer*4     attribs
      integer*2     bytes_per_line
      integer*4     storage
      character*1   memvec(16#7fffff)
      pointer       /storage/ memvec
      character*256 pathname_of_bitmap
      integer*4     disk_bitmap
      common /glob_stor/bytes_per_line, storage, attribs, disk_bitmap,
     +                  pathname_of_bitmap
c down to here!!!!

%include 'my_common_block.ftn'
      integer*2 number_of_bytes_per_pixel
      parameter(number_of_bytes_per_pixel = 1)

      integer*4 offset

C Convert a two-dimensional pixel position to a one-dimensional offset from
C the beginning of the bitmap.
      offset = (y * int4(bytes_per_line))
     1         + (x * number_of_bytes_per_pixel)

C Now, load the color index for this pixel into the bitmap.
      memvec(offset) = char(color_index)
      end
C *****************************************************************************
      SUBROUTINE DRAW_FIGURE()
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
      integer*2 x, y, q, color_index
C This simple routine calculates the color value
C for every pixel in the bitmap, but it does not actually store any values in
C the bitmap.  The SET_1_PIXEL_IN_EXTERNAL BITMAP routine does that.
%include 'my_common_block.ftn'

      do 10, x=0,200
          do 20, y=1,200
              color_index = mod(y,16)
              call set_1_pixel_in_external_bitmap(x, y, color_index)
20        continue
10    continue
      end
```

A Program That Demonstrates Double-Buffer Techniques (From Chapter 5)

```
      Program double_buffer_example
C This program demonstrates double buffering techniques. It uses the
C gpr_$allocate_buffer, gpr_$set_bitmap, gpr_$inq_visible_buffer,
C gpr_$select_display_buffer routines.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2  what_to_do_to_invis_bitmap
      integer*2  images_to_display, counter
      parameter (images_to_display = 20)
      integer*4  buffer_bitmap
      common /glob_stor/ buffer_bitmap

      PRINT *, 'Enter 0 for gpr_$clear_buffer,'
      PRINT *, '      1 for gpr_$undisturbed_buffer,'
      PRINT *, '   or 2 for gpr_$copy_buffer -- '
      READ  *, what_to_do_to_invis_bitmap

      call INIT(gpr_$borrow)      {Create the primary bitmap.}
      call CREATE_A_BUFFER_BITMAP {Create the buffer bitmap.}

      do counter = 1,images_to_display
        call DRAW_PATTERN_IN_INVISIBLE_BITMAP(counter)
        call MAKE_INVISIBLE_BITMAP_VISIBLE(what_to_do_to_invis_bitmap)
C and make the visible bitmap invisible.
        call PAUSE(1)
      end do

C Pause for 5 seconds; then terminate.
      call PAUSE(5)   {So user can examine the final image.}
      call gpr_$terminate(.false., status)
      end
C ***************************************************************************
      subroutine CREATE_A_BUFFER_BITMAP
%include 'my_common_block.ftn'
      integer*4  buffer_bitmap
      common /glob_stor/ buffer_bitmap
C This procedure allocates a buffer bitmap and then clears its contents.}
      call gpr_$allocate_buffer(display_bitmap, buffer_bitmap, status)
      call check('allocating a buffer bitmap')
      call gpr_$set_bitmap(buffer_bitmap, status)
      call gpr_$clear(0, status)
      end
C ***************************************************************************
      logical function  IS_DISPLAY_BITMAP_VISIBLE()
%include 'my_common_block.ftn'
      integer*4  buffer_bitmap
      common /glob_stor/ buffer_bitmap
      integer*4  visible_bitmap

      call gpr_$inq_visible_buffer(visible_bitmap, status)
      if (visible_bitmap .eq. display_bitmap) then
        is_display_bitmap_visible = .true.
      else
```

```
        is_display_bitmap_visible = .false.
      end if
      end
C ************************************************************************************
      subroutine DRAW_PATTERN_IN_INVISIBLE_BITMAP(count)
%include 'my_common_block.ftn'
      integer*4  buffer_bitmap
      common /glob_stor/ buffer_bitmap
      integer*2  count
C This procedure adds one circle to the bitmap that is currently invisible.

      integer*2  center(2)
      integer*2  radius
      integer*4  visible_bitmap, invisible_bitmap, fill_value
      logical    answer
      logical    is_display_bitmap_visible
      external   is_display_bitmap_visible

      answer = is_display_bitmap_visible()
C Make the invisible bitmap current so that we can draw to it:
      if (answer)  then
        call gpr_$set_bitmap(buffer_bitmap, status)
        call check('setting bitmap to buffer_bitmap')
      else
        call gpr_$set_bitmap(display_bitmap, status)
        call check('setting bitmap to display_bitmap')
      end if
      fill_value = MOD(count,16) + 1
      call gpr_$set_fill_value( fill_value, status)

C Draw a filled circle in the current bitmap.
      radius = 80
      center(1) = radius + (count * 12)
      if (MOD(count,2) .eq. 1) then
        center(2) = 100
      else
        center(2) = 400
      end if
      call gpr_$circle_filled(center, radius, status)
      call check('drawing filled circle')
      end
C ************************************************************************************
      subroutine MAKE_INVISIBLE_BITMAP_VISIBLE(op)
%include 'my_common_block.ftn'
      integer*4  buffer_bitmap
      common /glob_stor/ buffer_bitmap
      integer*2  op  {Double_buffer_option_t}
      integer*4  display_desc, option_desc, option_value
      integer*2  options
      logical    answer
      logical    is_display_bitmap_visible
      external   is_display_bitmap_visible

      answer = is_display_bitmap_visible()
      if (answer) then
        option_desc = display_bitmap
        display_desc = buffer_bitmap
      else
        option_desc =  buffer_bitmap
        display_desc = display_bitmap
      end if

      option_value = 0  {ignored unless options = gpr_$clear_buffer}
      options      = op

C Toggle the bitmaps.
      call gpr_$select_display_buffer(display_desc, option_desc,
     +                                 option_value,options, status)
      call check('selecting a display buffer')
      end
```

A Program That Demonstrates The Three Different Kinds of Block Transfers (BLTs) (From Chapter 5)

```
      Program blts
C This program contrasts the three BLT calls.  It demonstrates the
C gpr_$bit_blt, gpr_$pixel_blt, and gpr_$additive_blt calls.  This program
C must be run on a color node.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2     center(2)
      integer*2     radius
      integer*4     fill_value
      integer*2     main_mem_bm_size(2)
      integer*4     attrib_desc
      integer*4     main_mem_bitmap, source_bitmap_desc
      integer*2     source_window(4)
      integer*2     destination_origin(2)
      integer*2     source_plane, destination_plane
      integer*2     size(2)

      data fill_value/6/
      data main_mem_bm_size/100, 100/
      data center/50,50/
      data radius/40/
      data source_window/0,0,100,100/
      data size/500,500/

      call init(gpr_$borrow)

C Create a main memory bitmap.
      call gpr_$allocate_attribute_block(attrib_desc, status)
      call gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc,
     +                          main_mem_bitmap, status)

C Draw a filled circle in the main memory bitmap.
      call gpr_$set_bitmap(main_mem_bitmap, status)
      call gpr_$set_fill_value(fill_value, status)
      call gpr_$circle_filled(center, radius, status)

      source_bitmap_desc = main_mem_bitmap
      call gpr_$set_bitmap(display_bitmap, status)

C BLT the circle from main memory to display memory three different ways.
C First, BLT the contents of every plane in the main memory bitmap to every
C plane in the display memory bitmap.
C
      destination_origin(1) = 400  {X coordinate}
      destination_origin(2) = 0    {Y coordinate}
      call gpr_$pixel_blt(source_bitmap_desc, source_window,
     +                    destination_origin, status)
      call check('pixel blt')

C Second, BLT the contents of plane 1 in the main memory bitmap to plane 3
C of the display memory bitmap.
C
```

```
      source_plane = 1
      destination_plane = 3
      destination_origin(1) = 400
      destination_origin(2) = 200
      call gpr_$bit_blt(source_bitmap_desc, source_window, source_plane,
     +                  destination_origin, destination_plane, status)
      call check('bit blt')

C Third, BLT the contents of plane 0 in the main memory bitmap to every plane
C of the display memory bitmap.  Since plane 0 contains all zeros, the
C circle will probably be displayed in black and may therefore be invisible.
C
      source_plane = 0
      destination_origin(1) = 400
      destination_origin(2) = 400
      call gpr_$additive_blt(source_bitmap_desc, source_window,
     +                       source_plane, destination_origin, status)
      call check('additive blt')

C Pause 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Sets and Employs Two Attribute Blocks for the Same Bitmap (From Chapter 5)

```
      Program attribute_blocks
C This program allocates two different attribute blocks and assigns different
C attributes to them.  It demonstrates the gpr_$allocate_attribute_block and
C gpr_$set_attribute_block routines.   If you run this program on a monochrome
C display, you'll get one set of attributes, and if you run it on a color
C display, you'll get a different set of attributes.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*4   attrib_desc1, attrib_desc2
      integer*4   main_mem_bitmap
      integer*2   main_mem_bm_size(2)
      integer*2   source_window(4)
      integer*2   destination_origin(2)

      data main_mem_bm_size/400, 400/
      data source_window/0,0,400,400/
      data destination_origin/100, 100/

      call init(gpr_$borrow)

C Allocate two attribute blocks.  Both attribute blocks will begin with the
C default attributes.
      call gpr_$allocate_attribute_block(attrib_desc1, status)
      call gpr_$allocate_attribute_block(attrib_desc2, status)

C Allocate a main memory bitmap and make it current.  The third argument could
C have been attrib_desc1 or attrib_desc2; the results will be the same.
      call gpr_$allocate_bitmap(main_mem_bm_size, hi_plane,
     +                          attrib_desc1, main_mem_bitmap, status)
      call gpr_$set_bitmap(main_mem_bitmap, status)

C Define nondefault attributes for a monochrome node.
      if (hi_plane .eq. 0) then
C Associate the attributes in attrib_desc1 with the main memory bitmap.
        call gpr_$set_attribute_block(attrib_desc1, status)
C Assign one nondefault attribute value to this attribute block.}
        call gpr_$set_draw_width(int2(20), status)

C Define nondefault attributes for a color node.
      else
C Associate the attributes in attrib_desc2 with the main memory bitmap.
        call gpr_$set_attribute_block(attrib_desc2, status)
C Assign two nondefault attribute values to this attribute block.
        call gpr_$set_draw_width(int2(4), status)
        call gpr_$set_draw_value(3, status)
      end if

C Draw a line in the main memory bitmap from 0,0 to 300,300
      call gpr_$line(int2(300), int2(300), status)

C BLT the main memory bitmap to the screen.
      call gpr_$set_bitmap(display_bitmap, status)
      call gpr_$pixel_blt(main_mem_bitmap, source_window,
     +                    destination_origin, status)
      call check('doing pixel blt')

C Pause 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Converts a Color Bitmap to an Array, Manipulates the Array, and Converts the Array Back to a Bitmap (From Chapter 5)

```
      Program read_write_pixels
C This program demonstrates how to manipulate a bitmap with the
C gpr_$read_pixel and gpr_$write_pixel routines.  This program will work
C only on a color node.
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      integer*2  i,j
      integer*2  center(2)
      integer*2  radius
      integer*2  source_window(4)
      integer*2  destination_window(4)
      integer*4  pixel_array(100,100)
      integer*4  fill_color, new_fill_color, new_background_color
      parameter (fill_color = 2)
      parameter (new_fill_color = 3)
      parameter (new_background_color = 4)
      data       source_window/0,0,100,100/
      data       destination_window/0,200,100,100/
      data       center/50,50/
      data       radius/40/

      call init(gpr_$borrow)

C Draw a filled circle on the screen.
      call gpr_$set_fill_value(fill_color, status)
      call gpr_$circle_filled(center, radius, status)

C Convert a 100x100xhi_plane portion of the display bitmap into an array.
C Note that the call does not change the existing bitmap in anyway.
      call gpr_$read_pixels(source_window, pixel_array, status)
      call check('reading pixels')

C Examine each element of the array.  Give each element a new fill color
C and a new background color.  FORTRAN programmers should be aware that
C the array will appear backwards; that is, the first element is y and
C the second element is x.
      do i = 1,100
        do j = 1,100
          if (pixel_array(i,j) .eq. fill_color) then
            pixel_array(i,j) = new_fill_color
          else
            pixel_array(i,j) = new_background_color
          end if
        end do
      end do
C Now convert the modified array back to a bitmap and put it in the designated
C region of the display bitmap.
      call gpr_$write_pixels(pixel_array, destination_window, status)
      call check('writing pixels')

C Pause 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Manipulates a Main Memory Bitmap Through a Bitmap Pointer (From Chapter 5)

```
      program inq_bm_ptrs_in_main_mem
C This program demonstrates how to access the virtual addresses of a main
C memory bitmap.  It illustrates the gpr_$inq_bitmap_pointer routine.  This
C program allocates a main memory bitmap and then sets the value of every
C pixel associated with this bitmap to color index 5.  It then BLTs the bitmap
C to the screen.  This program must be run on a color node.
%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/gpr.ins.ftn´
%include ´my_common_block.ftn´
      integer*4  scan_line(15)
      integer*4  storage_ptr
      pointer   /storage_ptr/ scan_line
      integer*2  width_of_bitmap, height_of_bitmap, array_size
      integer*2  main_mem_bm_size(2)
      integer*2  stor_line_width, scan, c
      integer*2  dest_origin(2)
      integer*2  window(4)
      integer*2  plane, attrib_desc
      integer*4  main_mem_bitmap
      parameter (width_of_bitmap = 480)
      parameter (height_of_bitmap = 320)
      parameter (array_size = 15)  {array_size = width_of_bitmap / 32}
      data       main_mem_bm_size/width_of_bitmap,height_of_bitmap/
      data       window/0,0,width_of_bitmap,height_of_bitmap/
      data       dest_origin/10,10/

      call init(gpr_$borrow)
C Create a main memory bitmap.
      call gpr_$allocate_attribute_block(attrib_desc, status)
      call gpr_$allocate_bitmap(main_mem_bm_size, hi_plane, attrib_desc,
     +                          main_mem_bitmap, status)
C Set every pixel in this bitmap to color index 5.  We do this by setting
C every bit in plane 0 and plane 2 to ´1´, and by setting every bit in the
C other planes to ´0´.
      call gpr_$inq_bitmap_pointer(main_mem_bitmap, storage_ptr,
     +                             stor_line_width, status)
      do plane = 0,hi_plane
        do scan = 1,height_of_bitmap
          do c = 1,array_size        { array_size*32 = width_of_bitmap }
            if ((plane .eq. 0) .OR. (plane .eq. 2)) then
              scan_line(c) = 16#FFFFFFFF  {Set all 32 bits to ´1´}
            else
              scan_line(c) = 16#00000000  {Set all 32 bits to ´0´}
            end if
          end do
C In Pascal, pointers point to bytes. Therefore, since stor_line_width is a
C 2-byte integer, we must multiply it by 2 to get the number of bytes.
          storage_ptr = storage_ptr + (2 * stor_line_width)
        end do
      end do
C BLT the main memory bitmap to the screen.
      call gpr_$pixel_blt(main_mem_bitmap, window, dest_origin, status)
      call pause(5)   {Pause 5 seconds, then terminate.}
      call gpr_$terminate(.true.,status)
      end
```

A Program That Manipulates a Display Memory Bitmap Through a Bitmap Pointer (From Chapter 5)

```
      program inq_bm_ptrs_in_disp_mem
C This program demonstrates how to access the virtual address of a display
C memory bitmap.  It illustrates the gpr_$inq_bitmap_pointer,
C gpr_$enable_direct_access, and gpr_$remap_color_memory routines.
%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/gpr.ins.ftn´
%include ´my_common_block.ftn´
      integer*4   a_line(15), another_line
      integer*4   storage_ptr, storage_ptr_save
      pointer    /storage_ptr/ a_line
      pointer    /storage_ptr_save/ another_line
      integer*2   plane
      integer*2   storage_line_width, scan_line, c
      integer*2   rectangle(4)
      integer*2   width_of_rectangle, height_of_rectangle, array_size
      parameter  (width_of_rectangle = 480)
      parameter  (height_of_rectangle = 320)
      parameter  (array_size = 15)
      data        rectangle/0,0,width_of_rectangle,height_of_rectangle/
      call init(gpr_$borrow)
C Display a rectangle filled with color index 15.
      call gpr_$set_fill_value(15, status)
      call gpr_$rectangle(rectangle, status)
      call pause(1)

      call gpr_$inq_bitmap_pointer(display_bitmap, storage_ptr_save,
     +                             storage_line_width, status)
      do plane = 0,3
C The first time through the loop, the system will remap color memory so that
C the starting virtual address of plane 0 will be equal to storage_ptr_save.
C The next time through the loop, the system will remap color memory so that
C the starting virtual address of plane 1 will be equal to storage_ptr_save,
C and so on with planes 2 and 3.  We must reestablish the value of storage_ptr
C each time through the loop.
         call gpr_$remap_color_memory(plane, status)
         storage_ptr = storage_ptr_save
C It is good programming practice to call gpr_$enable_direct_access before
C writing directly to the bitmap (i.e., before changing bits without using
C GPR calls).

         call gpr_$enable_direct_access(status)

         do scan_line = 1,height_of_rectangle
           do c = 1,(width_of_rectangle / 32)
C Assign 32 bits.
             a_line(c) = 16#00F0F0F0
           end do
           storage_ptr  = storage_ptr + (2 * storage_line_width)
         end do
         call pause(1)
      end do
      call pause(2) {Pause 2 seconds, then terminate.}
      call gpr_$terminate(.true.,status)
      end
```

A Program That Manipulates a Hidden Display Memory Bitmap Through a Bitmap Pointer (From Chapter 5)

```
      PROGRAM inq_bm_ptrs_in_hdm
C This program demonstrates how to access the virtual addresses of a HDM
C bitmap.  It illustrates the gpr_$inq_bitmap_pointer routine.  This program
C allocates a HDM and then sets the value of every pixel associated with
C this bitmap to color index 5 (which is yellow on the default color map).
C It then BLTs the HDM bitmap to the screen.  This program must be run on a
C color node.
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      integer*4   a_line(4), another_line
      integer*4   hdm_ptr, hdm_ptr_save
      pointer    /hdm_ptr/ a_line
      pointer    /hdm_ptr_save/ another_line
      integer*2  source_window(4)
      integer*2  stor_line_width, index, plane, scan
      integer*4  attrib_desc, hdm_bitmap
      integer*2  size_of_hdm_bitmap(2)
      integer*2  dest_origin(2)
      integer*2  width_of_bitmap, height_of_bitmap, array_size
      parameter (width_of_bitmap = 128)
      parameter (height_of_bitmap = 64)
      parameter (array_size = 4)  {array_size = width_of_bitmap / 32}
      data       source_window/0,0,width_of_bitmap,height_of_bitmap/
      data       dest_origin/10,10/
      data       size_of_hdm_bitmap/width_of_bitmap,height_of_bitmap/
      call init(gpr_$borrow)
C Create hidden display memory bitmap.
      call gpr_$allocate_attribute_block(attrib_desc, status)
      call gpr_$allocate_hdm_bitmap(size_of_hdm_bitmap, hi_plane,
     +                              attrib_desc, hdm_bitmap, status)
      call gpr_$inq_bitmap_pointer(hdm_bitmap, hdm_ptr_save,
     +                             stor_line_width, status)
      do plane = 0,hi_plane
        call gpr_$remap_color_memory(plane, status)
        hdm_ptr = hdm_ptr_save
        call gpr_$enable_direct_access(status)
        do scan = 1,height_of_bitmap
          do index = 1,array_size
C Set every bit in plane 0 and plane 2 to '1'.  Set every bit in all other
C planes to '0'
            if ((plane .eq. 0) .OR. (plane .eq. 2)) then
              a_line(index) = 16#FFFFFFFF
            else
              a_line(index) = 16#00000000
            end if
          end do
          hdm_ptr = hdm_ptr + (2 * stor_line_width)
        end do
      end do
      call gpr_$pixel_blt(hdm_bitmap, source_window, dest_origin,
     +                    status) {BLT the HDM bitmap to the screen.}
      call pause(5)  {Pause 5 seconds, then terminate.}
      call gpr_$terminate(.true.,status)
      end
```

A Program That Draws a Line, Fills a Triangle, and Writes Some Text in Different Colors (From Chapter 6)

```
      Program setting_colors
C This program controls the colors used to draw a line, fill a triangle, and
C print text.  It demonstrates the gpr_$set_draw_value, gpr_$set_fill_value,
C gpr_$set_text_value, and gpr_$set_text_background_value.  Note that this
C program does not establish a new color map; therefore, the colors displayed
C on your screen will depend on whatever is stored in the current color map.
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
C ****************************************************************************
      call init(gpr_$borrow)

      call draw_a_colored_line
      call fill_a_triangle_with_color
      call write_text_in_color

{Pause 5 seconds, then terminate.}
      call pause(5)
      call gpr_$terminate(.false., status)
      end
C ****************************************************************************
      subroutine draw_a_colored_line

      call gpr_$set_draw_value(1, status)  {Draw value is 1}
      call gpr_$line(int2(90), int2(150), status)
      end
C ****************************************************************************
      subroutine fill_a_triangle_with_color
      integer*2  v1(2), v2(2), v3(2)
      data   v1/100,100/,  v2/100,300/,  v3/300,100/

      call gpr_$set_fill_value(2, status)    {Fill value is 2}
      call gpr_$triangle(v1, v2, v3, status)
      end
C ****************************************************************************
      subroutine write_text_in_color
      integer*2   font_id, string_length
      character*32 string
      data         string/'Greetings from the written world'/
      data         string_length/32/

      call gpr_$load_font_file('f9x15', int2(5), font_id, status)
      call check('loading the font')
      call gpr_$set_text_font(font_id, status)
      call gpr_$set_text_value(3, status)           {Text value is 3}

C The default text background index is 0.
      call gpr_$move(int2(20), int2(320), status)
      call gpr_$text(string, string_length, status)

C But we can establish a nondefault text background index.
      call gpr_$set_text_background_value(7,status)  {Text background is 7}
      call gpr_$move(int2(20), int2(350), status)
      call gpr_$text(string, string_length, status)
      end
```

A Program That Alters the Color Map (From Chapter 6)

```
      Program color_circles
C This program demonstrates how to build a new color map. It demonstrates
C the gpr_$set_color_map call.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
C ***************************************************************************
      call init(gpr_$borrow)
      call create_color_map
      call draw_concentric_circles

C Pause 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
C ***************************************************************************
C This subroutine creates a color map containing eight entries.
      subroutine create_color_map
      integer*2    start_index
      integer*2    number_of_entries
      integer*4    color_map(0:7)
      integer*4    color_entry
      external     color_entry

      color_map(0) = color_entry(int2(0),int2(0),int2(0))       {black}
      color_map(1) = color_entry(int2(0),int2(0),int2(255))     {dark blue}
      color_map(2) = color_entry(int2(50),int2(50),int2(255))
      color_map(3) = color_entry(int2(75),int2(75),int2(255))
      color_map(4) = color_entry(int2(100),int2(100),int2(255))
      color_map(5) = color_entry(int2(150),int2(150),int2(255))
      color_map(6) = color_entry(int2(200),int2(200),int2(255)) {light blue}
      color_map(7) = color_entry(int2(255),int2(255),int2(255)) {white}

C Now that the array 'color_map' contains the correct values, establish it
C as the real system color map.
      start_index = 0
      number_of_entries = 8
      call gpr_$set_color_map(start_index,number_of_entries, color_map,
     +                             status)
      call check('setting color map')
      end
C ***************************************************************************
      integer*4 function  color_entry(red, green, blue)
      integer*2   red, green, blue

      color_entry = (red * 16#010000) + (green * 16#000100) + blue
      return
      end
C ***************************************************************************
C This subroutine draws 7 concentric circles - each a lighter shade of blue.
C It draws the innermost circle in white.
      subroutine draw_concentric_circles
      integer*2  center(2)
      integer*2  radius
      integer*4  index
```

```
data center/300,300/

radius = 300
do index = 1,7
   call gpr_$set_fill_value(index,status)
   call gpr_$circle_filled(center,radius,status)
   radius = radius - 40
end do
end
```

A Program That Saves the Original Color Map, Loads a New Color Map, and Then Restores the Original Color Map (From Chapter 6)

```
      Program restore_color_table
C This program demonstrates how to save the current color map, create a new
C color map, and then restore the original color map.  It demontrates the
C gpr_$inq_color_map and gpr_$set_color_map routines.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*4   start_index
      integer*2   number_of_entries
      integer*4   old_color_map(0:15)
      logical     unobs
      parameter (start_index = 0)
      parameter (number_of_entries = 16)

      call init(gpr_$direct)

C Save color map the system is currently using.
      call gpr_$inq_color_map(start_index, number_of_entries,
     +                        old_color_map, status)
      call check('inquiring about color map')
      call draw_circles
      call pause(2)

C Create new color map
      call create_new_color_map
      call pause(2)

C Restore the color map the system was using at the beginning of the program.
      unobs = gpr_$acquire_display(status)
      call gpr_$set_color_map(start_index, number_of_entries,
     +                        old_color_map, status)
      call gpr_$release_display(status)
      call pause(2)

C Terminate the program.
      call gpr_$terminate(.false., status)
      end
C ****************************************************************************
C This subroutine draws 16 circles to illustrate the current contents of the
C color map.
      subroutine draw_circles
      integer*4   n
      integer*2   center(2)
      logical     unobs
      center(1) = 50
      unobs = gpr_$acquire_display(status)
      do n = 0,15
        call gpr_$set_fill_value(n, status)
        center(2) = 25*n
        call gpr_$circle_filled(center, int2(15), status)
      end do
      call gpr_$release_display(status)
      end
```

```
C **********************************************************************************
C Load a new color map.
      subroutine create_new_color_map
      integer*4   start_index
      integer*2   number_of_entries
      integer*4   new_color_map(0:15), a_color_map(0:15)
      logical     unobs
      integer*2  n

      new_color_map(0) = 0 {Keep color slot 0 black.}
      do n = 1,15
        new_color_map(n) = (16#010000 * (15 * n)) +
     +                     (16#000100 * 255) + (255 - (15 * n))
      end do

      unobs = gpr_$acquire_display(status)
      start_index = 0

      number_of_entries = 16
      call gpr_$set_color_map(start_index, number_of_entries,
     +                        new_color_map, status)
      call check('setting color map')
      call gpr_$inq_color_map(start_index, number_of_entries,
     +                        a_color_map, status)
      call gpr_$release_display(status)
      end
```

A Program That Creates a Plane Oriented Bitmap Containing a Special Color Map (From Chapter 6)

```
      Program plane_oriented_bitmaps
C This program creates a plane-oriented bitmap.  It demonstrates the
C gpr_$open_bitmap_file, gpr_$set_bitmap_file_color_map, gpr_$set_color_map,
C and gpr_$inq_bitmap_file_color_map routines.  Run this program twice.
C The first time, select the 'create' option so that the program will create
an
C external file bitmap.  The second time you run it, select the 'display'
C option so that the program will display the contents of this file on your
C screen.  You should run this program on a color node.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      character*256 pathname_of_bitmap
      integer*4  attribs, disk_bitmap
      common /glob_stor/ pathname_of_bitmap, attribs, disk_bitmap

      character*8 my_choice
      integer*2   choice, create, display
      parameter(create  = 0)
      parameter(display = 1)

C ****************************************************************************
      print *,'Do you want to create a bitmap or display a bitmap?'
      write(*,10)
10    format('(enter ''create'' or ''display'') -- ',$)
      read(*,20) my_choice
20    format(a)
      if (my_choice.eq.'create') then
          choice = create
          print *,'be patient; it will take some time to create the ',
     +            'bitmap.'
      else
          choice = display
      endif
      print *,'what is the pathname of the file you want to ',
     +         my_choice, ' -- '
      read(*,20) pathname_of_bitmap

      call init(gpr_$borrow)

      if (my_choice .eq. 'create') then     {Create the bitmap}
        call gpr_$allocate_attribute_block(attribs,status)
        call access_external_bitmap(gpr_$create)
        call create_bitmap_file_color_map
        call draw_figure_in_file_bitmap
      else                          {Display the bitmap}
        call display_the_bitmap
        call pause(5)
      end if

      call gpr_$terminate(.false., status)
      end
C ****************************************************************************
      subroutine access_external_bitmap(access)
```

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      integer*2 access
      character*256 pathname_of_bitmap
      integer*4  attribs, disk_bitmap
      common /glob_stor/ pathname_of_bitmap, attribs, disk_bitmap

      integer*2 version(2), size(2), groups, header(8)
      logical*4 created
      integer*4 bps,so
      equivalence(bps, header(5))
      equivalence(so,  header(7))

      version(1) = 1                                             {version}
      version(2) = 1
      size(1)    = 200  {x size of bitmap}                          {size}
      size(2)    = 200  {y size of bitmap}
      groups      = 1                                             {groups}
      header(1)   =  hi_plane + 1  {number of sections}
{start of header}
      header(2)   = 1              {pixel size}
      header(3)   = 0              {allocated_size)
      header(4)   = 0              {bytes per line}
      bps         = 0              {bytes per section}
      so          = 0              {storage offset}

      CALL gpr_$open_bitmap_file(access ,pathname_of_bitmap,
     +                           INT2(LEN(pathname_of_bitmap)),
     +                           version, size, groups, header,
     +                           attribs, disk_bitmap, created, status)
      CALL check('opening an external file bitmap')
      WRITE (*,10) header(3)
10    FORMAT('allocated_size = ', I2)
      end
C *****************************************************************************
      subroutine DRAW_FIGURE_IN_FILE_BITMAP {Draw a line in the file bitmap.}
C You can use the GPR line calls to draw lines in a plane-oriented bitmap
C however, these calls won't work in a pixel-oriented bitmap.
      character*256 pathname_of_bitmap
      integer*4  attribs, disk_bitmap
      common /glob_stor/ pathname_of_bitmap, attribs, disk_bitmap

      integer*4  draw_value
      parameter (draw_value = 15)

      call gpr_$set_bitmap(disk_bitmap, status)
      call gpr_$set_draw_value(draw_value, status)
      call gpr_$line(int2(150), int2(150), status)
      end
C *****************************************************************************
      subroutine CREATE_BITMAP_FILE_COLOR_MAP

      character*256 pathname_of_bitmap
      integer*4  attribs, disk_bitmap
      common /glob_stor/ pathname_of_bitmap, attribs, disk_bitmap
      integer*2  counter
      integer*2  blue_component, red_component, green_component
      integer*4  color_map(0:15)
      integer*4  starting_index
      integer*2  n_of_colors
      parameter (starting_index = 0)
      parameter (n_of_colors = 16)
      parameter (blue_component = 0)
      parameter (red_component = 255)

      color_map(0) = 0 {Keep slot 0 black}
      do counter = 1,15
        green_component = counter*16
        color_map(counter) = ((red_component   * 16#010000) +
     +                       (green_component * 16#000100) +
     +                       (blue_component  * 16#000001))
      end do
C We now establish the array as the color map for the external file bitmap.
      call gpr_$set_bitmap_file_color_map(disk_bitmap, starting_index,
     +                                  n_of_colors, color_map, status)
      call check('setting the bitmap file color map')
      end
C *****************************************************************************
      subroutine DISPLAY_THE_BITMAP

      character*256 pathname_of_bitmap
      integer*4  attribs, disk_bitmap
      common /glob_stor/ pathname_of_bitmap, attribs, disk_bitmap

      integer*4  color_map(0:15)
      integer*2  source_window(4)
      integer*2  dest_origin(2)
      integer*4  starting_index
      integer*2  n_of_colors
      parameter (starting_index = 0)
      parameter (n_of_colors = 16)
      data  source_window/0,0,200,200/
      data  dest_origin/400,400/

      call access_external_bitmap(gpr_$readonly)   {open the file for reading.}

C Load the external file color map into array variable color_map.
      call gpr_$inq_bitmap_file_color_map(disk_bitmap, starting_index,
     +                         n_of_colors, color_map, status)

C Load the colors from array variable color_map into the system color map.
      call gpr_$set_color_map(starting_index, n_of_colors, color_map,
     +                        status)

C BLT the external file bitmap to display memory.
      call gpr_$set_bitmap(display_bitmap, status)
      call gpr_$pixel_blt(disk_bitmap,source_window,dest_origin,status)
      call check('blting from disk to display')
      end
```

A Program That Demonstrates 24-Plane True-Color Programming (From Chapter 6)

```
      Program lines_in_true_color
C This program demonstrates true-color programming.  It draws 1020 lines, each
C line having a different color. Notice that we never call gpr_$set_color_map.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*4  color_total
      integer*2  r
      integer*2  y_coord_of_line, left_end_of_line, right_end_of_line
      integer*2  red_component, green_component, blue_component
      data green_component/255/
      data y_coord_of_line/0/
      data left_end_of_line/0/
      data right_end_of_line/1000/

      call init(gpr_$borrow_rgb)

C This program draws 1020 lines, each line a different color.
      do r = 1,4
        red_component = (r * 64) - 1  {red_component = 63,127,191,255}
        do blue_component = 1,255

CCreate a unique draw value.
          color_total = (red_component * 16#010000) +
     +                  (green_component * 16#000100) +
     +                   blue_component
          call gpr_$set_draw_value(color_total, status)

{Draw the line with this draw value.}
          y_coord_of_line = y_coord_of_line + 1
          call gpr_$move(left_end_of_line, y_coord_of_line, status)
          call gpr_$line(right_end_of_line, y_coord_of_line, status)
        enddo
      enddo

C Pause 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Demonstrates Color Zoom (From Chapter 6)

```
      Program zooming
C This program demonstrates how to magnify (zoom) the display bitmap.
C It uses the gpr_$color_zoom call.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2   x_mag_factor, y_mag_factor
      integer*2   center(2)
      integer*2   radius
      data        radius/50/
      data        center/100,100/

      PRINT *, 'Enter the magnification factor in the X dimension -- '
      READ  *, x_mag_factor
      PRINT *, 'Enter the magnification factor in the Y dimension -- '
      READ  *, y_mag_factor

      call init(gpr_$borrow)

      call gpr_$circle_filled(center, radius, status)
      call pause(1)

      if (x_mag_factor .gt. display_characteristics(21)) then
        PRINT *, 'X magnification factor is too high for this node.'
      else if (y_mag_factor .gt. display_characteristics(22)) then
        PRINT *, 'Y magnification factor is too high for this node.'
      else if ((y_mag_factor .eq. 1) .AND. (x_mag_factor .ne. 1)) then
        PRINT *, 'You cannot set the Y mag factor to 1 if the '
        PRINT *, 'X mag factor is greater than 1'
      else
        call gpr_$color_zoom(x_mag_factor, y_mag_factor, status)
        call check('zooming')
        call pause(5)
      end if

C Terminate.
      call gpr_$terminate(.false., status)
      end
```

A Program That Enables Two Event Types and Waits for One of Them To Occur (From Chapter 7)

```
      program simple_events
C This program enables two kinds of events.  it then waits for one of those
C events to occur and reports which one occurred.
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/kbd.ins.ftn'
%include 'my_common_block.ftn'
      integer*2 event_type
      character event_data
      integer*2 position(2)
      logical   unobs

      call init(gpr_$borrow)
      call enable_events
C wait for the user to type a lowercase letter or to depress a mouse button.
      unobs = gpr_$event_wait(event_type, event_data, position, status)
      call gpr_$terminate(.false., status)
C report on the event that occurred.
      if (event_type .eq. gpr_$keystroke) then
        print 10, event_data
10      format ('You typed the letter ', a)
      else
        print *, 'You hit a mouse button.'
      end if
      end
C ***************************************************************************
C This procedure enables the three mouse buttons and all lowercase letters.
      subroutine enable_events
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      integer*2 event_type
      integer*4 key_set(8), mouse_buttons(8)
      integer*2 setsize
      character event_data
      integer*2 ascii_value
      logical unobs
      parameter (setsize = 256)

      event_type = gpr_$keystroke
      call lib_$init_set(key_set, setsize)
      do ascii_value = 97,122
        call lib_$add_to_set(key_set, setsize, ascii_value)
      end do
      call gpr_$enable_input(event_type, key_set, status)

      event_type = gpr_$buttons
      call lib_$init_set(mouse_buttons, setsize)
      do ascii_value = 97,99 {Ascii values of mouse buttons}
        call lib_$add_to_set(mouse_buttons, setsize,
     +                          ascii_value)
      enddo
      call gpr_$enable_input(event_type, mouse_buttons, status)
      call check('enabling input')
      end
```

A Program That Leaves a Line Trail (From Chapter 7)

```
      Program locator_events
C This program enables and reports locator events.  It uses the s
C gpr_$enable_input and gpr_$event_wait routines.  The program allows you to
C sketch lines.  To begin, depress the left-most mouse button, then move the
C cursor.  To end, release the left-most mouse button.
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/kbd.ins.ftn'
%include 'my_common_block.ftn'
      integer*2 event_type
      integer*4 key_set(8)
      character event_data
      integer*2 position(2)
      logical   sketch_started, unobs, q

      call init(gpr_$borrow)
      call enable_events
      sketch_started = .false.
      q = .true.
      do while (q)
        unobs = gpr_$event_wait(event_type, event_data, position,
     +                          status)
        if ((event_type .eq. gpr_$buttons) .AND.
     +      (event_data .eq. KBD_$M1D)) then
               call gpr_$move(position(1), position(2),status)
               sketch_started = .true.
        else if ((event_type .eq. gpr_$locator_update) .AND.
     +          (sketch_started)) then
                 call gpr_$line(position(1), position(2), status)
        else if ((event_type .eq. gpr_$buttons) .AND.
     +           (event_data .eq. KBD_$M1U)) then
                 q = .false.
        end if
      end do
C Pause 5 seconds; then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
C ****************************************************************************
C This procedure enables the tracking of the mouse and the left-most mouse
button.
      subroutine enable_events
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      integer*2 event_type
      integer*4 mouse_buttons(8), key_set(8)
      event_type = gpr_$locator_update
      call gpr_$enable_input(event_type, key_set, status)
      event_type = gpr_$buttons
      call lib_$init_set(mouse_buttons, int2(256))
      call lib_$add_to_set(mouse_buttons, int2(256), int2(97))
      call lib_$add_to_set(mouse_buttons, int2(256), int2(65))
      call gpr_$enable_input(event_type, mouse_buttons, status)
      call check('enabling input')
      end
```

A Program That Monitors Input Events in Multiple Windows (From Chapter 7)

```
      Program gpr_in_multiple_windows
C This program creates a GPR program that runs in one shell and two windows.
C It primarily demonstrates the pad_$create_window, gpr_$event_wait, and
C gpr_$set_window_id calls (though several other GPR and PAD calls are also
C used).  In this program, all output from writeln statements will be printed
C in stdout (which is probably the shell that you invoke the program from).
C We create two windows by calling pad_$create_window twice.  We also
C call gpr_$init twice, once for each window, so that direct mode graphics can
C run in both windows.  The user will move the cursor into one of the two
C windows.  The program will draw a line into the chosen window.  We used the
C gpr_$set_window_id call to mark the two windows with a unique identification
C character.  When the user moves the cursor into one of the two windows, the
C gpr_$event_wait routine will return the identification character of the
C appropriate window.
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'
      logical    unobs
      integer*2  event_type
      character  event_data
      integer*2  position(2)

      call create_two_windows
      PRINT *, 'Two empty windows just appeared.  Move the cursor into '
      PRINT *, 'one of them.  The program will draw a line in the '
      PRINT *, 'selected window.'

      PRINT *, 'Waiting for an event'
      unobs = gpr_$event_wait(event_type, event_data, position, status)

      PRINT *, event_data
      call draw_a_line_in_window(event_data)

C Pause 2 seconds; then terminate.
      call  pause(2)
      call gpr_$terminate(.false., status)
      end
C ***************************************************************************
      subroutine ASSIGN_SOME_STUFF_TO_BITMAPS(graphics_strid)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/pad.ins.ftn'
%include 'my_common_block.ftn'
      integer*2  graphics_strid
      integer*2  event_type
      integer*4  key_set(8)
      character  event_data

C enable one event.
      event_type = gpr_$entered_window
      call gpr_$enable_input(event_type, key_set, status)
      call check('enabling input')
      call pad_$set_auto_close(graphics_strid, int2(1), .true., status)
      call gpr_$set_obscured_opt(gpr_$pop_if_obs, status)
      call gpr_$set_auto_refresh(.true.,status)
      end
```

```
C ****************************************************************************
      subroutine CREATE_TWO_WINDOWS
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include '/sys/ins/pad.ins.ftn'
C This procedure creates and initializes two windows.
      integer*2  mode
      integer*2  graphics_strid1, graphics_strid2
      integer*2  window1(4), window2(4)
      integer*2  display_bm1_size(2), display_bm2_size(2)
      integer*4  display_bm1, display_bm2, status
      integer*2  hi_plane
      parameter (mode = gpr_$direct)
      parameter (hi_plane = 3)
      data      window1/0,0,400,400/
      data      window2/500,0,200,200/
      data      display_bm1_size/400,400/
      data      display_bm2_size/200,200/
C Create two windows.
      call pad_$create_window('', int2(0), pad_$transcript, int2(1),
     +                        window1, graphics_strid1, status)
      call pad_$create_window('', int2(0), pad_$transcript, int2(1),
     +                        window2, graphics_strid2, status)
C Initialize graphics in both windows and give each window a distinct char id.
      call gpr_$init(mode, graphics_strid1, display_bm1_size, hi_plane,
     +               display_bm1, status)
      PRINT *, display_bm1
      call check('initing 1')
      call gpr_$set_window_id('1', status)
      call check('setting window id')
      call ASSIGN_SOME_STUFF_TO_BITMAPS(graphics_strid1)

      call gpr_$init(mode,graphics_strid2,display_bm2_size,
     +               hi_plane,display_bm2,status)
      PRINT *, display_bm2
      call check('initing 2')
      call gpr_$set_window_id('2', status)
      call ASSIGN_SOME_STUFF_TO_BITMAPS(graphics_strid2)
      end
C ****************************************************************************
      subroutine DRAW_A_LINE_IN_WINDOW(window_that_cursor_is_in)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
      integer*4  display_bm1, display_bm2, status
      character window_that_cursor_is_in
      logical   unobs
      PRINT *, 'Drawing a line'
      if (window_that_cursor_is_in .eq. '1') then
        call gpr_$set_bitmap(display_bm1, status)
      else
        call gpr_$set_bitmap(display_bm2, status)
      end if

      unobs = gpr_$acquire_display(status)
      call gpr_$line(int2(100), int2(100), status)
      call gpr_$release_display(status)
      end
```

A Program That Creates a Nondefault Cursor Pattern (From Chapter 8)

```
      Program nondefault_cursor_example
C This program creates a nondefault cursor pattern.  It demonstrates the
C gpr_$set_cursor_position, gpr_$set_cursor_pattern,and gpr_$set_cursor_active
C routines.  The program draws a pattern in a main memory bitmap, and then
C establishes this bitmap as the current cursor pattern.  It then activates
C this cursor and gives it a starting position of 200,200.  The cursor will
C automatically follow the motion of the locator device.  When you move the
C locator device, you'll see the default nonblinking cursor pattern.  When
C you stop moving the locator device, you'll see the nondefault cursor pattern
C (which blinks).
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*4  cbd  {Cursor bitmap descriptor}
      integer*4  create_cursor_pattern
      external   create_cursor_pattern

      call init(gpr_$borrow)

      cbd = create_cursor_pattern()
      call initialize_cursor(cbd)

C Pause for 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
C ***************************************************************************
      integer*4 function CREATE_CURSOR_PATTERN()
      integer*2  size_of_bitmap(2)
      integer*4  attributes_descriptor
      integer*4  cursor_bitmap_descriptor
      data       size_of_bitmap/16,16/

C Allocate a small main memory bitmap.
      call gpr_$allocate_attribute_block(attributes_descriptor, status)
      call gpr_$allocate_bitmap(size_of_bitmap, hi_plane,
     +                          attributes_descriptor,
     +                          cursor_bitmap_descriptor, status)
      create_cursor_pattern = cursor_bitmap_descriptor

C Draw an arrow pattern inside the main bitmap.
      call gpr_$set_bitmap(cursor_bitmap_descriptor, status)
      call gpr_$move(int2(8), int2(15), status)
      call gpr_$line(int2(8), int2(0), status)
      call gpr_$line(int2(1), int2(7), status)
      call gpr_$line(int2(15),int2(7), status)
      call gpr_$line(int2(8), int2(0), status)
      end
 {**************************************************************************}
      subroutine INITIALIZE_CURSOR(cursor_bitmap_descriptor)
      integer*4  cursor_bitmap_descriptor
      logical    cursor_active
      integer*2  cursor_position(2)
      integer*2  cursor_origin(2)
      parameter (cursor_active = .true.)
      data           cursor_position/200,200/
      data           cursor_origin/8,0/

C Make the main memory bitmap into the current cursor pattern.
      call gpr_$set_cursor_pattern(cursor_bitmap_descriptor, status)

C Establish 200,200 as the starting cursor position.
      call gpr_$set_cursor_position(cursor_position, status)

C Make the cursor visible.
      call gpr_$set_cursor_active(cursor_active, status)

C Sets position 8,0 as the cursor origin.
      call gpr_$set_cursor_origin(cursor_origin, status)
      end
```

A Program That Tracks a Nondefault Cursor as the Mouse Moves (From Chapter 8)

```
      Program tracking_the_cursor
C This program shows how you can display a nondefault cursor wherever the
C mouse tracks to.  It uses a combination of cursor and event calls to
C demonstrate this feature.  Enter <CTRL-Q> to exit from the program.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2 event_type
      integer*2 mouse_position(2), cursor_position(2)
      CHARACTER event_data
      integer*4 key_set(8)
      integer*4 cbd
      integer*4 create_cursor_pattern
      external  create_cursor_pattern

      call init(gpr_$borrow)

      cbd = create_cursor_pattern()
      call initialize_cursor(cbd)

C Activate the locator.
      event_type = gpr_$locator
      call gpr_$enable_input (event_type, key_set, status)

      do while (.true.)
         unobs = gpr_$event_wait(event_type, event_data, mouse_position,
     +                           status)
         if (event_type .eq. gpr_$locator) then
           cursor_position(1) = mouse_position(1)
           cursor_position(2) = mouse_position(2)
           call gpr_$set_cursor_position(cursor_position, status)
           cursor_position(1) = cursor_position(1) + 25
           call gpr_$circle(cursor_position, int2(1), status)
           call check('drawing circle')
         end if
      end do

C Terminate the graphics package.
      call gpr_$terminate(.false., status)
      end
C ***************************************************************************
      integer*4   function CREATE_CURSOR_PATTERN()
      integer*4   cursor_bitmap_descriptor
      integer*2   size_of_bitmap(2)
      integer*4   attributes_descriptor
      integer*2   center(2)
      data        size_of_bitmap/16,16/
      data        center/8,8/

      call gpr_$allocate_attribute_block(attributes_descriptor, status)
      call gpr_$allocate_bitmap(size_of_bitmap, hi_plane,
     +                          attributes_descriptor,
     +                          cursor_bitmap_descriptor, status)
      create_cursor_pattern = cursor_bitmap_descriptor
```

```
C Draw the cursor pattern (a small filled circle) inside the main bitmap.
      call gpr_$set_bitmap(cursor_bitmap_descriptor, status)
      call gpr_$circle_filled(center, int2(6), status)
      end
C *********************************************************************************
      subroutine INITIALIZE_CURSOR(cursor_bitmap_descriptor)
      integer*4  cursor_bitmap_descriptor
      integer*2  cursor_origin(2)
      data       cursor_origin/8,1/

      call gpr_$set_cursor_active(.false., status)
      call gpr_$set_cursor_origin(cursor_origin, status)
      call gpr_$set_cursor_pattern(cursor_bitmap_descriptor, status)
      call gpr_$set_cursor_active(.true., status)
      end
```

A Program That Demonstrates Clipping (From Chapter 9)

```
      Program clipping
C  This program demonstrates how to create and activate a clipping window,
C  and how a clipping window affects a drawing.  This program creates and
C  activates a 200x200 clipping window.  Inside the clipping window,
C  we draw a filled circle.  The parts of the circle outside the clipping
C  window are not drawn.  If you experiment with the radius size, this
C  effect will become more apparent.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2   clip_window(4)
      integer*2   center(2)
      integer*2   radius
      data center/500,500/

      call init(gpr_$borrow)

      clip_window(1) = 400   {x coord of window base}
      clip_window(2) = 400   {y coord of window base}
      clip_window(3) = 200   {x size of window}
      clip_window(4) = 200   {y size of window}
      call gpr_$set_clip_window(clip_window, status) {Create clipping window.}
      call gpr_$set_clipping_active(.true., status){Activate clipping window.}

      radius = 116
      call gpr_$circle_filled(center, radius, status)  {Draw filled circle}

C Pause 5 seconds; then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Sets a Plane Mask (From Chapter 9)

```
      Program masking
C This program creates a plane mask on a color node.  It demonstrates the
C gpr_$set_plane_mask_32 routine.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2  center(2)
      integer*2  radius
      integer*4  mask
      data center/500,500/
      data radius/200/

      call init(gpr_$borrow)

C We set the fill value to 15, which corresponds to each bit in planes 0
C through 3 being set to a 1.
      call gpr_$set_fill_value(15, status)

C However, we now set a plane mask on planes 0, 1, and 2, meaning that plane
C 3 cannot be altered.
      mask = (2 ** 0) + (2 ** 1) + (2 ** 2) {Set emulation technique}
      call gpr_$set_plane_mask_32(mask, status)

C Therefore, although the fill value is 15, the effective fill value is
C actually only 7 (0111). The circle will be filled with whatever color
C corresponds to index 7.
      call gpr_$circle_filled(center, radius, status)  {Draw filled circle}

C Pause 5 seconds; then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Direct Mode Program That Acquires and Releases the Display (From Chapter 10)

```
      program acquire_release
C This program acquires and releases the display 5000 times.  It demonstrates
C the gpr_$acquire_display and gpr_$release_display routines.  To prove that
C the display is really being released, move to another window at some point
C during this program's execution and try to type something.  If the display
C is really being released, the typed letters will appear.  This program will
C only run in an unobscured window.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2  x, y, count, radius
      integer*2  center(2)
      parameter  (radius = 4)
      logical    unobs

      call init(gpr_$direct)

      do count = 5,5000
C We do not need to have the display acquired to do these calculations.
        x = MOD((count ** 2), display_characteristics(5))
        y = MOD((count * x),  display_characteristics(6))

        unobs = gpr_$acquire_display(status) {Acquire the display.}
         call gpr_$line(x, y, status)              {Draw a line.}
         call check('Drawing line')
         center(1) = x
         center(2) = y
         call gpr_$circle_filled(center, radius, status) {Draw circle.}
         call check('Drawing circle')
        call gpr_$release_display(status)        {Release the display.}
      end do

C Terminate program.
      call gpr_$terminate(.false., status)
      end
```

A Direct Mode Program That Draws in the Unobscured Regions of a Window (From Chapter 10)

```
      Program writing_in_obscured_windows
C This program draws lines in the nonobscured parts of the window that the
C program runs in.  It demonstrates the gpr_$set_obscured_opt and
C gpr_inq_vis_list routines.  After starting this routine, you should obscure
C portions of the window by growing or popping other windows on top of the
C window running the program.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2   old_x, old_y, new_x, new_y, count, n
      integer*2   what_to_do_if_obscured
      integer*2   slots_available
      integer*2   slots_total
      integer*2   vis_list(4,32), a_rectangle(4)
      logical     unobs
      parameter (slots_available = 32) {Set an arbitrary limit of 32}
      data  what_to_do_if_obscured/gpr_$ok_if_obs/

      call init(gpr_$direct)
      call gpr_$set_clipping_active(.true., status)

      call gpr_$set_obscured_opt(what_to_do_if_obscured, status)

      do count = 1,3000 {This loop draws up to 3000 lines.}
         new_x = MOD ((count ** 2),600)
         new_y = MOD ((count * new_x), 610)

         unobs = gpr_$acquire_display(status)
C Find the visible slots every time display is acquired.
         call gpr_$inq_vis_list(slots_available, slots_total, vis_list,
     +                          status)
         call check('inquiring vis list')

C This loop draws a line from the old coordinates to the new coordinates
C slots_total times.  There is no harm (other than lost time) in drawing
C a line over the same coordinates.
         do n = 1, slots_total
            do n2 = 1,4
               a_rectangle(n2) = vis_list(n2,n)
            end do
            call gpr_$set_clip_window(a_rectangle, status)
            call gpr_$move(old_x, old_y, status)
            call gpr_$line(new_x, new_y, status)
         end do

         old_x = new_x
         old_y = new_y
         call gpr_$release_display(status)
      end do

C {Terminate program.
      call gpr_$terminate(.false., status)
      end
```

A Direct Mode Program That Demonstrates Window Refresh (From Chapter10)

```
      SUBROUTINE DRAW()
C This is the refresh procedure (subroutine).
%INCLUDE '/sys/ins/base.ins.ftn'
%INCLUDE '/sys/ins/gpr.ins.ftn'
%INCLUDE 'my_common_block.ftn'

C Draws a box bisected by a line.
      IF (.NOT.gpr_$acquire_display(status)) CALL ERROR_$PRINT(STATUS)
      CALL gpr_$draw_box(INT2(200), INT2(200), INT2(600), INT2(600),
     +                   status)
      CALL gpr_$move(INT2(200), INT2(200), status)
      CALL gpr_$line(INT2(600), INT2(600), status)
      CALL gpr_$release_display(status)
      end
C ****************************************************************************
      PROGRAM refresh_example

C This program contains a refresh procedure.  It demonstrates the
C gpr_$set_refresh_entry routine.  The program draws a simple design.  You
C should obscure a portion of the window the program is running in and then
C pop the window so that the program will require a refresh.  The refresh
C will redraw the simple design.  To leave the program, type the letter Q.

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

      LOGICAL     unobscured
      INTEGER*2   ev_pos(2), ev_type
      CHARACTER*1 ev_char
      INTEGER*4   keys(8)

      EXTERNAL DRAW
%INCLUDE 'my_common_block.ftn'

      data        keys/8*0/

      call lib_$add_to_set(keys, int2(256), int2(ichar('q')))
      call lib_$add_to_set(keys, int2(256), int2(ichar('q')))
      call init(gpr_$direct)

c Draw a picture.
      call draw()

C Establish the refresh procedure.  If window needs to be refreshed as
C the result of a pop, the draw procedure will automatically be called.

      CALL gpr_$set_refresh_entry(draw, INT4(0), status)
      CALL check('setting the refresh entry procedure.')

C Enter the letter Q to exit the program.
      CALL gpr_$enable_input(gpr_$keystroke, keys, status)
      IF (.NOT.gpr_$event_wait(ev_type, ev_char, ev_pos, status))
     +    CALL ERROR_$PRINT(STATUS)
C Terminate.
      CALL gpr_$terminate(.false., status)
      END
```

A Direct Mode Program That Accepts High-Level Input From The Input Pad (From Chapter 10)

```
      PROGRAM high_level_input
C This program prompts the user for the coordinates of a line, and then draws
C a line based on the user´s input.
C
%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/gpr.ins.ftn´
%include ´my_common_block.ftn´

      integer*2 x_start, y_start, x_stop, y_stop
      logical   unobs

10    FORMAT (´Enter x coord of line start: ´, $)
20    FORMAT (´Enter y coord of line start: ´, $)
30    FORMAT (´Enter x coord of line stop:  ´, $)
40    FORMAT (´Enter y coord of line stop:  ´, $)

      call init(gpr_$direct)
      call gpr_$set_cursor_active(.true.,status)

      WRITE (*, 10)
      READ *, x_start
      WRITE (*, 20)
      READ *, y_start
      WRITE (*, 30)
      READ *, x_stop
      WRITE (*, 40)
      READ *, y_stop

      unobs = gpr_$acquire_display(status)
      call gpr_$move(x_start, y_start, status)
      call gpr_$line(x_stop, y_stop, status)
      call gpr_$release_display(status)

C Pause 5 seconds, then terminate.
      call pause(5)
      call gpr_$terminate(.false.,status)
      end
```

A Program That Sets the Raster Operation for a Draw Operation (From Chapter 11)

```
      Program intersecting_lines
C This program draws two intersecting lines to demonstrate raster operations.
C It uses the gpr_$set_raster_op and gpr_$inq_raster_ops routines.
C Assuming that the default color map is loaded, the program will draw one
C red line and one green line.  Because we set the raster operation to 7,
C the intersection of the two lines will be drawn in blue.  This program
C should be run on a color node.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2  raster_op
      integer*2  raster_op_array(0:31)
      integer*2  plane

      call init(gpr_$borrow)

      call gpr_$set_draw_width(int2(20), status)
      call gpr_$set_draw_value(1, status)  {Draws a red line.}
      call gpr_$move(int2(0), int2(200), status)
      call gpr_$line(int2(400), int2(200), status)

      raster_op = 6
      do plane = 0, hi_plane
         call gpr_$set_raster_op(plane, raster_op, status)
      end do

      call gpr_$set_draw_value(2, status)
      call gpr_$move(int2(200), int2(0), status)
      call gpr_$line(int2(200), int2(400), status) {Draws a blue line.}

      call gpr_$inq_raster_ops(raster_op_array, status)
      do plane = 0,hi_plane
         WRITE (*,10) plane, raster_op_array(plane)
      enddo
10    format('Raster op for plane', I2, ' = ', I2)

C Pause 5 seconds then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Sets the Raster Operation for a BLT (From Chapter 11)

```
      Program raster_ops_in_blts
C This program demonstrates how you can use raster operations with BLTs.
C It uses the gpr_$set_raster_op routine.  Assuming that the default color
C map is loaded, the program will draw one blue rectangle and one yellow
C rectangle.  Then it will BLT the blue rectangle to the area covered by
C the yellow rectangle.  Because the raster operation is 1 for each plane,
C the BLT will produce a red rectangle.  This program should be run on a
C color node.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2  raster_op
      integer*2  plane
      integer*2  source_window(4)
      integer*2  destination_origin(2)
      integer*2  rectangle1(4)
      integer*2  rectangle2(4)

      data destination_origin/400,400/
      data rectangle1/0,0,200,200/
      data source_window/0,0,200,200/
      data rectangle2/400,400,200,200/

      call init(gpr_$borrow)

      call gpr_$set_fill_value(3, status)
      call gpr_$rectangle(rectangle1, status)  {Draw a blue rectangle.}
      call gpr_$set_fill_value(5, status)
      call gpr_$rectangle(rectangle2, status)  {Draw a yellow rectangle.}
      call pause(2)

      raster_op = 1  {Destinaton = Source AND Destination}
      do plane = 0,hi_plane
        call gpr_$set_raster_op(plane, raster_op, status)
      end do

C BLT the blue rectangle to the area covered by the yellow rectangle.
      call gpr_$pixel_blt(display_bitmap, source_window,
     +                    destination_origin, status)

C Pause 5 seconds then terminate.
      call pause(5)
      call gpr_$terminate(.false., status)
      end
```

A Program That Restricts Raster Operations to Particular Categories (From Chapter 11)

```
      Program raster_op_categories
C This program demonstrates how to set different raster operations for
C three different categories of GPR routines.  It demonstrates the
C gpr_$raster_op_prim_set and gpr_$set_raster_op routines.  (Nothing is
C actually drawn on the screen.)
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      integer*2   raster_op
      integer*2   plane_to_set_rop_on
      integer*2   prim_set
      parameter  (plane_to_set_rop_on = 0)

      call init(gpr_$borrow)

C Set the raster operation to 5 for all line drawing routines.
      prim_set = (2 ** gpr_$rop_line)  {Set emulation technique}
      raster_op = 5
      call gpr_$raster_op_prim_set(prim_set, status)
      call gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status)

C Set the raster operation to 7 for all fill routines.
      prim_set = (2 ** gpr_$rop_fill)  {Set emulation technique}
      raster_op = 7
      call gpr_$raster_op_prim_set(prim_set, status)
      call gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status)

C Set the raster operation to 13 for all blt routines.
      prim_set = (2 ** gpr_$rop_blt)
      raster_op = 13
      call gpr_$raster_op_prim_set(prim_set, status)
      call gpr_$set_raster_op(plane_to_set_rop_on, raster_op, status)

      call gpr_$terminate(.false., status)
      end
```

A Program That Sets the Decomposition Technique (From Appendix E)

```
      PROGRAM set_the_decomposition_technique
C This program sets the decomposition technique to gpr_$non_overlapping_tris
C on certain node types.  It demonstrates the gpr_$pgon_decomp_technique
C routine.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      INTEGER*2         node_type
      INTEGER*2         prim_set
      INTEGER*2         set_size

      CALL init(gpr_$borrow)

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_$ctl_mono_4) .OR.
     +    (node_type .EQ. gpr_$ctl_color_4) .OR.
     +    (node_type .EQ. gpr_$ctl_color_1)) THEN
      call gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status)
      call check('setting pgon decomp technique')
      end if

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_$rop_fill)
      call gpr_$raster_op_prim_set(prim_set, status)
      call check('setting the raster op prim set')


c **************************************************
c Graphics application code goes here.
c **************************************************

      call gpr_$terminate(.false.,status)
      end
```

A Program That Fills a Polygon Using the Triangle Technique (From Appendix E)

```
      PROGRAM triangle_technique
C This program demonstrates the triangle technique.  It illustrates the
C gpr_$set_triangle_fill_criteria and gpr_$multitriangle routines.
C
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gpr.ins.ftn'
%include 'my_common_block.ftn'

      logical     unobs
      integer*2   prim_set
      integer*2   t_list(30,7)
      integer*2   winding_set(2)
      integer*4   pixel_array(1)
      integer*2   n_triangles
      integer*2   n_positions
      integer*2   node_type
      integer*2   x_array(0:2), y_array(0:2)
      integer*2   x,y
      integer*2   list_size
      integer*2   window(4)
      parameter  (list_size = 30)
      parameter  (n_positions = 3)
      data        window/200,200,1,1/


      call init(gpr_$direct)
C If the node is not a DN3000, DN570, or DN580, set the decomposition
C technique to nonoverlapping_tris.
      node_type = display_characteristics(1)
      if ((node_type .EQ. gpr_$ctl_mono_1) .OR.
     +    (node_type .EQ. gpr_$ctl_mono_2) .OR.
     +    (node_type .EQ. gpr_$ctl_mono_4) .OR.
     +    (node_type .EQ. gpr_$ctl_color_4) .OR.
     +    (node_type .EQ. gpr_$ctl_color_1)) THEN
      call gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status)
      call check('setting pgon decomp technique')
      end if

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_$rop_fill)
      call gpr_$raster_op_prim_set(prim_set, status)
      call check('setting the raster op prim set')

      unobs = gpr_$acquire_display(status)
      call gpr_$read_pixels(window, pixel_array, status)
      call check('reading pixels')

C 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
      call gpr_$start_pgon(x, y, status)
      call gpr_$pgon_polyline(x_array, y_array, n_positions, 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
      call gpr_$start_pgon(x, y, status)
      call gpr_$pgon_polyline(x_array, y_array, n_positions, 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
      call gpr_$start_pgon(x, y, status)
      call gpr_$pgon_polyline(x_array, y_array, n_positions, 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
      call gpr_$start_pgon(x, y, status)
      call gpr_$pgon_polyline(x_array, y_array, n_positions, status)
C **********


      winding_set(1) = gpr_$parity
      call gpr_$set_triangle_fill_criteria(winding_set,status)
      call check('setting triangle fill crit with gpr_$parity.')

      call gpr_$close_return_pgon_tri(list_size, t_list, n_triangles,
     +                                  status)
      call check('closing pgon')

C Draw the triangles with a parity fill.
      call gpr_$multitriangle(t_list, n_triangles, status)
      call check('filling triangles')

      call pause(5)
      call gpr_$clear(pixel_array(1), status)
      call check('clearing')

      winding_set(1) = gpr_$nonzero
      call gpr_$set_triangle_fill_criteria(winding_set,status)
      call check('setting triangle fill crit with gpr_$nonzero.')

C Draw the triangles with a nonzero fill.
      call gpr_$multitriangle(t_list, n_triangles, status)
      call check('filling triangles')

      call pause(5)

      call gpr_$clear(pixel_array(1), status)
      call check('clearing')

      winding_set(1) =  gpr_$specific
      winding_set(2) =  2
      call gpr_$set_triangle_fill_criteria(winding_set,status)
      call check('setting triangle fill crit with gpr_$specific.')

C Draw the triangles with a specific winding number fill.
      call gpr_$multitriangle(t_list, n_triangles, status)
      call check('filling triangles')

      call pause(5)
      call gpr_$release_display(status)
      call gpr_$terminate(.FALSE.,status)
      end
```

Appendix D

Node-Dependent Data

Table D-1 lists information specific to various Domain node types.

Table D-1. Node-Specific Data

Node Type	Amount of Visible Display Memory	Amount of Hidden Display Memory	Number of Planes
DN100	800 x 1024	224 x 1024	1
DN300, DN320, or DN330	1024 x 800	1024 x 224	1
DN400, DN420	800 x 1024	224 x 1024	1
DN460	1024 x 800	1024 x 224	1
DN550, DN560			
4-plane interactive	1024 x 800	1024 x 1024 & 1024 x 224	4
8-plane imaging	1024 x 800	1024 x 224	8
8-plane interactive	1024 x 800	1024 x 1024 & 1024 x 224	8
24-plane imaging	1024 x 800	512 x 512 & 512 x 512	24
DN570	1024 x 800	1024 x 219	8
DN580, DN580-T	1280 x 1024	224 x 1024	8
DN590, DN590-T	1280 x 1024	224 x 1024	24
DN600, DN660			
4-plane interactive	1024 x 1024	1024 x 1024	4
8-plane imaging	1024 x 1024	none	8
8-plane interactive	1024 x 1024	1024 x 1024	8
24-plane imaging	1024 x 1024	512 x 512	24
DN3000 M	1280 x 1024	768 x 1024	1
DN3000 C	1024 x 800	1024 x 224	4
DN3000 E	1024 x 800	1024 x 224	8

D.1 Limitations

The following GPR routines are only supported on DN550s, DN560s, DN600s, and DN660s:

- 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 an HDM bitmap on all devices except the DN570 is 224 x 224. On the DN570, the dimensions of the largest HDM bitmap is 224 x 219.

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

The gpr_$pgon_decomp_technique routine 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.

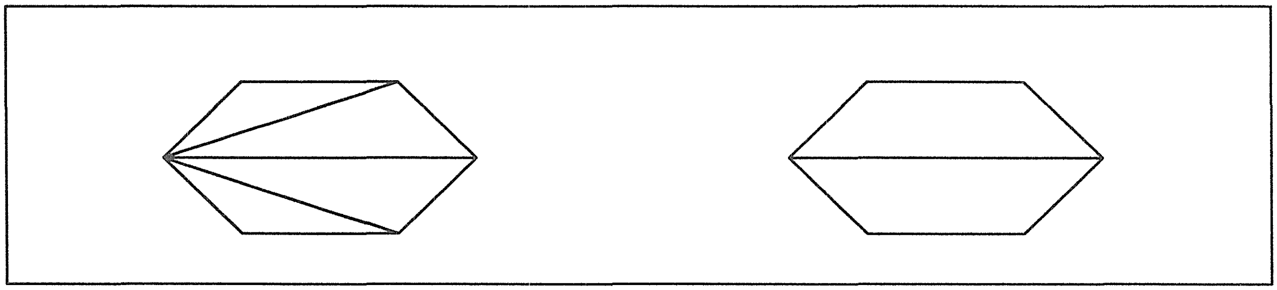


Figure E–1. A Polygon Decomposed into Triangles and Trapezoids

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, was 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.

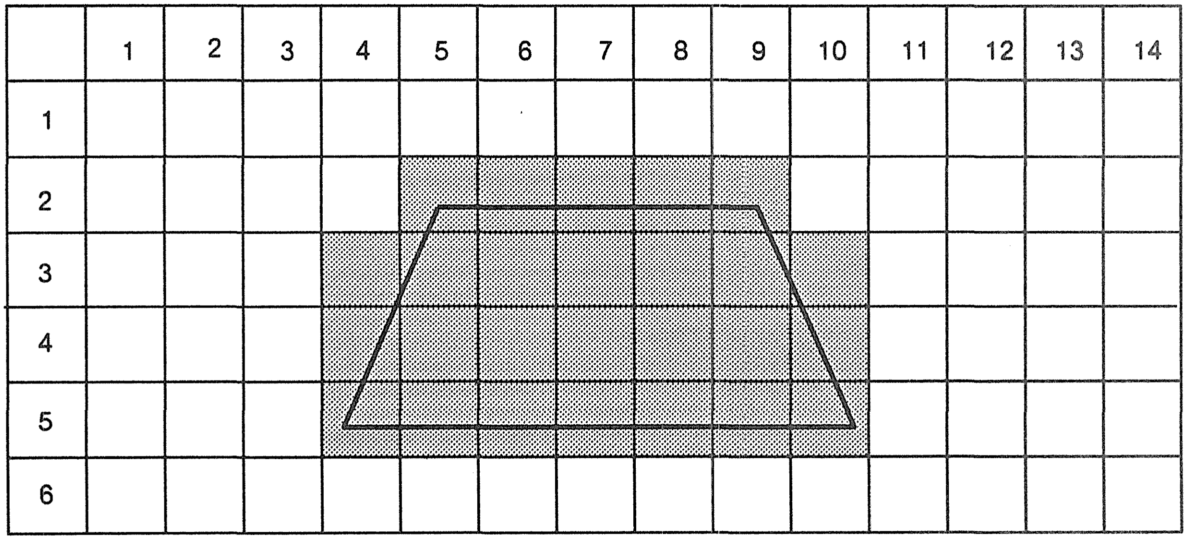


Figure E–2. The Pixels Rendered for a Trapezoid with the Trapezoid Technique

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.

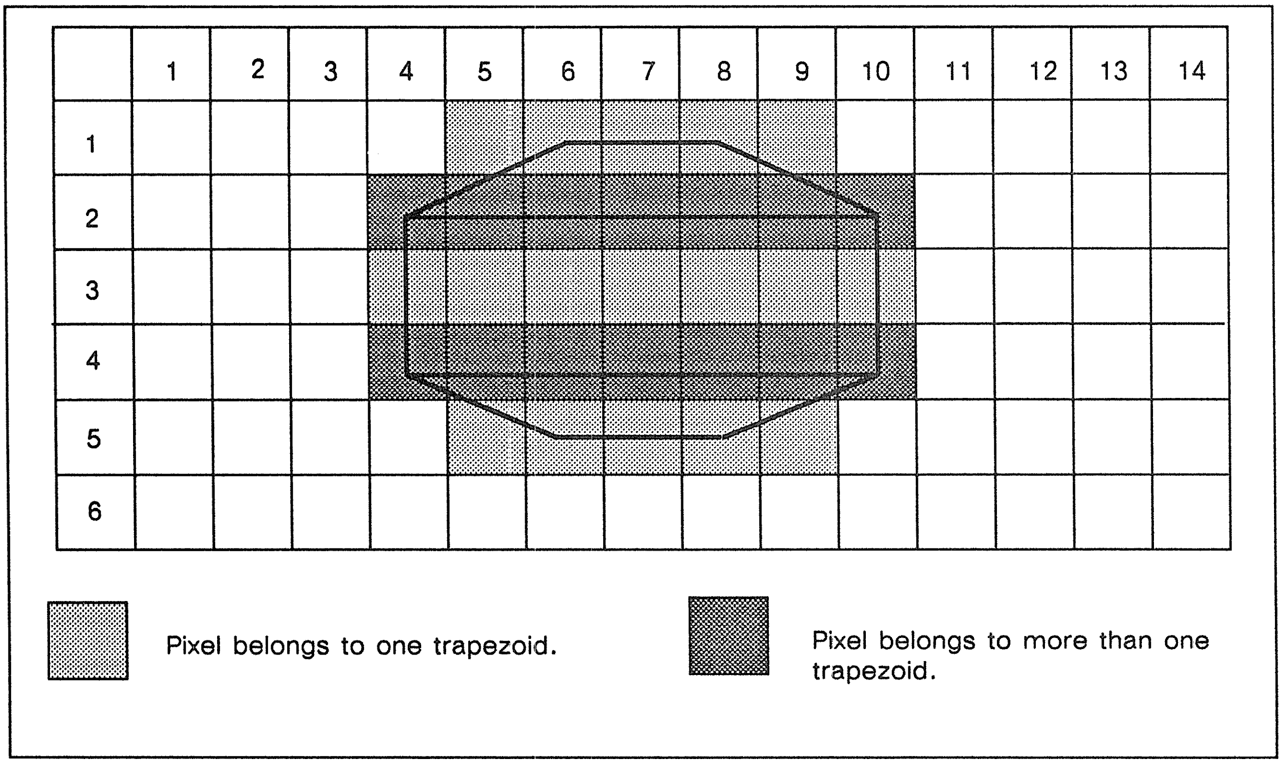

1
2
3
4
5
6
7
8
9
10
11
12
13
14
1
2
3
4
5
6
Pixel belongs to one trapezoid.
Pixel belongs to more than one trapezoid.

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

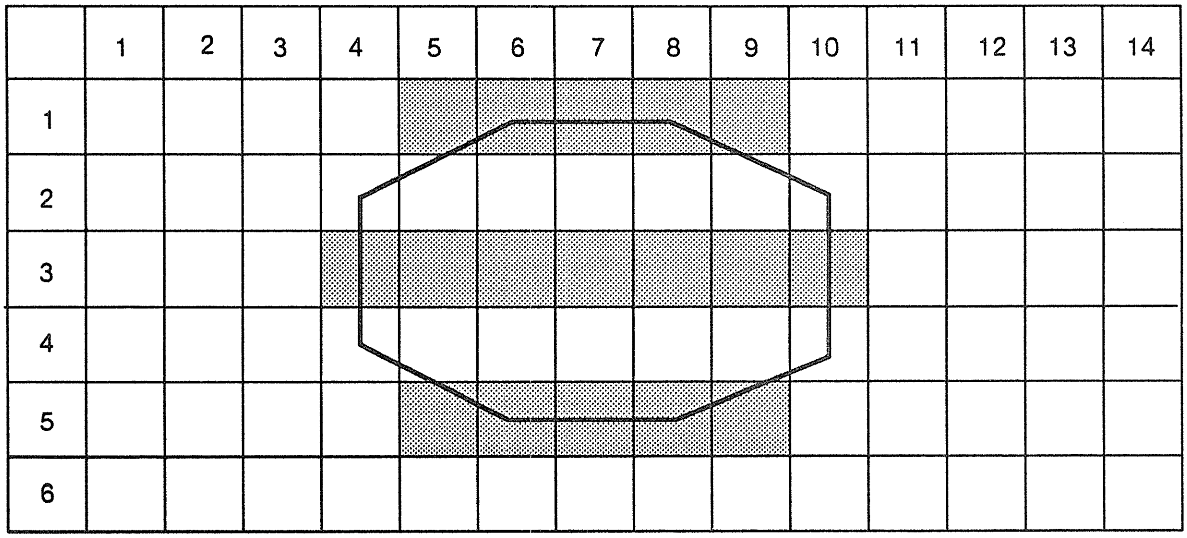


Figure E–4. Six–sided Polygon Decomposed and Rendered with the Trapezoid Technique. The Raster Operation was Set to XOR.

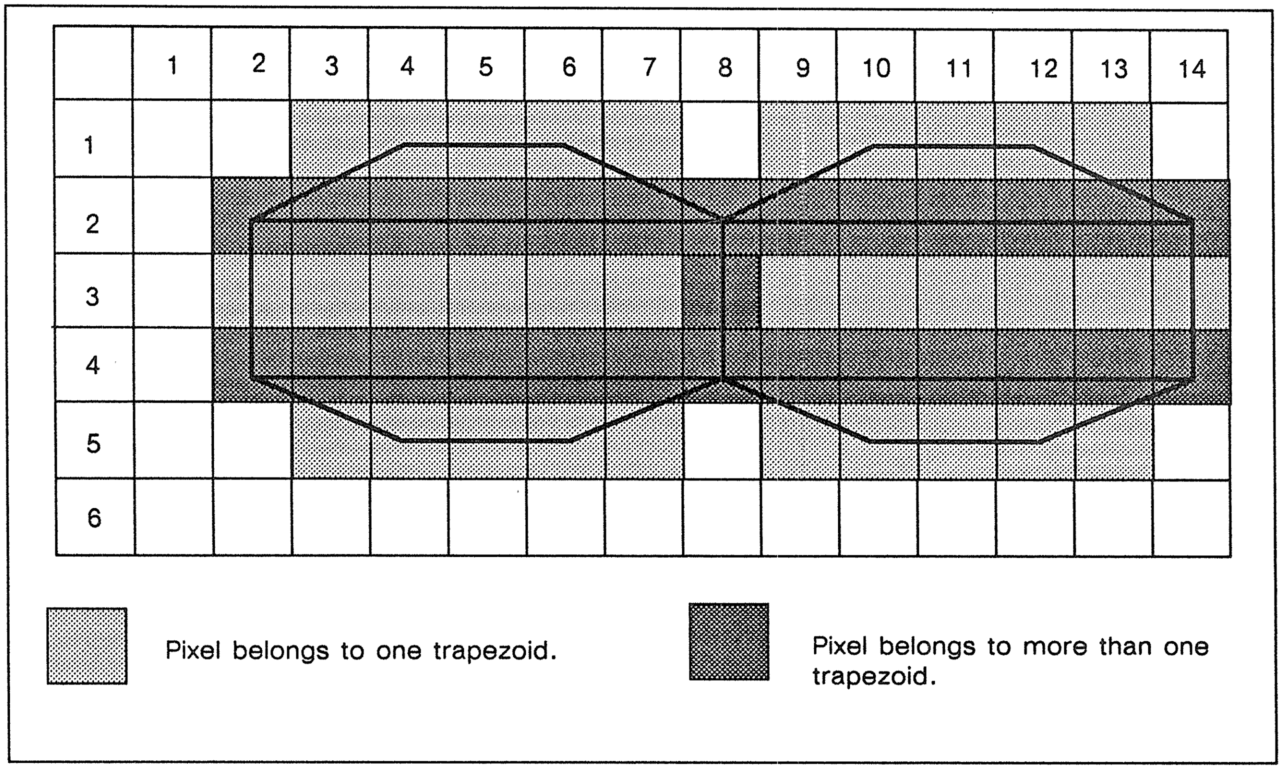

1 2 3 4 5 6 7 8 9 10 11 12 13 14
1 2 3 4 5 6
Pixel belongs to one trapezoid.
Pixel belongs to more than one trapezoid.

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

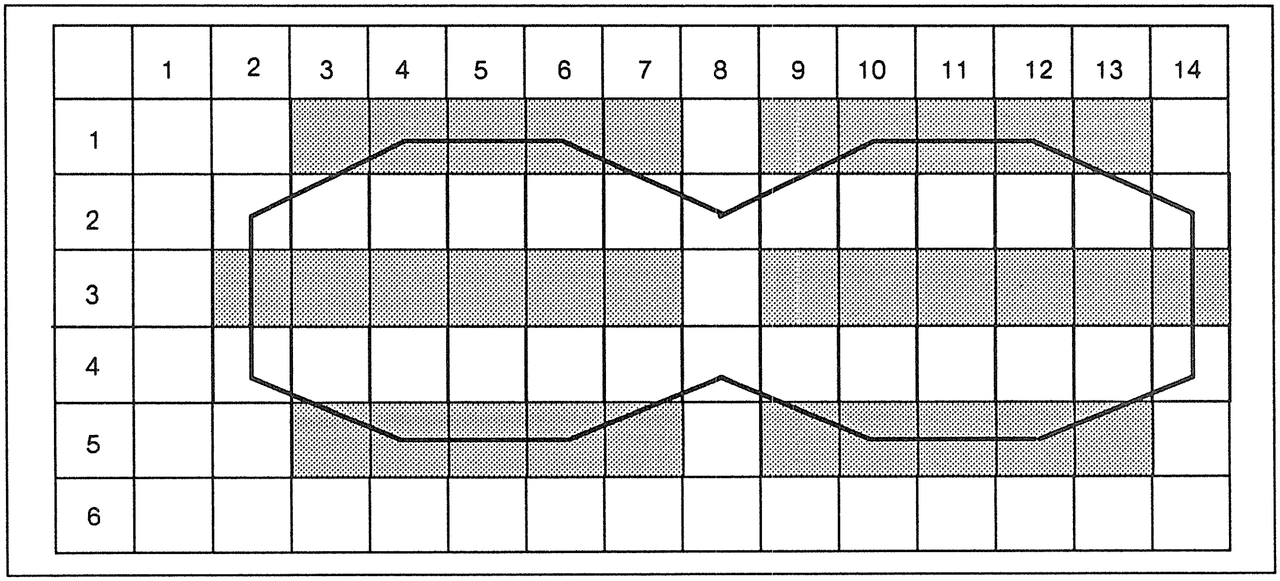


Figure E-6. Two Adjacent Six-sided Polygons Decomposed and Rendered with the Trapezoid Technique. The Raster Operation was Set to XOR.

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.

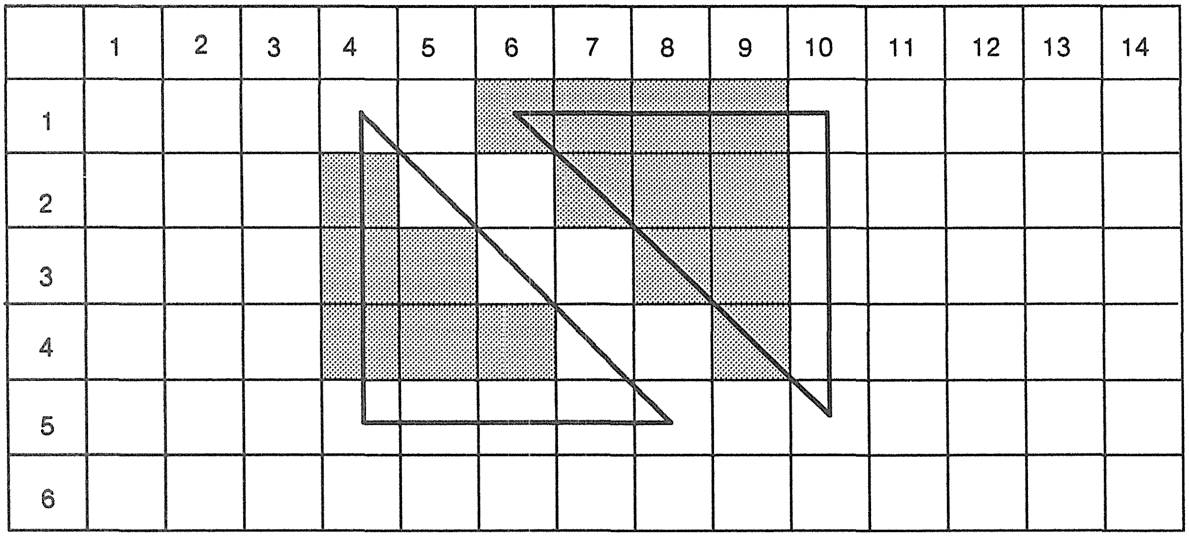


Figure E-7. Interior Pixels of Two Triangles Decomposed into Triangles

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).

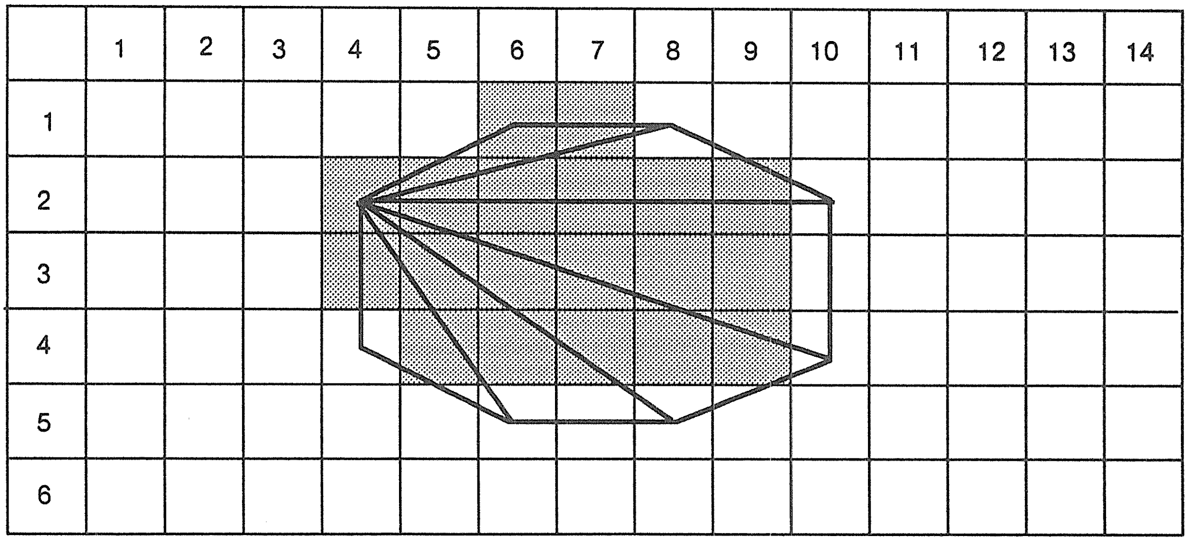


Figure E-8. Six-sided Polygon Decomposed into Triangles

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.

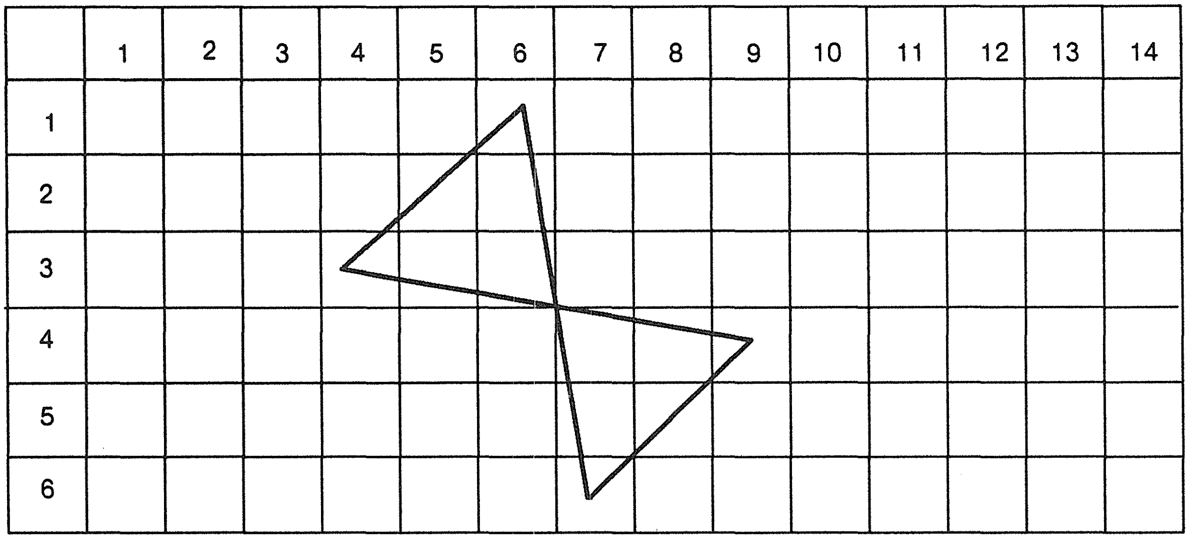


Figure E-9. A Sample Self-Intersecting Polygon

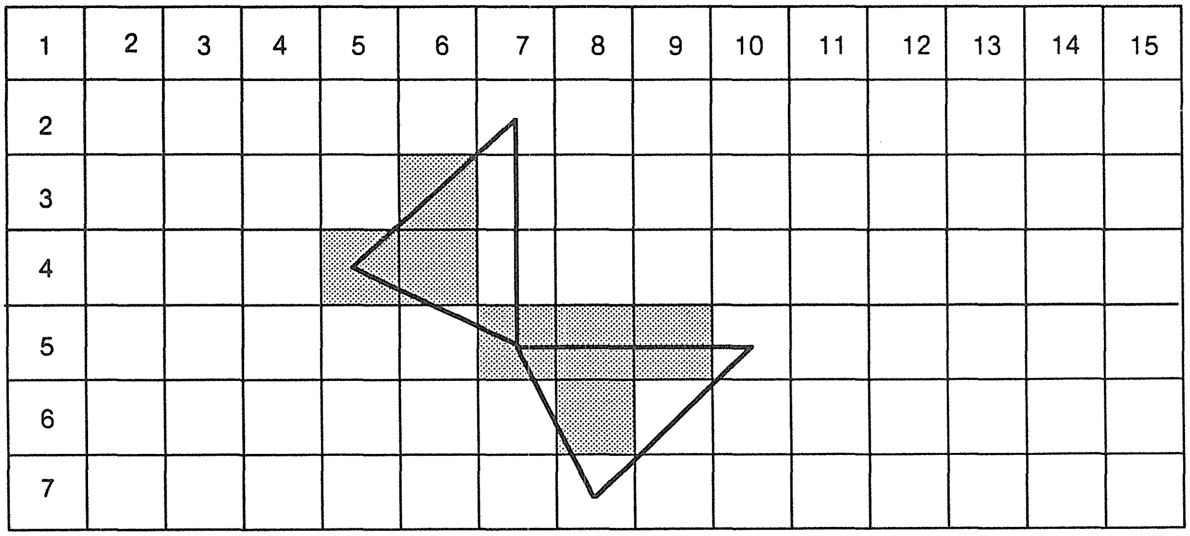


Figure E-10. The Pixels Rendered for the Polygon in Figure E-9 with the Triangle Technique

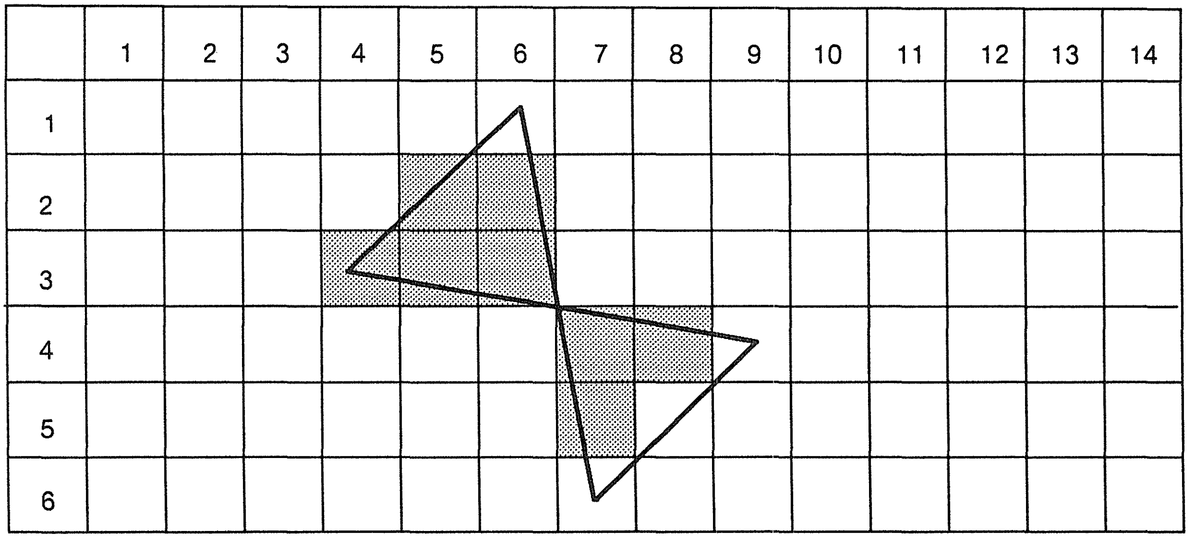


Figure E–11. The Pixels Rendered for the Polygon in Figure E–9 with the Render–Exact Technique

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).

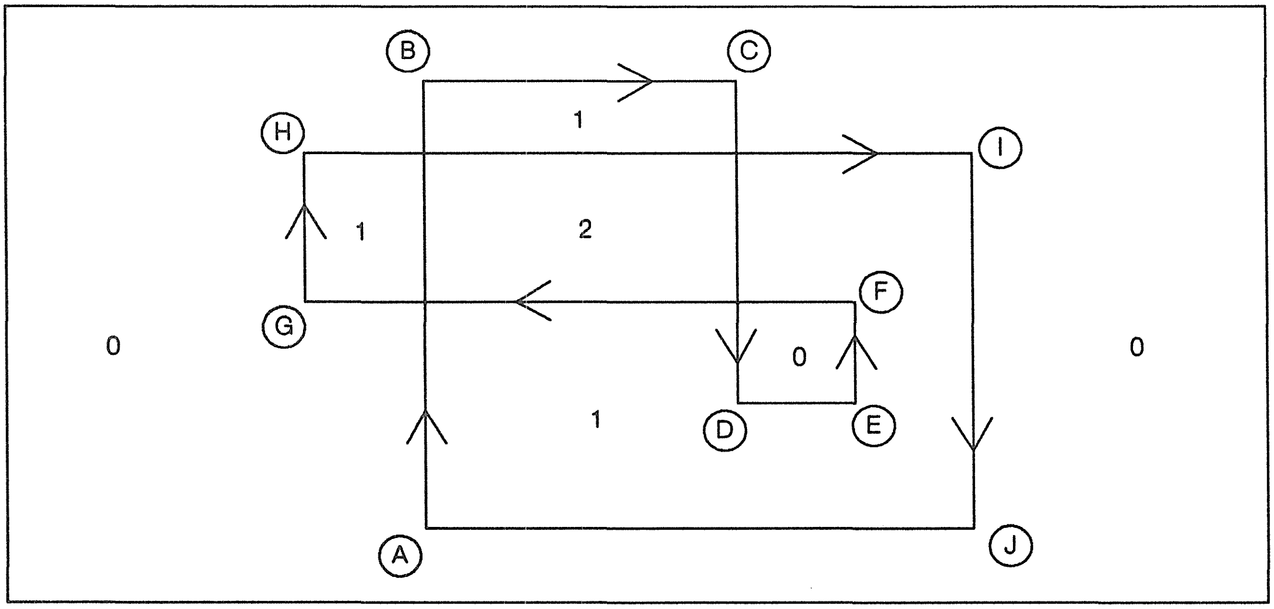


Figure E-12. The Winding Numbers of a Complex Polygon

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.

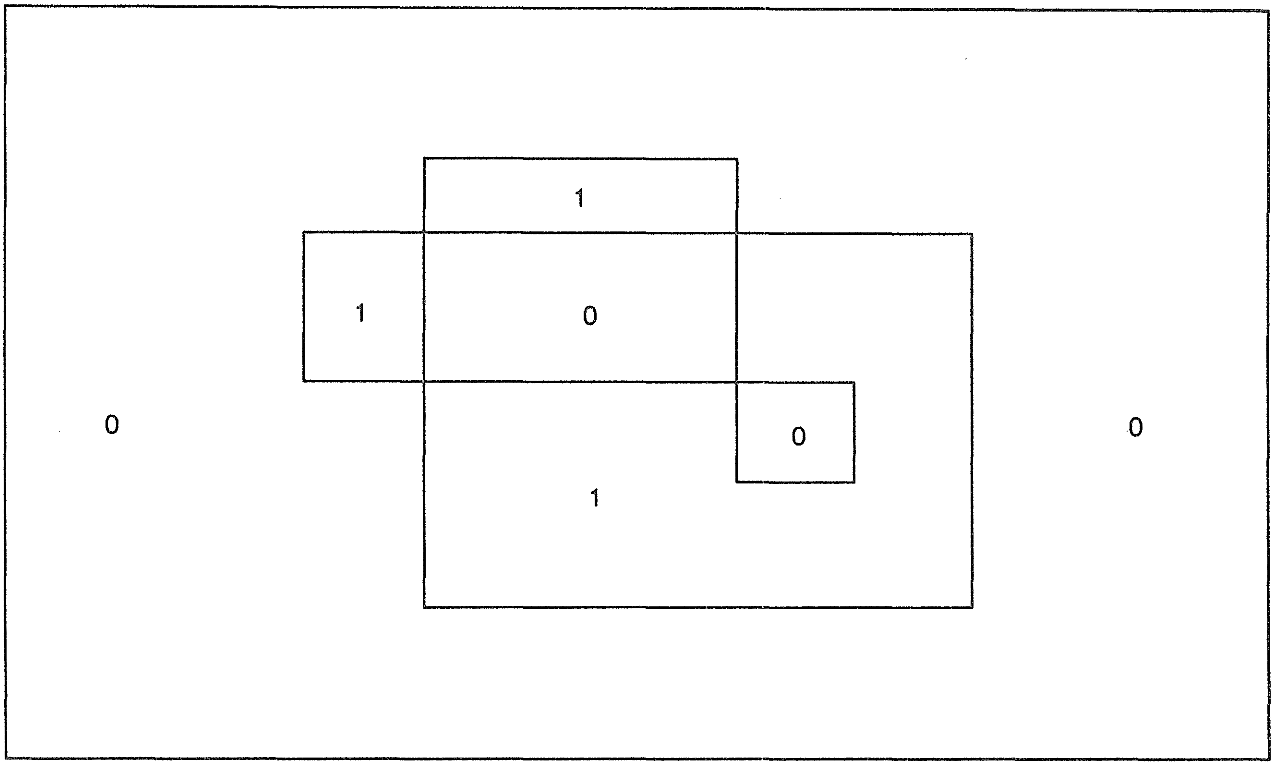


Figure E–13. The Parity Filling Numbers of a Complex Polygon

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.7.)

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
DN300/320/330	X	
DN400/420/460	X	
DN550/560	X	
DN570/580/590		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

Model	Decomposition Technique Used		
	Trapezoids	Triangles	Render Exact
DN300/320/330	software	software	software
DN400/420/460	software	software	software
DN550/560	microcode	software	software
DN570/580/590	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.

E.7.1 Program to Set the Decomposition Technique

This sample program is intended as a shell for future enhancements: the actual graphics application has been omitted. The purpose of this program 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.

```
PROGRAM set_the_decomposition_technique;
{This program sets the decomposition technique to gpr_$non_overlapping_tris
 on certain node types.  It demonstrates the gpr_$pgon_decomp_technique
 routine.}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/pfm.ins.pas';
VAR
   prim_set   : gpr_$rop_prim_set_t;  { The set of primitives that raster }
                                      { operations will affect.}

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
BEGIN
    init(gpr_$borrow);

{Set the decomposition technique on certain node types.}
    CASE display_characteristics.controller_type of
         gpr_$ctl_mono_1,
         gpr_$ctl_mono_2,
         gpr_$ctl_mono_4,
         gpr_$ctl_color_1,
         gpr_$ctl_color_4 : begin
            gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);
            check('establishing the pgon decomp technique');
                            end;
    END;

{Establish the set of raster operations for lines and fills.}
    prim_set := [gpr_$rop_line, gpr_$rop_fill];
    gpr_$raster_op_prim_set(prim_set, status);
    check('establishing the raster op prim set');

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

{Terminate the program.}
    gpr_$terminate(false,status);
END.
```

E.7.2 Sample Program to Draw a Polygon

This sample program demonstrates 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 program begins 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.

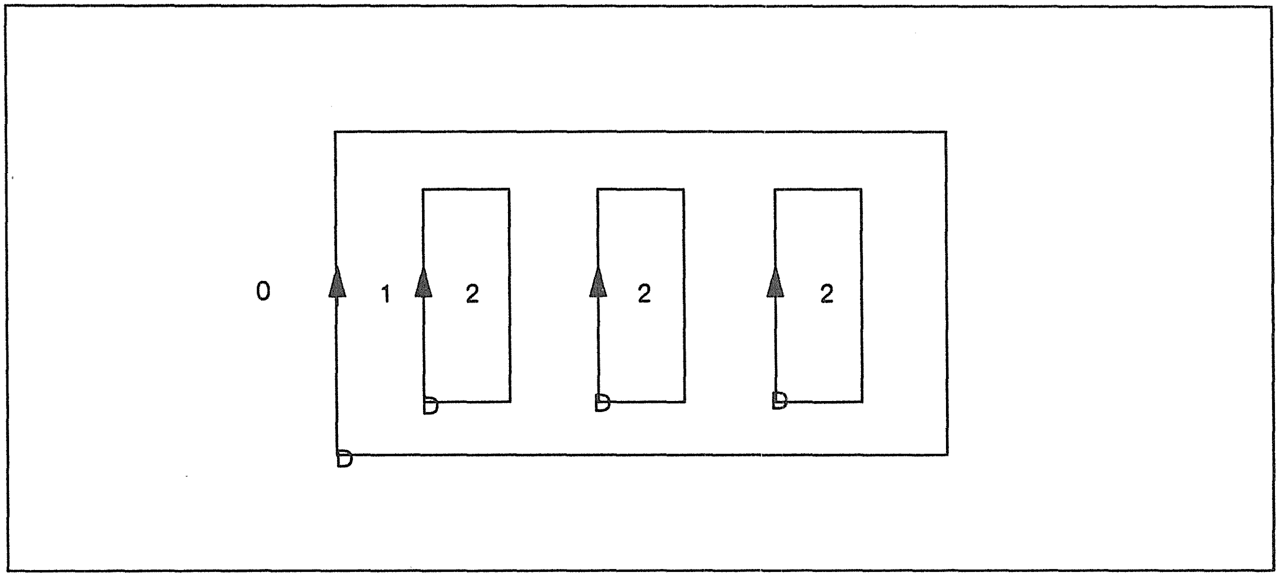


Figure E-14. How the System Fills a Polygon

```
PROGRAM triangle_technique;
{This program demonstrates the triangle technique.  It illustrates the
 gpr_$set_triangle_fill_criteria and gpr_$multitriangle routines.
}
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/pfm.ins.pas';

VAR
    prim_set              : gpr_$rop_prim_set_t;
    x, y                  : gpr_$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;

%include 'my_include_file.pas';{Contains the init, check, and pause routines.}
```

```
BEGIN   { Main }
    init(gpr_$direct);

{Set the decomposition technique on certain node types.}
    CASE (display_characteristics.controller_type) of
         gpr_$ctl_mono_1,
         gpr_$ctl_mono_2,
         gpr_$ctl_mono_4,
         gpr_$ctl_color_1,
         gpr_$ctl_color_4 : begin
            gpr_$pgon_decomp_technique(gpr_$non_overlapping_tris, status);
            check('establishing the pgon decomp technique');
                            end;
    END;

    prim_set := [gpr_$rop_line, gpr_$rop_fill];
    gpr_$raster_op_prim_set(prim_set, status);
    check('establish the raster op prim set');

    window.window_base.x_coord := 200;
    window.window_base.y_coord := 200;
    window.window_size.x_size  := 1;
    window.window_size.y_size  := 1;

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

    gpr_$read_pixels(window,pixel_array, status);
    check('performing a read_pixels');

    x := 50;
    y := 600;
    gpr_$start_pgon(x, y, 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);

    x := 150;
    y := 500;
    gpr_$start_pgon(x, y, 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_$pgon_polyline(x_array, y_array, n_positions, status);
   x := 350;
   y := 500;
   gpr_$start_pgon(x, y, 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);

   x := 550;
   y := 500;
   gpr_$start_pgon(x, y, 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('performing a pgon polyline');

   winding_set.wind_type := gpr_$parity;
   gpr_$set_triangle_fill_criteria(winding_set,status);
   check('performing a set triangle fill criteria');

   list_size := 30;
   gpr_$close_return_pgon_tri(list_size, t_list, n_triangles, status);
   check('performing a close return pgon tri');

{Draw the triangles with a parity fill.}
   gpr_$multitriangle(t_list, n_triangles, status);

{Keep image displayed on screen for five seconds, then clear the screen.}
   pause(5.0);
   gpr_$clear(pixel_array[1], status);

   winding_set.wind_type := gpr_$nonzero;
   gpr_$set_triangle_fill_criteria(winding_set,status);
   check('performing a set triangle fill criteria');

{Draw the triangles with a nonzero fill.}
   gpr_$multitriangle(t_list, n_triangles, status);
```

```
    check('performing a gpr_$multitriangle with a nonzero fill');

{Keep image displayed on screen for five seconds, then clear the screen.}
    pause(5.0);
    gpr_$clear(pixel_array[1], status);

    winding_set.wind_type := gpr_$specific;
    winding_set.winding_no := 2;
    gpr_$set_triangle_fill_criteria(winding_set,status);
    check('performing a set triangle fill criteria');

{Draw the triangles with a specific winding number fill.}
    gpr_$multitriangle(t_list, n_triangles, status);
    check('performing a gpr_$multitriangle');

{Pause 5 seconds, release the display, and Terminate the program.}
    pause(5.0);
    gpr_$release_display(status);
    gpr_$terminate(FALSE,status);
END.
```

Appendix F

Imaging Mode

The DN550, DN560, DN600, and DN660 class nodes support the following two different color modes of operation:

- Interactive mode.
- Imaging mode.

All other nodes support only interactive mode.

Chapter 6 described interactive mode, which supports all the GPR color operations.

In this Appendix, we describe imaging mode. The advantage of imaging mode is that it allows your program to use more planes of display memory than interactive mode does. Therefore, more colors can be displayed simultaneously in imaging mode than in interactive mode. The disadvantage of imaging mode is that you cannot use the GPR draw, fill, or text routines in imaging mode. Thus, it is *much* harder to program in imaging mode.

From imaging mode, you can only use the calls listed below:

gpr_$inq_imaging_format
gpr_$select_color_frame
gpr_$set_color_map
gpr_$set_imaging_format
gpr_$terminate
gpr_$write_pixels

F.1 What Is Imaging Mode?

You use imaging mode to increase the number of colors that can simultaneously be displayed on a DN550/560/600/660 monitor. For example, in the default mode (interactive mode), a node with two boards of display memory supports four planes of display memory. However, in imaging mode, a node with two boards of display memory supports eight planes of display memory. Similarly, a node with three boards of display memory supports eight planes of display memory in interactive mode and 24 planes in imaging mode. These results are summarized in Table F-1.

Table F-1. Effect of Display Memory and Mode on Color for DN550/560/600/660 Nodes

Number of boards of display memory	Mode	Planes	Maximum number of colors that can be simultaneously displayed
2	interactive	4	16
	imaging	8	256
3	interactive	8	256
	imaging	24	16.7 million

For some nodes, going to imaging mode from interactive mode reduces the amount of visible and/or hidden display memory (as noted in Appendix D).

F.2 Selecting Between Interactive Mode and Imaging Mode

By default, your node is in interactive mode. Use the gpr_$set_imaging_format to change to imaging mode, or to change back to interactive mode from imaging mode. You must be in borrow mode in order to call gpr_$set_imaging format. (Imaging mode is supported only in borrow mode.)

To find out which mode you are in (i.e., interactive or imaging) call the gpr_$inq_imaging_format routine.

It is possible to write a program that displays figures in interactive mode and then to switch to imaging mode in the middle of the program. 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.

F.3 Writing to a Bitmap in Imaging Mode

You cannot call any GPR draw, fill, or text routines in imaging mode. Therefore, it is rather difficult to write to an imaging mode bitmap. The best way to write pixel values from a pixel array into an imaging mode bitmap is to call gpr_$write_pixels. (You cannot, however, call gpr_$read_pixels.)

F.4 Using Color in Imaging Mode

The color map works in imaging mode just as it does in interactive mode for 8- and 24-plane displays. From imaging mode, you can adjust the color map with the gpr_$set_color_map routine just as you would from interactive mode. See Chapter 6 for a description of color and the color map.

Note the distinction between the 24-plane true-color DN590 and the 24-plane imaging mode of the DN550/560/600/660. For both machines, color lookup through a 256x3 color map matrix is identical. However, on the DN590, you can draw lines of a particular color by calling gpr_$set_draw_value and gpr_$line, but in a 24-plane imaging mode, you would have to draw the line with a gpr_$write_pixels routine.

F.5 Using gpr_$select_color_frame

The system does permit you to select a frame with gpr_$select_color frame in 24-plane imaging mode just as you would from 8-plane interactive mode.

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

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 map entry

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

Color table

See Color map.

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 map 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 main memory and performs graphic operations there, bypassing the display.

Picture element

A single element of a two-dimensional displayed image or of a two-dimension allocation 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.

Index

The letter *f* means "and the following page"; the letters *ff* mean "and the following pages". Symbols are listed at the beginning of the index. Entries in color indicate procedural information.