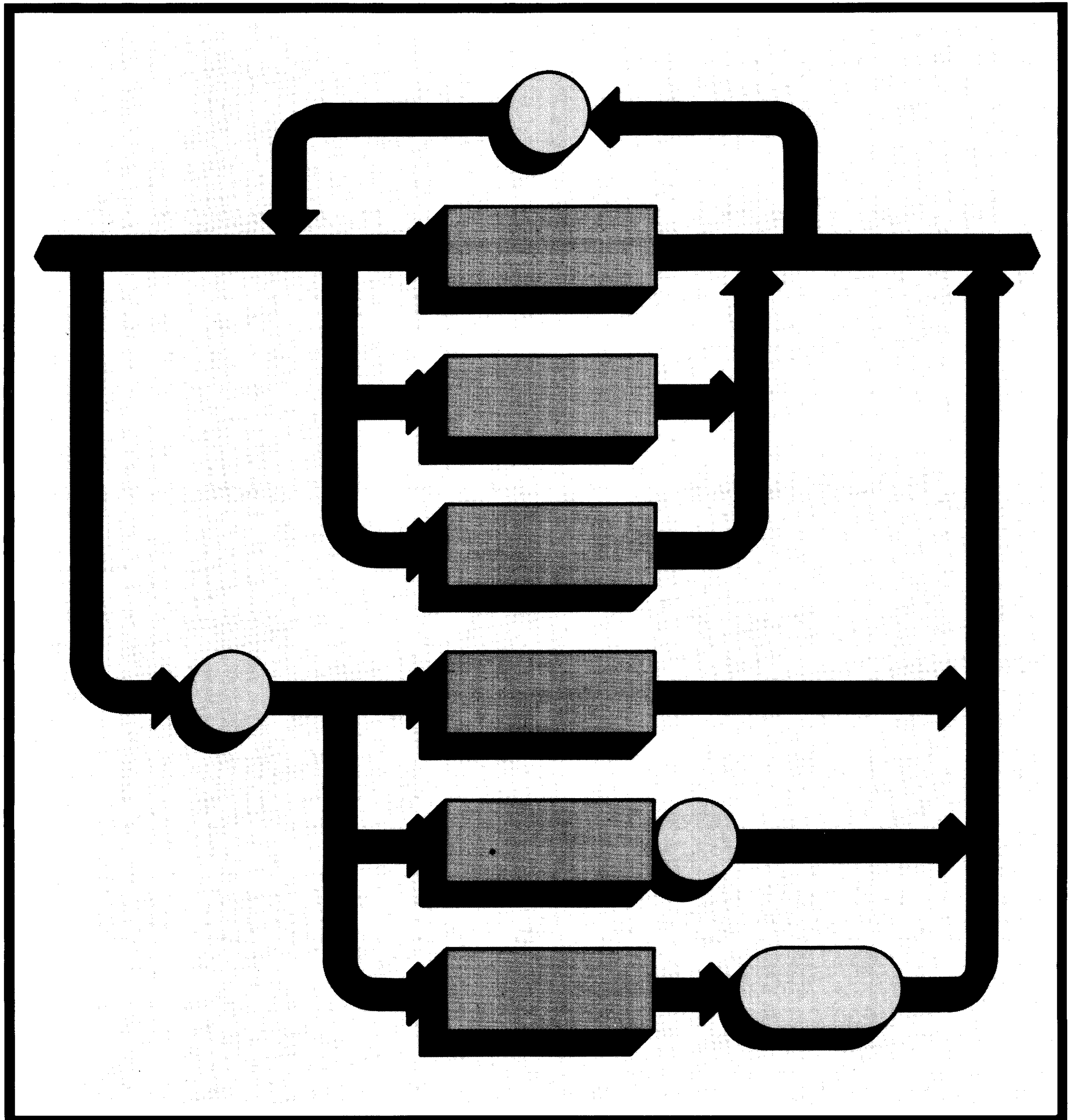
HP 9000 Series 200 Computers

HEWLETT PACKARD

Pascal 3.0 Graphics Techniques



Pascal 3.0 Graphics Techniques

for the HP 9000 Series 200 Computers

Manual Part No. 98615-90035

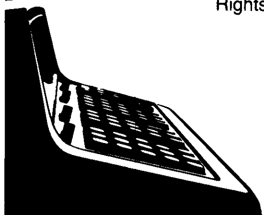
© Copyright 1984, 1985, Hewlett-Packard Company.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).



Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

June 1984...First Edition
October 1984...Update
November 1984...First Edition with previous updates merged.
March 1985...Update
This update added references to using the HP 46060 Mouse for graphics input.

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard computer system products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of shipment.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR CONSEQUENTIAL DAMAGES.

HP 9000 Series 200

For the HP 9000 Series 200 family, the following special requirements apply. The Model 216 computer comes with a 90-day, Return-to-HP warranty during which time HP will repair your Model 216, however, the computer must be shipped to an HP Repair Center.

All other Series 200 computers come with a 90-Day On-Site warranty during which time HP will travel to your site and repair any defects. The following minimum configuration of equipment is necessary to run the appropriate HP diagnostic programs: 1) ½ Mbyte RAM; 2) HP-compatible 3½" or 5¼" disc drive for loading system functional tests, or a system install device for HP-UX installations; 3) system console consisting of a keyboard and video display to allow interaction with the CPU and to report the results of the diagnostics.

To order or to obtain additional information on HP support services and service contracts, call the HP Support Services Telemarketing Center at (800) 835-4747 or your local HP Sales and Support office.

* For other countries, contact your local Sales and Support Office to determine warranty terms.

Table of Contents

Chapter 1: Introduction to Graphics

Welcome . . . 1
Why Graphics? . . . 2
Using the Graphics Library . . . 4
    INCLUDE files . . . 5
The Graphics Programs . . . 6
    Customizing the Programs for Your System . . . 6
    Drawing Lines . . . 7
    Scaling . . . 9
    Setting the Aspect Ratio . . . 11
    Defining a Viewport . . . 13
        Virtual Coordinates and World Coordinates . . . 13
        Specifying the Viewport . . . 14
    Labelling a Plot . . . 15
    Setting the Character Size . . . 16
    Centering Labels . . . 17
    Setting the Label's Direction . . . 17
    Bold Labels . . . 19
    Axes and Tick Marks . . . 20
    Clipping Lines . . . 23
        A User-Defined Clipping Algorithm . . . 24
    Labelling Axes . . . 25

Chapter 2: Miscellaneous Graphics Concepts

Setting the Display Limits . . . 33
More on Defining a Viewport . . . 34
Calculating Window Limits . . . 36
Drawing a Window Frame . . . 37
Turning Displays On and Off . . . 39
Conversion Between Coordinate Systems . . . 40
More on Labelling a Plot . . . 43
    The Character Cell . . . 43
    Setting Character Size . . . 45
    Setting the Label's Direction . . . 48
    Justifying Labels . . . 50
CRT Drawing Modes . . . 55
Faster Drawing Procedures . . . 56
Selecting Line Styles . . . 57
Isotropic Scaling . . . 59
Axes and Grids . . . 62

Logarithmic Plotting 64
Homemade Mathematical Functions 64
Taking a Number to a Power 64
The Logarithm to Any Base 65
Back to Logarithmic Axes.... 65
Storing and Retrieving Images 68
Data-Driven Plotting 71
Many Lines in One Step 71
Drawing Multi-Line Objects 72
What's in a Polygon? 74
When to Use Which Polygon? 74
Polygon Filling 74
Shading Graphs 78
Highlighting Data Curves 79

Chapter 3: External Graphics Displays and Plotters

Selecting a Plotter 81
Dumping Raster Images 82
External Color Displays 84
External Plotter Control 85
Controlling Pen Speed 85
Controlling Pen Acceleration 86
Controlling Pen Force 86

Chapter 4: Interactive Graphics

Introduction 87
Characterizing Graphic Interactivity 88
Selecting Input Devices 89
Single Degree of Freedom 89
Non-separable Degrees of Freedom 92
Separable Degrees of Freedom 92
All Continuous 92
All Quantizable 93
Mixed Modes 93
Echoes 94
The Built In Echo 94
Rubber Echoes 97
Tablets and Aspect Ratios 98

Chapter 5: Color Graphics

Color! 99
The DGL Color System 99
  Color As An Attribute 99
  The Color Table 100
  Default Colors 100
    The Primary Colors 100
    The Business Colors 100
  Monochromatic Defaults 101
  If You Don't Like the Defaults 101
Models for Color Specification 102
  The RGB Model (Red, Green, Blue) 102
  The HSL Model (Hue, Saturation, Luminosity) 103
  Which Model? 107
Color Spaces 109
  Primaries and Color Cubes 109
    The RGB Color Cube 110
    The CMY Color Cube 111
  The HSL Color Cylinder 112
Reality Intrudes 114
Plotters 114
Frame Buffers 115
  Frame Buffer Depth 115
  Faking More Colors From a Frame Buffer 116
    Dithering 117
    Creating a Dithered Color 118
  If You Need More Colors 121
  Frame Buffer Contents 121
The Model 236 Color Computer Color System 122
  The Color Map 122
  True User Definable Color 123
  Retroactive Color Changes 123
  If You Need More Than 16 Colors 123
    Optimizing For Dithering 124
Resolution and Color Models 126
  RGB Resolution 126
  HSL Resolution 126
Writing Modes and Color 127
  Dominant Writing 128
  Non-Dominant Writing 128
  Erasing 128
  Complementary Writing 128
    Making Sure Echoes Are Visible 129
  Drawing Modes and the Frame Buffer 129
  Special Considerations 132
    Text 132
    Polygons 132

Effective User of Color . . . 133
Seeing Color . . . 133
It's All Subjective, Anyway . . . 133
Mixing Colors . . . 134
Designing Displays . . . 134
Objective Color Use . . . 135
Color Blindness . . . 135
Subjective Color Use . . . 135
Choosing Colors . . . 135
Psychological Color Temperature . . . 136
Cultural Conventions . . . 136
Reproducing Color Graphics . . . 137
Color Gamuts . . . 137
Color Hard Copy . . . 137
Photographing the CRT . . . 138
Plotting and the CRT . . . 138
Color References . . . 139

Appendix A: Listings of Example Programs

AxesGrid . . . 142
BAR_KNOB . . . 149
BAR_KNOB2 . . . 152
CharCell . . . 157
COLOR . . . 158
CsizeProg . . . 165
DataPoint . . . 166
DrawMdPrg . . . 166
FillProg . . . 169
FillGraph . . . 170
GStorProg . . . 171
IsoProg . . . 180
JustProg . . . 186
LdirProg . . . 190
LOCATOR . . . 191
LogPlot . . . 194
MarkrProg . . . 196
PLineProg . . . 197
PolyProg . . . 198
SinAspect . . . 199
SinAxes1 . . . 200
SinAxes2 . . . 204
SinClip . . . 209
SinLabel1 . . . 213
SinLabel2 . . . 214
SinLabel3 . . . 215
SinLine . . . 216
SinViewpt . . . 216
SinWindow . . . 217

Appendix B: Graphics Procedure Reference

AWAIT_LOCATOR 220
CLEAR_DISPLAY 225
CONVERT_WTODMM 226
CONVERT_WTOLMM 227
DISPLAY_FINIT 228
DISPLAY_INIT 232
DISPLAY_TERM 237
GRAPHICSERROR 238
GRAPHICS_INIT 240
GRAPHICS_TERM 241
GTEXT 242
INPUT_ESC 244
INQ_COLOR_TABLE 247
INQ_PGN_TABLE 249
INQ_WS 251
INT_LINE 258
INT_MOVE 260
INT_POLYGON 262
INT_POLYGON_DD 265
INT_POLYLINE 269
LOCATOR_INIT 271
LOCATOR_TERM 274
MAKE_PIC_CURRENT 275
MARKER 276
MOVE 277
OUTPUT_ESC 278
POLYGON 283
POLYGON_DEV_DEP 286
POLYLINE 290
SAMPLE_LOCATOR 292
SET_ASPECT 294
SET_CHAR_SIZE 296
SET_COLOR 297
SET_COLOR_MODEL 300
SET_COLOR_TABLE 302
SET_DISPLAY_LIM 306
SET_ECHO_POS 309
SET_LINE_STYLE 311
SET_LOCATOR_LIM 315

SET_LINE_WIDTH . . . 319
SET_PGN_COLOR . . . 320
SET_PGN_LS . . . 323
SET_PGN_STYLE . . . 327
SET_PGN_TABLE . . . 328
SET_TEXT_ROT . . . 331
SET_TIMING . . . 332
SET_VIEWPORT . . . 334
SET_WINDOW . . . 336
Module Dependency Table . . . 339

Subject Index

Introduction to Graphics

Chapter 1

Welcome

One of the most exciting features of your Series 200 computer is its graphics capability. It is much easier to grasp the trends, relative sizes or quantities represented by data if it is presented in a graphical form, as opposed to tabular form.

This manual will introduce you to the set of graphics routines in the Series 200 Device-independent Graphics Library (DGL) graphics package. The goals of the DGL package are:

1. As its name implies, it is a device-independent package. Thus, programs running on one computer or implementation should transport to another computer or implementation of DGL with a minimum of conversion effort.
2. It is reasonably small. DGL is not meant to be an exhaustive library containing routines to do all conceivable grapics operations, but it gives you enough capability to develop them yourself.
3. DGL is quite fast. Many of its operations press the capabilities of the Motorola 68000 processor, the CPU in the Series 200 machines.

This manual is meant to teach you how to use the routines incorporated into DGL to produce highly readable and visually acceptable output. The manual assumes you are familiar with the Pascal programming language, and that you have access to a *Pascal 3.0 Workstation System Manual*, a *Pascal 3.0 Procedure Library* manual, and the textbook *An Introduction to Programming and Problem Solving with Pascal*, and that you will look up any programming/syntactic topics you don't understand.

Most of the demonstration programs and routines in the next three chapters of this manual are stored for your convenience on the `DGLPRG:` disc which was shipped with this manual. You are encouraged to run these programs as you are reading the manual, as they will make understanding the concepts much easier.

Note

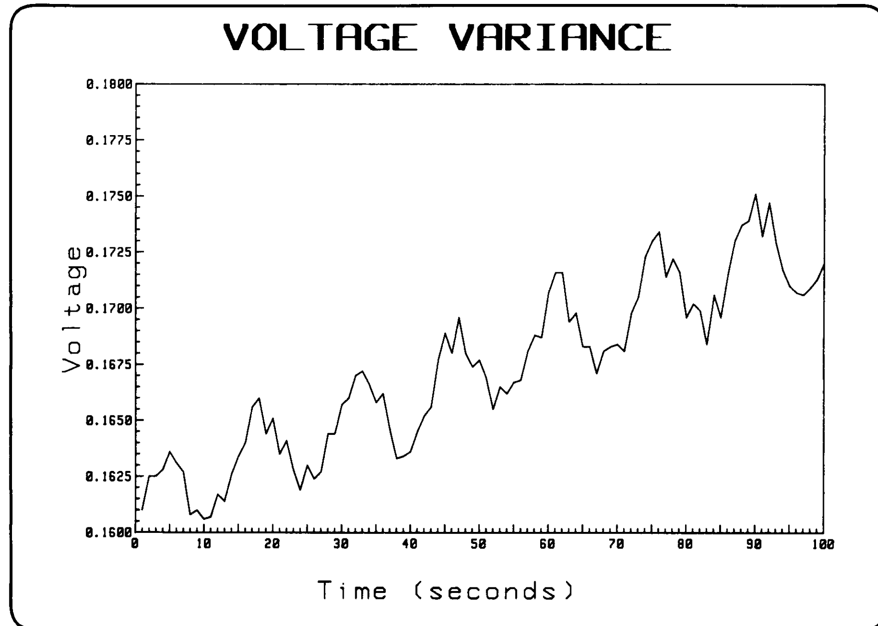
The demonstration programs and routines on the `DGLPRG:` disc are for the purpose of instruction only. They are not part of the DGL package, and as such, they are not covered by any warranty, expressed or implied. Hewlett-Packard shall not be liable for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these routines.

Why Graphics?

Below is some data. As quickly as you can, determine if its overall trend is steady, rising or falling. Are there any periodic motions to it? If so, how many cycles are represented in the one hundred points?

Voltage Variance		Voltage Variance	
Time (sec)	Voltage	Time (sec)	Voltage
1	0.1610	51	0.1669
2	0.1625	52	0.1655
3	0.1625	53	0.1665
4	0.1628	54	0.1662
5	0.1636	55	0.1667
6	0.1631	56	0.1668
7	0.1627	57	0.1681
8	0.1608	58	0.1688
9	0.1610	59	0.1687
10	0.1606	60	0.1707
11	0.1607	61	0.1716
12	0.1617	62	0.1716
13	0.1614	63	0.1694
14	0.1626	64	0.1698
15	0.1634	65	0.1683
16	0.1640	66	0.1683
17	0.1656	67	0.1671
18	0.1660	68	0.1681
19	0.1644	69	0.1683
20	0.1651	70	0.1684
21	0.1635	71	0.1681
22	0.1641	72	0.1698
23	0.1628	73	0.1705
24	0.1619	74	0.1723
25	0.1630	75	0.1730
26	0.1624	76	0.1734
27	0.1627	77	0.1714
28	0.1644	78	0.1722
29	0.1644	79	0.1716
30	0.1657	80	0.1696
31	0.1660	81	0.1702
32	0.1670	82	0.1699
33	0.1672	83	0.1684
34	0.1666	84	0.1706
35	0.1658	85	0.1696
36	0.1662	86	0.1715
37	0.1646	87	0.1730
38	0.1633	88	0.1737
39	0.1634	89	0.1739
40	0.1636	90	0.1751
41	0.1645	91	0.1732
42	0.1652	92	0.1747
43	0.1656	93	0.1729
44	0.1677	94	0.1717
45	0.1689	95	0.1710
46	0.1680	96	0.1707
47	0.1696	97	0.1706
48	0.1680	98	0.1709
49	0.1674	99	0.1713
50	0.1677	100	0.1720

A wise old computer programmer once said, "A graphical output is equivalent to 1K words of text." He was right. Unless both hemispheres of your brain are hyperdeveloped, it probably took a minute or two to answer each of the previous questions. Below is a graph of the data in the table. Observe that the graphical nature of the output makes what the data is doing much clearer. This clarity and understandability at a glance is what computer graphics is all about.



A progressive example of how this plot was created is given through the rest of this chapter. Each installment demonstrates more of the graphics routines available. The successive plots, all representing the same data, become clearer and clearer as we learn some of the concepts of computer graphics and how to implement them with the routines available to us.

Using the Graphics Library

To run the demonstrations programs in this manual, you must use the DGL routines contained in the GRAPHICS library file on the LIB: disc. The first step, then, is to make these libraries accessible to the demonstration programs at the appropriate times.

There are two times when the GRAPHICS modules need to be accessible:

- When the program is compiled, and
- When the program is loaded.

The simplest way to make the GRAPHICS library accessible during compilation and loading is to use the **What** command to make GRAPHICS the system library. To do this:

1. At the Main Command Level, press (W) to invoke the **What** command.
2. Press (B) to indicate you want to change the system library setting, and type the complete file specification for the GRAPHICS library file. Be sure to type a period after the file name, to prevent the system from appending a suffix to the name. For example, if the GRAPHICS file is still on the LIB: disc, you would type:

   `LIB:GRAPHICS.` (Return)
3. Press (Q) to exit from the **What** command. When you begin compiling and running the demonstration programs, make sure the GRAPHICS library file is on-line[1].

Note

If you have plenty of memory in your computer, you can speed *compilation* by copying the GRAPHICS file into a memory (RAM:) volume of about 400 blocks. Be sure to use the **What** command to change the system library to RAM:GRAPHICS if you do this. You can also speed program *execution* by permanently loading the GRAPHICS file with the **Permanent** command.

---

1 "On-line" means that it is accessible at that moment. This could mean either that the library is in a memory volume, or the library is on a disc and the disc is currently in a drive.

INCLUDE files

In many of the following programs, there is a compiler directive called INCLUDE. This causes the compiler to access the specified file, compile the contents as if it were in the original file, and when the end of the file is reached, return to the original file and continue compilation.

One advantage to INCLUDE files is that many programs can use the *same* file, not merely copies of the file. This makes it much easier to make modifications to the routines, because only one copy of the routine need be changed. If the routine had been physically copied into each program that used it, every occurrence of it would have to be individually changed.

The INCLUDE directives used in the program files assume there is a volume on-line which contains the text files for all the necessary inclusions. Again, if you have enough memory, the INCLUDE process could be speeded up tremendously by placing the necessary files in a memory volume.

Here is some information to help you define how large the "enough memory," referred to in the previous paragraph, is. Below is a list of files at least some of which you will probably want to permanently load (the main advantage to permanently loading is very fast access) and the amount of memory they consume. The approximate file sizes are expressed in 256-byte blocks.

File Name		Approximate File Size
EDITOR	(subsystem)	240 blocks
FILER	(subsystem)	230 blocks
COMPILER	(subsystem)	835 blocks
LIBRARIAN	(subsystem)	225 blocks
LIBRARY	(library)	65 blocks
IO	(library)	245 blocks
GRAPHICS	(library)	810 blocks

You must also take into account any memory volumes you have defined, and the size of the program you are dealing with, etc.

The Graphics Programs

All of the following plots use Cartesian (rectangular) coordinates: "X" specifies the left-right distance (with values increasing as you go to the right), and "Y" specifies the down-up distance (with values increasing as you go up).

In the programs in this chapter and the next, each program name is identical to the file name which contains it. It is not mandatory that the program name is the same as the file name, but it helps to find the file.

All the examples that follow get the Y-value from a function called `DataPoint`. This function, given an X value, merely returns the appropriate Y value each time it is invoked. You could just as well be reading values from a voltmeter, temperature sensor, anemometer, or any other device that you can connect to a computer. Since this function does not change from example to example, and since it represents any generic data-defining process, the function will not be listed at each update of the plotting program. For reference, though, it is listed in the appendix.

Customizing the Programs for Your System

The demonstration programs on the DGLPRG: disc will send graphics output to the current console of a Series 200 Computer. The "current console" is the CRT where alpha is displayed after the system is booted; i.e., the CRT where the Pascal system command lines appear. Graphics display device selection is performed by the DISPLAY_INIT procedure. If you would like to use a different CRT (or other graphics device) as your display device, you must change the DISPLAY_INIT procedure call accordingly.

The first parameter in the DISPLAY_INIT procedure call is called the **device selector**. It specifies which display device you would like to use for graphics output. The demonstration programs declare the device selector as a constant with the name `CrtAddr`. Graphics display devices are selected as follows:

- A device selector of 3 specifies the current console as the graphics display device (again, this is where the command line appears). This is the value used in all of the demonstration programs. If the current console has no graphics hardware, the system may search for another display that *does* have graphics hardware and make it the graphics display device.
- A device selector of 6 specifies any other internal CRT as the graphics display device (if one exists). Internal refers to any display whose frame buffer resides in the system's "internal space," i.e., any CRT which does not require a select code and/or bus address to access it.
- A device selector in the range 8 through 31 specifies the select code of the interface to which the desired graphics display device is attached.
- A device selector in the range 100 through 3199 specifies the composite HP-IB select code/bus address of the desired HP-IB graphics display device.

The second parameter is the DISPLAY_INIT procedure call is called the **control value**. It is used to specify device-dependent characteristics of the graphics display device. The demonstration programs declare the control value as a constant called `ControlWord`. For complete details on this value, refer to the DISPLAY_INIT section of Appendix B. However, there are two cases that are worth discussing:

- If you have a Model 237 and are using the bit-mapped display as your current console, you may remove the type-ahead buffer echo at the bottom of the screen (and use the entire display for graphics) by specifying a device selector (`CrtAddr`) of 3 and a control value (`ControlWord`) of 256:

  ```
  CrtAddr=          3;
  ControlWord=      256;
  ```

  The value of 0 (used in the demonstration programs) retains the type-ahead buffer.
- If you have an HP 98627A RGB interface connected to a 60 Hz, non-interlaced color monitor[1], you can designate it as the graphics display device by specifying the interface's select code as the device selector (`CrtAddr`), and a control value (`ControlWord`) of 256 (specifying US STD, 512 x 390, 60Hz refresh). See the table in the DISPLAY_INIT section of Appendix B for details.

The control values are not merely "magic numbers." Bits 10, 9 and 8 in the control value allow you to specify what kind of CRT you wish to interface to (in the case of external monitors), or to set characteristics of the display (in the case of the Model 237 bit-mapped display). The value of 256 is not necessary if you are plotting on a U.S. Standard display (see the "External Color Displays" section in this chapter); 0 defaults to the same characteristics as does 256.

The final parameter in the DISPLAY_INIT procedure call is an integer variable that will be assigned 0 if the display device was successfully initialized, or a non-zero value if initialization failed. For more details, refer to the DISPLAY_INIT section of Appendix B.

By modifying the device address and/or the control value, images which were drawn on one device can be drawn on another device with a minimum of effort.

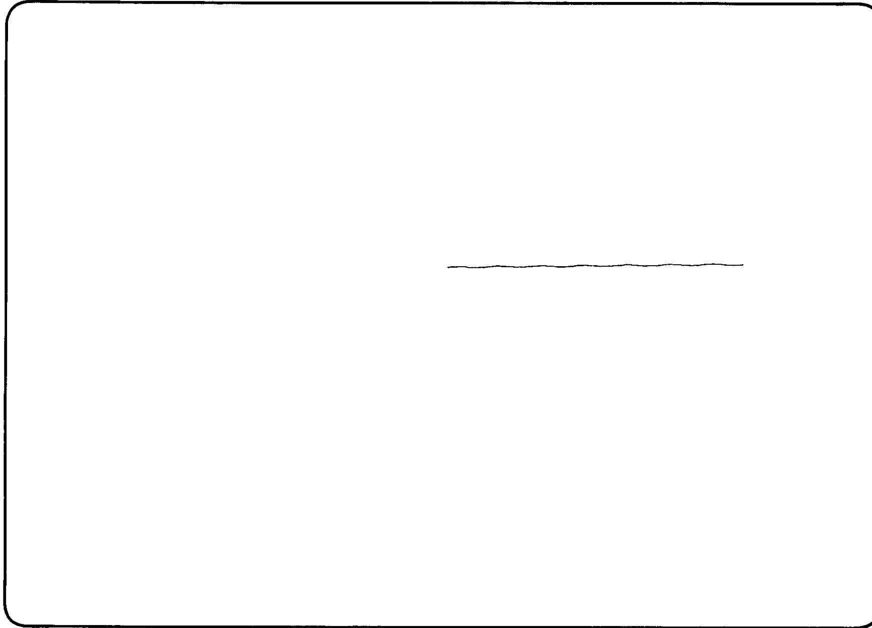
There are some limitations, though. If you are doing an operation on one display device, and attempt to send the image to another device which does not support that operation, it won't work.

Drawing Lines

You are encouraged to compile and run the following programs on your computer as they are presented. Turn on your machine and load the Pascal system (if you don't know how to do this, see Chapter 2 of the *Pascal 3.0 User's Guide*). This program, as most of the following programs, use the compiler directive `INCLUDE`. Compile and run the following program; it is on the file "SinLine" on your `DGLPRG:` disc.

1 Depending on your choice of color monitor, there may be more specification necessary in the control value variable of the DISPLAY_INIT procedure. See the "External Color Displays" section in Chapter 3.

To move the pen somewhere, you call the procedure MOVE, and to draw lines, you call the procedure LINE. Both of these procedures have two parameters: the X and Y coordinates of the point you want to move or draw to. The following program does just that.



```
program SinLine(output);
import dgl_lib;                          {get graphics routines}
const
  CrtAddr=       3;                      {address of internal CRT}
  Control=       0;                      {device control; 0 for CRT}
var
  ErrorReturn:   integer;                {variable for initialization outcome}
  X:             integer;
  Y:             real;
$include 'DGLPRG:DataPoint'$             {function: y:=f(x) }
$page$  {*****************************************************************}
begin                                    {body of program "SinLine"}
graphics_init;                           {initialize graphics system}
display_init(CrtAddr,Control,ErrorReturn);      {which output device?}
if ErrorReturn=0 then begin              {output device initialization OK?}
  for X:=1 to 100 do begin               {100 points total}
    Y:=DataPoint(X);                     {get a point from the function}
    if X=1 then move(X/100,Y)            {move to the first point...}
    else line(X/100,Y);                  {...and draw to all the rest}
  end; {for X:=1 to 100}
end; {ErrorReturn=0?}
graphics_term;                           {terminate the graphics package}
end.                                     {program "SinLine"}
```

Probably the first reaction you had when looking at the previous plot was that the plot doesn't show you anything. But as you can see, this simple program is all you need to draw a rudimentary plot.

You must always execute the procedure GRAPHICS_INIT before any other graphics routine; if you don't, almost every graphics routine called will either be ignored or will cause an error. As its name implies, it initializes the graphics system; that is, it sets various graphics parameters to their default values. These are the operations performed by the GRAPHICS_INIT routine:

- Sets the aspect ratio to 1;
- Sets the virtual coordinates and viewport limits to range from 0.0 to 1.0 in both the X and Y directions;
- Sets the world coordinate limits to range from −1.0 to +1.0 in both the X and Y directions;
- Sets the starting position to 0,0 in world coordinates; and
- Sets all attributes to their default values.

In case there were any unfamiliar concepts referred to above, don't panic. We will soon cover all the above topics, and more.

The following lines comprise the real guts of the `SinLine` program:

```
if X=1 then move(X/100,Y)
else line(X/100,Y);
```

In a loop, the statement moves to the first point returned by the `DataPoint` function, and draws to all the rest. Each successive point is determined by the loop control variable `X` for the X direction and the value returned by the function `DataPoint` for the Y direction.

The call to the routine GRAPHICS_TERM should be the last graphics routine called. It terminates the graphics package.

Scaling

Probably the first reaction you had when looking at the previous plot was "That doesn't show me anything..." That's true; it doesn't show much information. There are two reasons for this. The first is that there is not enough variation in the curve; it's too flat to show us anything. The second is that it is all compressed on the right half of the screen. If we exaggerated the Y direction to the point where we could see the variations, the lines would be close enough to vertical that it would be somewhat difficult to interpret the curve. Therefore we must expand it toward the left.

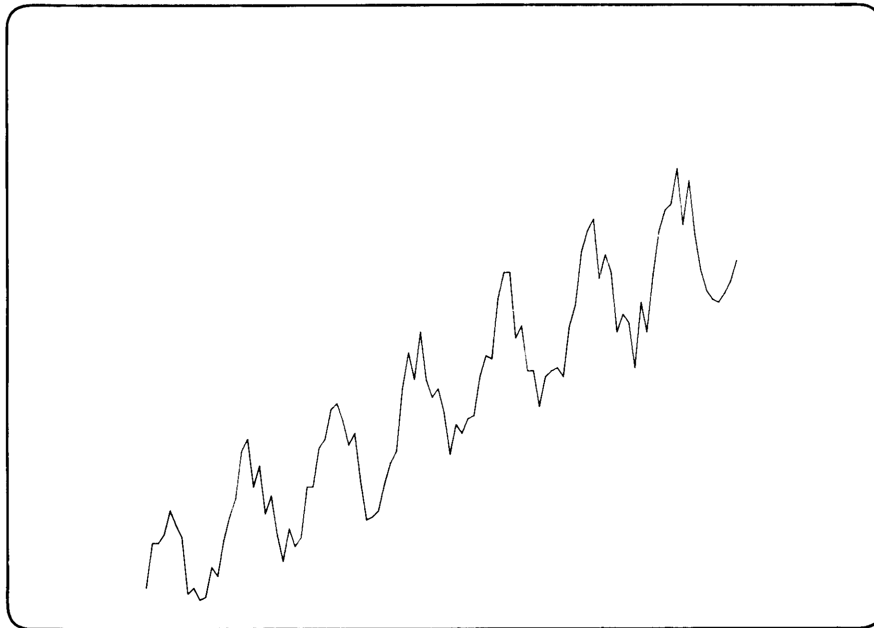
Both of these problems can be remedied by scaling. In this context, scaling could be defined as "defining the values the computer considers to be at the edges of the active plotting surface." The SET_WINDOW procedure defines the transformation used to map coordinates between the virtual display coordinate system (the coordinate system used by the DGL to describe the display device) and the world coordinate system (the coordinate system used by the user). Typically, the left edge is the smaller X, the right edge is the larger X, the bottom is the smaller Y, and the top is the larger Y[1]. Thus any point you plot which falls into these ranges will be visible.

1 This is by convention only. If you specify a value for the left (or bottom) edge which is greater than the value of the right (or top) edge, it is perfectly legal. The only restriction is that the left edge must not *equal* the right edge. The same goes for the bottom and top edges.

In our progressive example, the statement calling SET_WINDOW says that an X value of 0 should be precisely on the left edge of the screen, an X value of 100 should be on the right edge, a Y value of 0.16 is on the bottom, and a Y value of 0.18 is on the top.

The procedure SET_WINDOW typically causes **anisotropic** scaling to be invoked. Anisotropic scaling means that one unit in the X direction is not forced to be the same length as one unit in the Y direction. Conversely, **isotropic** scaling means that one unit in the X direction is equal to one unit in the Y direction, as in a road map. Isotropic scaling is desirable in many cases. In many other cases, however, it is not. In this example, we are graphing the voltage from a sensor versus time, and it makes no sense at all to force seconds to be just as "long" as volts. Since we are dealing with data types which are not equal, it would be better to use unequal, or anisotropic, scaling.

We said that the SET_WINDOW procedure "typically" causes anisotropic scaling to be invoked because there is no procedure that guarantees that the scaling will be isotropic. You can, by doing calculations with aspect ratios, figure what the exact values are to send to SET_WINDOW to force isotropic scaling. This will be covered in the next chapter. Here is the next version of our progressive example. It is in the file "SinWindow" on the DGLPRG disc.



```
program SinWindow(output);
import dgl_lib;                              {get graphics routines}
const
  CrtAddr=          3;                       {address of internal CRT}
  ControlWord=      0;                       {device control; 0 for CRT}
var
  ErrorReturn:   integer;                    {variable for initialization outcome}
  X:             integer;
  Y:             real;
$include 'DGLPRG:DataPoint'$                 {function: y:=f(x) }
```

```
{****************************************************************************}
begin                                          {body of program "SinWindow"}
graphics_init;                                 {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin                    {output device initialization OK?}
  set_window(0,100,0.16,0.18);                 {scale the window for the data}
  for X:=1 to 100 do begin                     {100 points total}
    Y:=DataPoint(X);                           {get a point from the function}
    if X=1 then move(X,Y)                      {move to the first point...}
    else line(X,Y)                             {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                                 {terminate the graphics package}
end.                                           {program "SinWindow"}
```

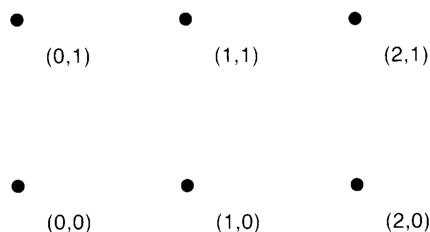
Granted, it would be nice to know what we are plotting, and what the units are, etc., but we'll get there in due time.

Setting the Aspect Ratio

You may have noticed on the last plot that the curve did not extend to the right and left edges of the screen. In fact, the area of screen which was used was exactly as wide as the screen is high. Thus, the **aspect ratio**—the width of the screen divided by the height—is exactly 1. This was the second operation done by the procedure GRAPHICS_INIT, mentioned previously.

For most applications, one would not want to be restricted to using only a square area in the middle of the screen. The procedure used to change the aspect ratio of the plotting surface is SET_ASPECT. When calling the SET_ASPECT procedure, only the ratio of the two parameters is used; thus, the values may be virtually anything, as long as the ratio between them is reasonable.

To set the aspect ratio such that it will use the entire screen of a Model 36 computer, call the SET_ASPECT procedure with parameters 511 and 389. These are the number of pixels in the X direction *minus one*, followed by the number of pixels in the Y direction *minus one*. Distance measures the amount of space *between* pixels[1], not the number of pixels. To illustrate the reason why 1 must be subtracted from both values, imagine a *very* low-resolution graphics display: 3 pixels in the X direction by 2 pixels in the Y direction.



1 The word "pixel" is a blend of the two words "picture element," and it is the smallest addressable point on a plotting surface. A Model 36 computer has 512×390-pixel resolution: thus there can be no more than 512 dots drawn in any one row of the CRT, or 390 dots drawn in any one column.

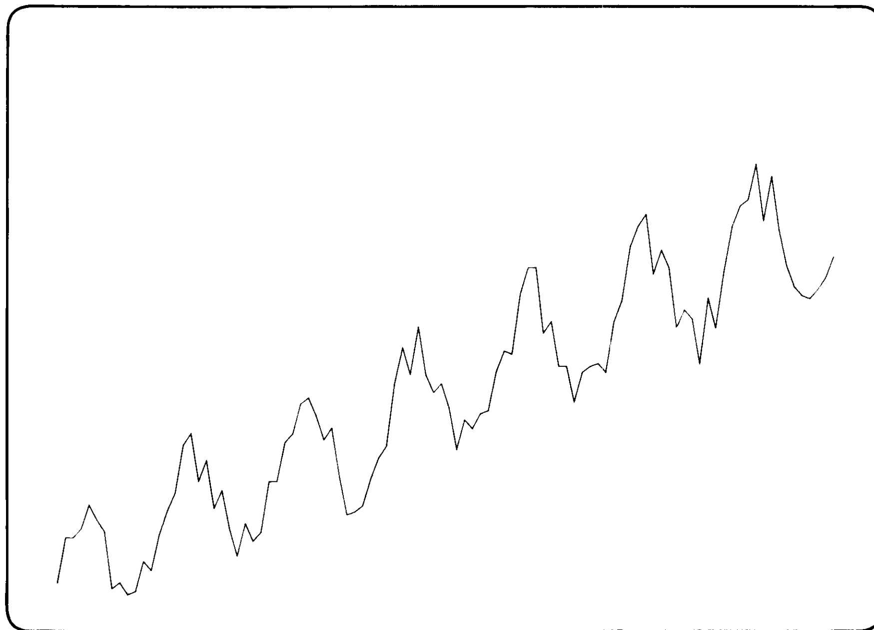
As you can see, the distance between the rightmost pixels and the leftmost pixels is 2, and the distance between the uppermost pixels and the lowest pixels is 1. Thus, the ratio of width to height of this plotting surface is 2:1, rather than 3:2, as it would be if number of pixels were used.

From the previous explanation, it follows that the correct values to pass to the SET_ASPECT procedure would be 511 and 389 for the Models 217 and 236; 399 and 299 for the Models 216, 220 and 226; 1023 and 751 for the Model 237 (with type-ahead buffer); or 1023 and 767 for the Model 237 (with type-ahead buffer removed). These numbers are the numbers of pixels in the X and the Y directions, respectively, for those computers.

In the next version of our progressive example, the only change is that the aspect ratio has been altered so the whole screen has been used. The following statement was placed immediately prior to the SET_WINDOW statement:

```
set_aspect(511,389);
```

This program may be found on file "SinAspect" on the `DGLPRG:` disc.



This plot looks better than the last one; the whole screen is being used. There is still one problem, though. We can see *relative* variations in the data, but what are the units being used? We saw at the very beginning of the chapter that we were measuring voltage, but with the plot at its current state, we don't know if the height of the curve is signifying differences of micro-volts, millivolts, megavolts, dozens of volts, or what? And we probably wouldn't want the text (explaining units, etc.) to be written in the same area that the curve is in, as it could obstruct part of the data curve. Therefore, we need to be able to specify a subset of the screen for plotting the curve and put explanatory information outside this area. The next section tells you how to do this.

Defining a Viewport

A viewport is a subset of the plotting area into which the window limits are linearly mapped. It is specified in **virtual coordinates**.

Virtual Coordinates and World Coordinates

Before we define a viewport, we need to know about the two different types of units which exist. These two types of units are **virtual display coordinates** and **world coordinates**. Since a viewport is a "window" onto which the world coordinates are mapped, and in order for viewports to be predictable, they must be specified in units which are not dependent upon the user's graphical model—the world coordinates. Since world coordinates are associated with the graphical model employed by the user, and virtual coordinates are associated with the display device, it makes much more sense to use virtual coordinates when specifying the limits of a viewport. (Note that world coordinates are set when specifying a window—they both start with "w"—and virtual coordinates are set when specifying a viewport—they both start with "v".) Virtual coordinates always range from 0.0 to 1.0 in one direction, and 0.0 to a number dictated by the aspect ratio in the other direction. A viewport is associated with the display device, rather than the graphical model used in your program.

These are the most important characteristics of virtual coordinates:

- The lower left of the plotting area is *always* 0,0.
- Virtual coordinates are isotropic; that is, one unit in the X direction is the same distance as one unit in the Y direction.
- Virtual coordinates are limited to the range 0 through 1. The maximum coordinate on one side is 1, and the maximum coordinate on the other side is less than or equal to 1.

The following discussion assumes that the aspect ratio is set such that the whole screen is used: 511/389 for the Models 217 and 236; 399/299 for the Models 216, 220 and 226; or 1023/767 for the Model 237. Since the height of the screen is less than the width of the screen, the longer edge is in the X direction, therefore, Xmax in virtual coordinates is 1.0. If the screen had been higher than it is wide, Ymax in virtual coordinates would have been 1.0. That was the easy part. Once you've decided which edge is longer, and thus defined the units along that edge, you need to find out the length of the *shorter* sides in virtual coordinates. Typically, these values will be known because you explicitly specify the aspect ratio yourself. However, if you don't know the aspect ratio (and therefore the virtual coordinates maxima), you can interrogate the system with a call ot the INQ_WS procedure[1]. This will be done in the next chapter. For now, though, we'll just observe that the virtual coordinate limits (for the entire screen, remember) are 0.0 to 1.0 in X, and 0.0 through 299/399 = 0.749373433584 (on the Models 216, 220 and 226), 0.0 through 389/511 = 0.761252446184 (on the Models 217 and 236), or 0.0 through 767/1023 = 0.749755620723 on the Model 237).

1 The INQ_WS procedure is a DGL procedure through which you can find out various parameters of the graphics system.

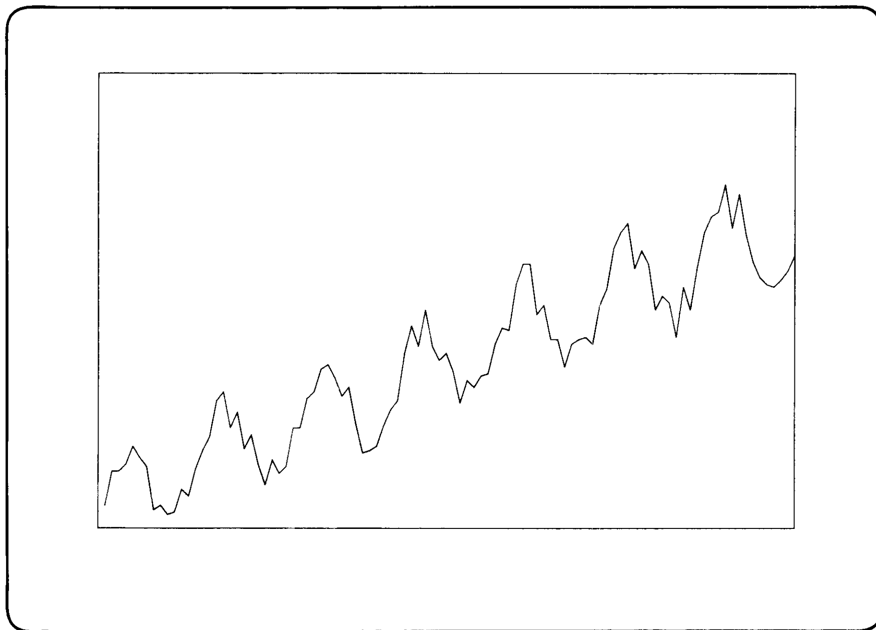
Specifying the Viewport

The SET_VIEWPORT procedure sets up a transformation which will convert points in world coordinates into points on the plotting surface. The call to SET_VIEWPORT in the following program specifies that the lower left-hand corner of the viewport area is at 0.10,0.12 and the upper right-hand corner is at 0.99,0.70.

```
set_viewport(0.10,0.99,0.12,0.70);
```

This is the area which the SET_WINDOW procedure affects. We will also draw a box around the viewport limits by drawing the rectangle bounded by −1 and 1 in both the X and Y directions. (The default window limits are −1 to 1 in both directions.) It is done in this example so you can see the area specified by the SET_VIEWPORT procedure call.

And here is the output from the next version of our progressive example (found on file "SinViewpt" on the DGLPRG: disc). The only change is that a call to SET_VIEWPORT has been placed immediately after the line calling SET_ASPECT.

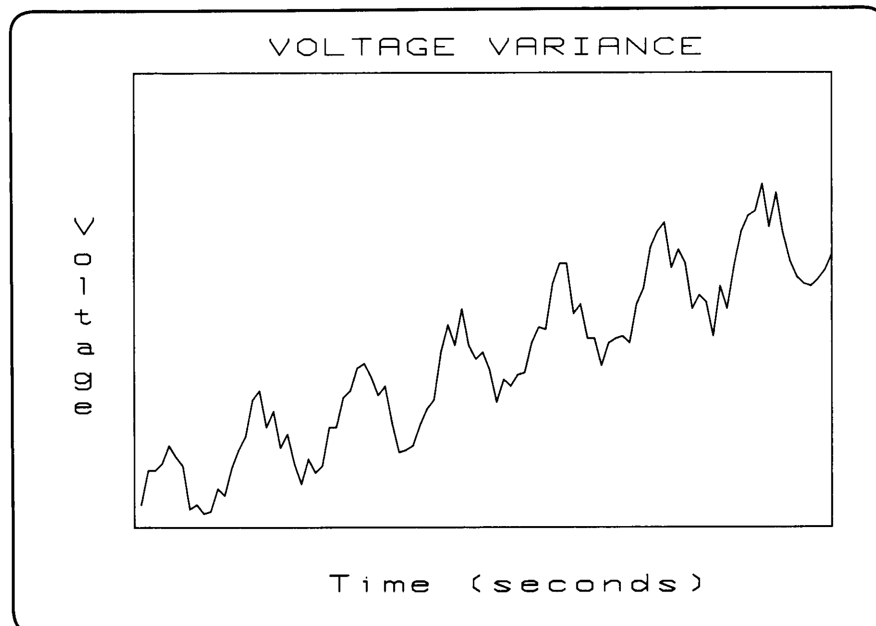


Labelling a Plot

With the inclusion of the call to the SET_VIEWPORT procedure, we have enough room to include labels on the plot. Typically, in a Y-vs-X plot like this, there is a title for the whole plot centered at the top, a Y-axis title on the left edge, and a X-axis title at the bottom.

The DGL procedure GTEXT writes text onto the graphics screen. You can position the label by calling MOVE to get to the point at which you want the label to be placed. It is the lower left corner of the label which ends up at the point to which you moved. In other words, we will move to the position on the screen at which we want the lower left corner of the text to be placed.

Notice in the following plot that the Y-axis label on the left edge of the screen is created by writing one letter at a time. We only need to move to the position of the first character in that label because we terminate each one-character GTEXT call with a carriage return/linefeed. This causes the pen to go one line down, ready for the next (one-character) line of text. (There is another way to plot vertical labels; we'll see it shortly.)




```
program SinLabel1(output);
import dgl_lib, dgl_inq;                    {get graphics routines}
const
  CrtAddr=          3;                      {address of internal CRT}
  ControlWord=      0;                      {device control; 0 for CRT}
var
  ErrorReturn:  integer;                    {variable for initialization outcome}
  Strng:        string[7];                  {seven characters in 'Voltage'}
  Character:    integer;                    {loop counter for labelling}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$                {function: y:=f(x) }
$page$  {********************************************************************}
begin                                       {body of program "SinLabel1"}
graphics_init;                              {initialize graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
```

```
if ErrorReturn=0 then begin              {output device initialization OK?}
  set_aspect(511,389);                   {use the whole screen}
  move(-0.45,0.9);                       {starting point for the title}
  gtext('VOLTAGE VARIANCE');             {label the plot}
  Strng:='Voltage';                      {the y-axis label}
  move(-0.95,0.3);                       {starting point for the y-axis title}
  for Character:=1 to strlen(Strng) do  {follow every character...}
    gtext(str(Strng,Character,1)+chr(13)+chr(10));      {...with a CR/LF}
  move(-0.3,-0.9);                       {starting point for the x-axis label}
  gtext('Time (seconds)');               {x-axis label}
  set_viewport(0.1,0.99,0.12,0.7);       {define subset of screen}
  move(-1,-1); line(1,-1); line(1,1); line(-1,1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);           {scale the window for the data}
  for X:=1 to 100 do begin               {100 points total}
    Y:=DataPoint(X);                     {get a point from the function}
    if X=1 then move(X,Y)                {move to the first point...}
    else line(X,Y);                      {...and draw to all the rest}
  end; {for X:=1 to 100}
end; {ErrorReturn=0?}
graphics_term;                           {terminate the graphics package}
end.                                     {program "SinLabel1"}
```

This gets the point across, but it would be nice if we could cause some labels to be more obvious by making them bigger; for example, on the main title. Also, you may want the Y-axis title to be turned on its side, and not do the carriage return/line feed trick we did last time.

Setting Character Size

The DGL procedure SET_CHAR_SIZE sets two attributes[1] of all subsequent characters, namely the width and height of the character cells. A character cell contains a character and some blank space above, below, to the left of, and to the right of the character. This blank space allows packing character cells together without making the characters illegible. The amount of blank space depends, of course, on which character is contained in the cell. The values sent to SET_CHAR_SIZE are expressed in world coordinates:

```
set_char_size(Width, Height);
```

When a character size is selected, the width and height associated with a character cell are defined for an unrotated character cell. Thus, when a character is rotated, its shape does not change, even though its width (measured along the X axis) and height (measured along the Y axis) are not the same directions as the display device's axes.

The ability to specify character sizes in world coordinates is valuable when doing graphical output in which the labels are to remain with the objects they describe. In these cases, the characters are scaled using the same scaling as the objects drawn.

In the following program (program `SinLabel2` on a file by the same name on the `DGLPRG:` disc), the character width and height are defined to be something on the order of `2*0.04`. The reason that a 2 was used in these expressions is that the current (default) window limits were −1 to 1, for a distance of 2. The 0.04 comes from the fact that we wanted 4% of the window distance in that direction.

1 An *attribute*, in this context, is a piece of information which helps define or describe some object.

Centering Labels

In that last program, the labels looked reasonably centered. This was only because the starting point was arrived at in a hit-and-miss manner. The main characteristic of labels which makes it difficult to center them is this: the reference point of a label is the lower-left corner of the label. That is, the point you moved to just prior to writing the label will end up at the lower-left hand corner of the label. If we want our labels to be centered, we must figure out how long each label is, subtract half that length from the X position of where we want the center of the label to be placed, and then write the label.

We know what the characters' sizes are; we can set it with the SET_CHAR_SIZE procedure. We can also determine how long the string of text to be labelled is. This is found by using the standard procedure STRLEN. If you give it a string, it will return the length (in characters) of that string.

Horizontal centering of a string, then, can be accomplished by subtracting the value returned by the following expression from the desired X position of the center of the label[1]:

```
(strlen(Text)*CharWidth)/2
```

Thus, if we want a label centered horizontally about the point `X`, and at a Y value of `Y`, we could say:

```
move(X-(strlen(Text)*CharWidth)/2,Y);
```

Setting the Label's Direction

Quite often, labels need to be at some other angle than horizontal. We saw a few pages ago that a vertical label could be done—albeit somewhat clumsily—by labelling one horizontal character at a time, and following each by a carriage return/line feed. What we need is a way to specify that we want labels to be plotted at whatever angle we specify.

Through the DGL procedure SET_TEXT_ROT, you can specify the amount of rotation you want the label to undergo. However, you must specify this in two pieces: the X displacement and the Y displacement. For example:

`set_text_rot(2,-1);`	Label goes down and right; a $-26.57°$ angle.
`set_text_rot(1,0);`	Label is horizontal; default direction.
`set_text_rot(87,87);`	Label goes up and right at a 45° angle.
`set_text_rot(0,5);`	Vertical label; ascending.
`set_text_rot(-1,0);`	Upside-down label.

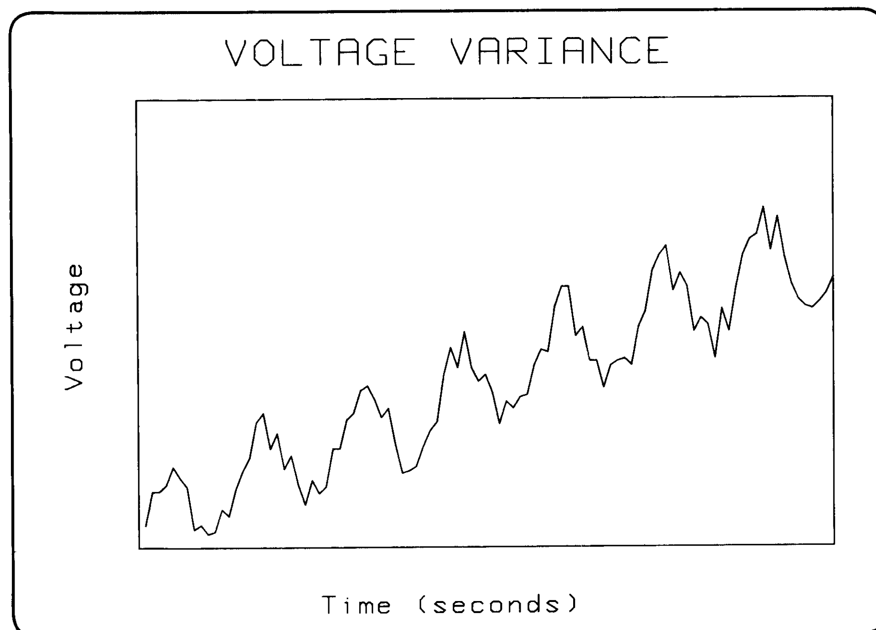
The SET_TEXT_ROT procedure deals only with the ratio of the run and rise parameters. Thus, multiplying both parameters by the same number will not change the angle at which the subsequent labels are written. The third example above, which sets both the run and the rise to 87, could have used any two numbers as parameters, as long as they equaled each other. Going 87 units up for every 87 units to the right yields the same angle as going 19 units up for every 19 units to the right, etc.

1 This is quite close to the truth, but is an approximation. There is an inter-character gap, which is the space caused by the fact that a character is placed inside a *character cell*, and it is complicated because the amount of space on the left side of a character is different from the amount of space on the right. See the *Character Cell* section in the next chapter.

Any particular *angle* you want can be passed to the SET_TEXT_ROT procedure by operating on the angle with the cosine and sine functions. For example, to cause labels to be written at an angle of $\pi/4$ (a 45° angle), you could use the following statement. It assumes there is a constant called `Pi` which has a value approximately equal to 3.1415926535897.

```
set_text_rot(cos(Pi/180*45),sin(Pi/180*45));
```

With these two statements, we can make a marked improvement in the quality of the output. The next version of our progressive example uses them.




```
program SinLabel2(output);
import dgl_lib, dgl_inq;                      {get graphics routines}
const
  CrtAddr=       3;                           {address of internal CRT}
  ControlWord=   0;                           {device control; 0 for CRT}
var
  CharWidth:     real;                        {width of character in world coords}
  CharHeight:    real;                        {height of character in world coords}
  Text:          string[20];                  {temporary holding place for text}
  ErrorReturn:   integer;                     {variable for initialization outcome}
  X:             integer;
  Y:             real;
$include 'DGLPRG:DataPoint'$                  {function: y:=f(x) }
$page$  {*********************************************************************}
begin                                         {body of program "SinLabel2"}
graphics_init;                                {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
```

```
if ErrorReturn=0 then begin                  {output device initialization OK?}
  set_aspect(511,389);                       {use the whole screen}
  CharWidth:=2*0.04;                         {char width: 4% of screen width}
  CharHeight:=2*0.08;                        {char height: 8% of screen height}
  set_char_size(CharWidth,CharHeight);       {install character size}
  Text:='VOLTAGE VARIANCE';                  {define the text to be labelled}
  move(-(strlen(Text)*CharWidth)/2,0.9);{go to start point for centered label}
  gtext(Text);                               {label the text}
  set_text_rot(0,1);                         {vertical labels}
  CharWidth:=2*0.025;                        {char width: 2.5% of screen width}
  CharHeight:=2*0.04;                        {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);       {install character size}
  Text:='Voltage';                           {define the text to be labelled}
  move(-0.9,-(strlen(Text)*CharWidth)/2);        {start point of centered label}
  gtext(Text);                               {label the text}
  set_text_rot(1,0);                         {horizontal labels}
  Text:='Time (seconds)';                    {define the text to be labelled}
  move(-(strlen(Text)*CharWidth)/2,-0.92);       {start point of centered label}
  gtext(Text);                               {label the text}
  set_viewport(0.1,0.99,0.12,0.7);           {define subset of screen}
  move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);               {scale the window for the data}
  for X:=1 to 100 do begin                   {100 points total}
    Y:=DataPoint(X);                         {get a point from the function}
    if X=1 then move(X,Y)                    {move to the first point...}
    else line(X,Y);                          {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                               {terminate the graphics package}
end.                                         {program "SinLabel2"}
```

Bold Labels

Many times it's nice to have the most important titles not only in large letters, but **bold** letters, to make them stand out even more. It is possible to achieve this effect by plotting the label several times, moving the label's starting position just slightly each time. In the following version of the program (on file "SinLabel3" on your DGLPRG: disc), notice the FOR loop used when labelling the main title. The loop variable, X, goes from −3 to 3. This is the offset in the X direction of the label's starting position.

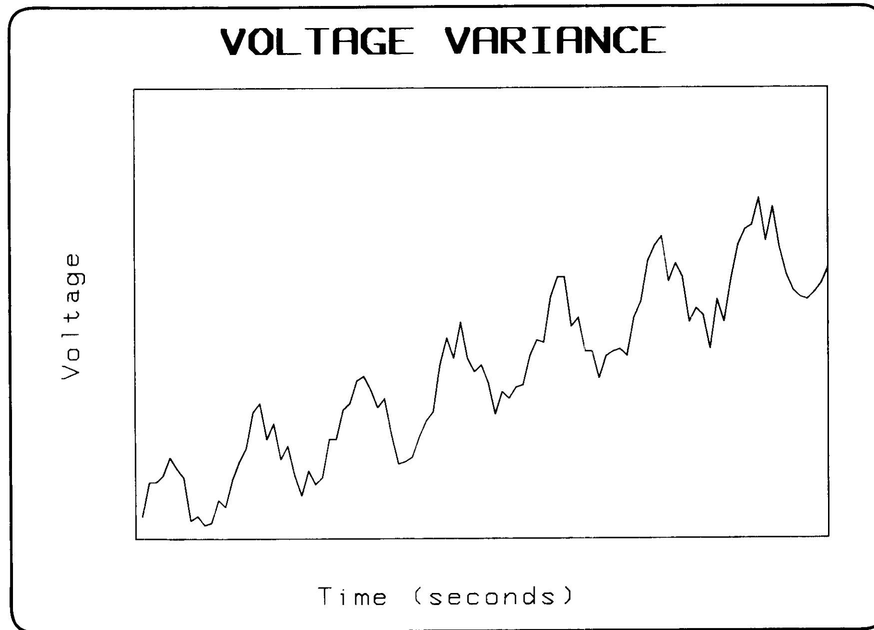
The only change in the program was that the statements labelling the main title:

```
move(-(strlen(Text)*CharWidth)/2,0.9);
gtext(Text);
```

were replaced by the following:

```
for X:=-3 to 3 do begin
  move(-(strlen(Text)*CharWidth)/2+X*0.002,0.9);
  gtext(Text);
end;
```

This method can also be used for offsetting in the Y direction. Or, offset both X and Y. This will give you characters which are thick in a diagonal direction, which makes them look like they are coming out of the page at you. However, a more typical bolding is produced by offsetting only in the X direction.



Now we know what we are measuring—voltage vs. time—but we still do not know the units being used. What we need is an X-axis and a Y-axis, to show us where to put the numbers.

Axes and Tick Marks

When drawing axes, they are typically composed of a straight line defining the axis itself, and short lines, perpendicular to the axes, to indicate the spacing of units. These short lines are called **tick marks**. Usually, the tick marks are grouped into multiples of a nice round number so as to make it easier to understand where the multiples are. These groups are delimited by causing the first tick mark in each group to be larger than the rest.

When writing an axis routine, it is almost always desirable to cause a major tick mark to be coincident with the other axis. For example, if you draw an X axis and select a major tick count of five, it would probably be undesirable to have a minor tick mark (say, two ticks to the right of a major tick) cross the Y axis. This would mean that you would have to go three ticks to the right of the Y axis to find a major tick, but only two ticks if you were going to the left.

Following are some sections of code that do the processing necessary for an axis; an X-axis in this case. A Y-axis proceeds with similar steps. Assume the following variables are defined:

Spacing:	The distance between tick marks on the axis.
Location:	The Y-value of the X-axis.
Xmin,Xmax:	The left and right ends of the X-axis, respectively.
Major:	The number of tick marks to go before drawing a major tick mark. If Major = 5, every fifth tick mark will be major.
Majsize:	The length, in current units, of the major tick marks.
Minsize:	The length, in current units, of the minor tick marks.

The first thing you would do is to draw the axis itself. Its length would be from `Xmin` to `Xmax`, and its Y-position would be `Location`:

```
move(Xmin,Location);
line(Xmax,Location);
```

If the lengths of the major and minor tick marks are `Majsize` and `Minsize`, then half those lengths would be on each side of the axis. Rather than dividing by two at every tick, let's do the divisions once and put the values into their own variables:

```
SemiMinsize:=Minsize*0.5;
SemiMajsize:=Majsize*0.5;
```

We need to round the starting value down to the next major tick mark. The function being used here is a user-defined rounding routine which can round down, up, or to the nearest multiple of the specified value.

```
X:=Round2(Xmin,Spacing*Major,Down);
```

If you do not need or want to force a major tick mark to be at X=0, you could replace the previous statement with the following, which forces a tick, *not* necessarily a major one, to be at zero:

```
X:=Round2(Xmin,Spacing,Down);
```

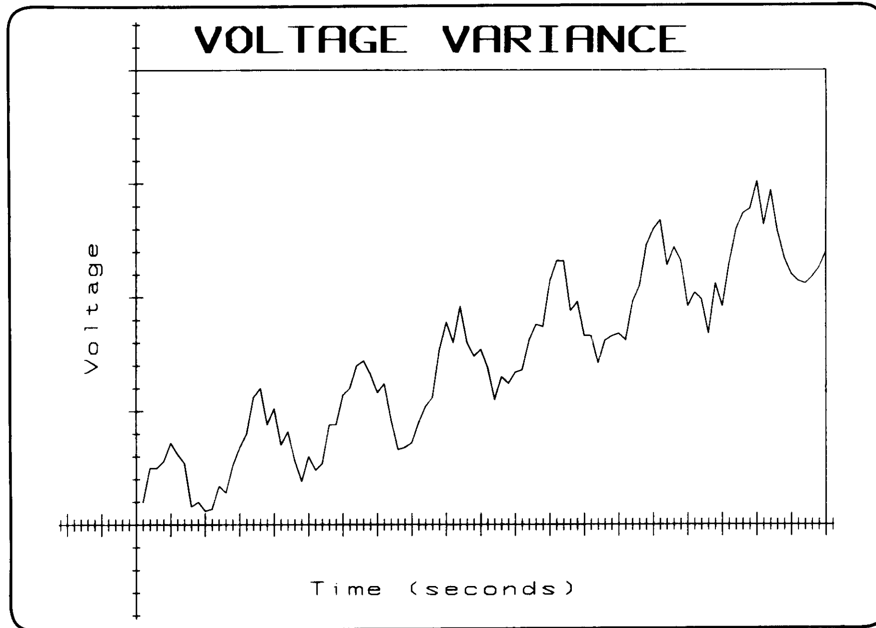
Or, you may not want to round at all; you may want to start making tick marks at the value of `Xmin` no matter what its value—whether it's a nice round number or not. In this case, replace the previous statement with this:

```
X:=Xmin;
```

Now we need to draw all the tick marks. The distance between consecutive ticks is defined by `Spacing`. Every Nth tick will be a major tick, where N is the current value of `Major`. A counter (of type INTEGER or some subrange) will be employed which will be incremented at every iteration and will wrap around. Every time the counter's value is zero, it is time for another major tick mark.

```
Counter:=0;
while X<=Xmax do begin
  if Counter=0 then begin
    move(X,Location-SemiMajsize);
    line(X,Location+SemiMajsize);
  end  {Counter=0?}
  else begin
    move(X,Location-SemiMinsize);
    line(X,Location+SemiMinsize);
  end;  {else begin}
  Counter:=(Counter+1) mod Major;
  X:=X+Spacing;
end;  {while}
```

Here is the next version of our progressive example. It draws both an X and a Y axis. For a complete listing of this program, see the Appendix.




```
program SinAxes1(output);
import dgl_lib, dgl_inq;                    {get graphics routines}
const
  CrtAddr=      3;                          {address of internal CRT}
  ControlWord=  0;                          {device control; 0 for CRT}
type
  RoundType=    (Up, Down, Near);           {used by procedure Round2}
var
  CharWidth:    real;                       {width of char in world coords}
  CharHeight:   real;                       {height of char in world coords}
  Text:         string[20];                 {temporary holding place for text}
  ErrorReturn:  integer;                    {variable for initialization outcome}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$                {function: y:=f(x) }
        •
        •
        •
```

Procedures Xaxis and Yaxis, and function Round2 go here.

```
        •
        •
        •
begin                                       {body of program "SinAxes1"}
graphics_init;                              {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin                 {output device initialization OK?}
  set_aspect(511,389);                      {use the whole screen}
  CharWidth:=2*0.04;                        {char width: 4% of screen width}
  CharHeight:=2*0.08;                       {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);      {install character size}
  Text:='VOLTAGE VARIANCE';                 {define text to be labelled}
  for X:=-3 to 3 do begin                   {make "bold" label}
    move(-(strlen(Text)*CharWidth)/2+X*0.002,0.9);      {center label}
    gtext(Text);                            {label the text}
  end;
```

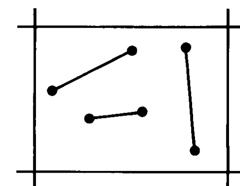
```
    set_text_rot(0,1);                  {vertical labels}
    CharWidth:=2*0.025;                 {char width: 2.5% of screen width}
    CharHeight:=2*0.04;                 {char height: 4% of screen height}
    set_char_size(CharWidth,CharHeight); {install character size}
    Text:='Voltage';                    {define the text to be labelled}
    move(-0.9,-(strlen(Text)*CharWidth)/2);      {start point of centered label}
    gtext(Text);                        {label the text}
    Text:='Time (seconds)';             {define the text to be labelled}
    set_text_rot(1,0);                  {horizontal labels}
    move(-(strlen(Text)*CharWidth)/2,-0.92);     {start point of centered label}
    gtext(Text);                        {label the text}
    set_viewport(0.1,0.99,0.12,0.7);    {define subset of screen}
    move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
    set_window(0,100,0.16,0.18);        {scale the window for the data}
    Xaxis(1,0.16,-50,150,5,0.001,0.0005); {draw the x-axis}
    Yaxis(0.001,0,0.1,0.2,5,2,1);       {draw the y-axis}
    for X:=1 to 100 do begin            {100 points total}
      Y:=DataPoint(X);                  {get a point from the function}
      if X=1 then move(X,Y)             {move to the first point...}
      else line(X,Y);                   {...and draw to all the rest}
    end;  {for X:=1 to 100}
  end;  {ErrorReturn=0?}
  graphics_term;                        {terminate the graphics package}
  end.                                  {program "SinAxes1"}
```

This version is better than the last; it has axes and we can see the units they're delimiting, but obviously, there is a big problem. Not only do the axes and tick marks appear where we want them, they are also many places where we don't want them. We want the axes to stop at the limits of the window, and we also want the tick marks to extend only toward the interior of the graph. What we want is *clipping.*

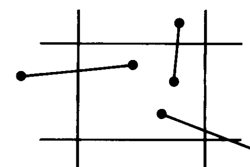
Clipping Lines

Clipping is a method of defining edges of a plotting area, and drawing things which are cut off at those defined edges if they hang over. This is analogous to describing a large drawing on a huge sheet of paper, and but only drawing those parts which are inside some rectangle. What this means is that when clipping is invoked, everything inside the rectangle should look identical to the image (inside the same rectangle) created when clipping is not invoked. Only the things outside the rectangle are affected. Clipping affects lines, text, markers, and polygons.

Clipping a line consists of determining how much of a line is within the clipping limits, and then drawing only the visible part. There are four distinct cases:

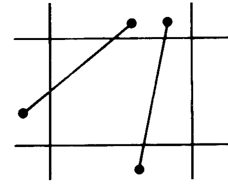


- The line is contained entirely within the clip limits. Therefore, using the original endpoints, draw the entire line.

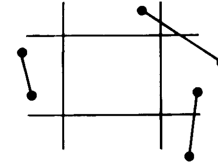


- One endpoint is within the clip limits, but the other one is outside. Therefore, find the intersection between the line to be clipped and all clip limits which intersect it (two at the most). Draw the line from the visible endpoint to the closest edge-intersection.

- Both endpoints are outside the clip limits, but some middle part of the line is visible. Do the same operation as for the single invisible endpoint above, but for both endpoints.



- The entire line is invisible. Reject it; do nothing.



DGL clips images at the display limits—those limits set by the SET_DISPLAY_LIM routine. Often, however, you may wish to clip at other boundaries than the logical display limits. In addition, the parameters for SET_DISPLAY_LIM are expressed in millimeters. Millimeters are quite adequate for setting display limits, but are usually clumsy to work with when the rest of the graph is in world coordinates. But there is a way to do it. There is a DGL routine called CONVERT_WTODMM, which converts world coordinates to millimeters on the display surface. However, SET_DISPLAY_LIM may reset the view surface limits, so some redefinition of other parameters may be necessary. Thus, you can clip using these two routines in conjunction with each other.

A User-Defined Clipping Algorithm

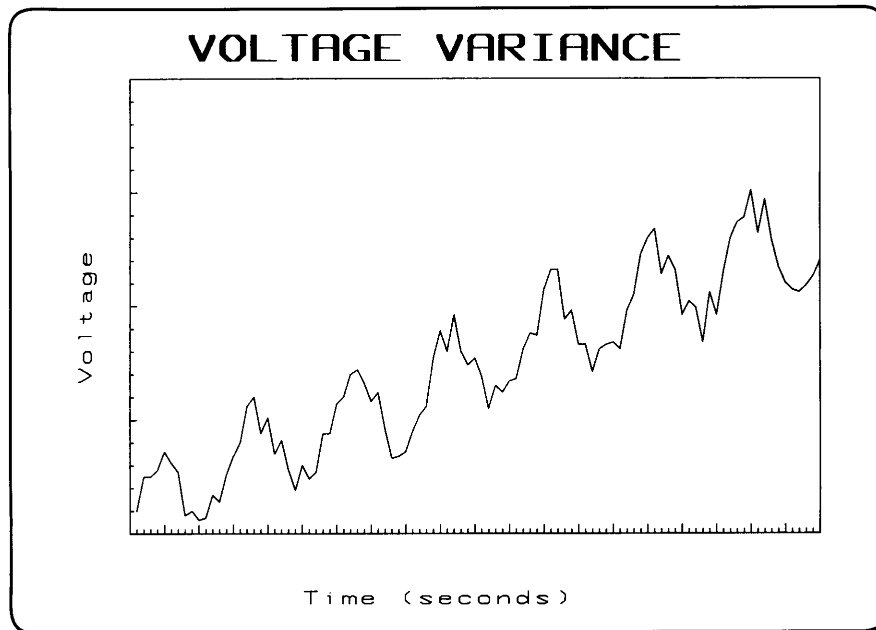
In the appendix is a listing of the program "SinClip", which uses a clipping routine[1] called `ClipDraw`. Also included is a routine to which you pass the desired clip limits: `ClipLimit`. The clip limits may be inside, outside, or coincident with the window edges. After the clipping limits have been defined, a line is passed to the clipping routine. Both endpoints of a line must be known, because intersections between the line being drawn and the edges of the clipping area must be calculated.

These two clipping-related routines allow lines to be clipped outside of any desired rectangular area. However, the axis routines used in the last demonstration program must be modified to call the clipping routine. In addition, there is another modification which would be very convenient to have:

It would be nice if we didn't have to pass the Xmin and Xmax or Ymin and Ymax to their respective routines so they would know where to start drawing tick marks. To do this, we'll just use the global variables `ClipXmin`, `ClipXmax`, `ClipYmin`, `ClipYmax`. Then we'll round the lower window limits *down* to the next value which would have a major tick mark. We round to a major tick mark because (in this case) we want the value of 0 to have a major tick, regardless of whether zero is on the plotting surface.

Installing the modified axis routines results in the following plot. The program may be found on file "SinClip" on the `DGLPRG:` disc.

1 This clipping routine was adapted from a routine on page 66 of the excellent book:
Principles of Interactive Computer Graphics, William M. Newman and Robert F. Sproull, Second Edition, 1979, McGraw-Hill.



This is a good general-purpose clipping routine which is independent of the output device used, *and of the DGL implementation used.* But as we noted earlier, only lines sent to the CLIP-DRAW routine were clipped, and therefore text, written by a call to GTEXT, in addition to markers and polygons, were not clipped.

These axes look much better. Now we know where the numbers should be placed on the axes. Let's learn a little about labelling numbers.

Labelling Axes

In the process of labelling axes, we need to know how to convert numbers to strings which look just like the numbers. The reason for this is that the labelling procedure GTEXT can only accept a string for an input parameter.

There is a standard procedure in Series 200 Pascal called STRWRITE. This allows you to use regular output formats, but, instead of sending the data to a file, the data is put into a string variable. The same format-controlling numbers after colons that can be used for WRITELN can be used for STRWRITE. Let's assume there are three variables defined:

- A string variable `Strng`. This variable will receive the string version of the value converted from REAL;
- An integer `I`. This is merely for a value returned from the STRWRITE routine. It indicates the location of the next unused character in the string;
- And a REAL variable called X which we want to convert to a string.

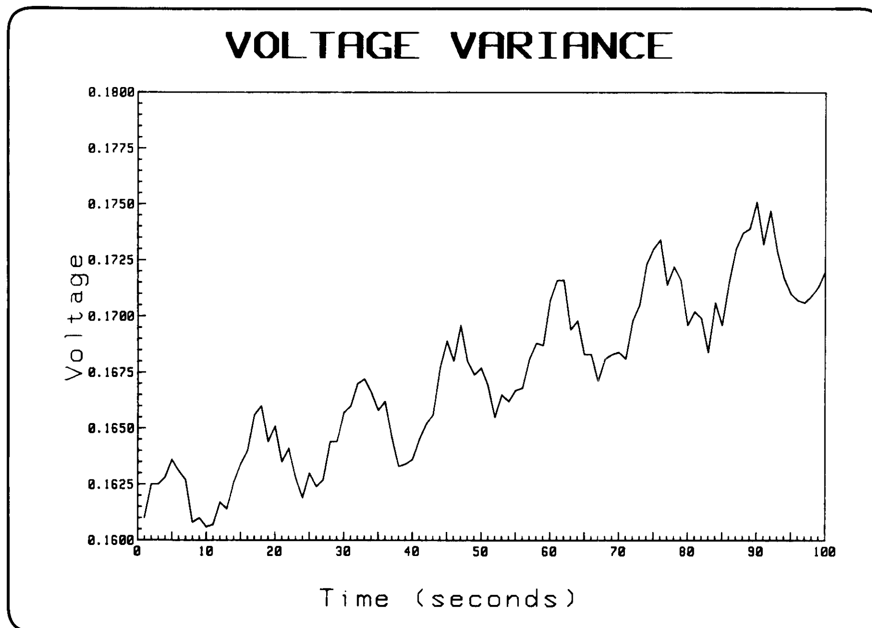
The actual conversion would be accomplished through the following statement:

```
strwrite(Strng,1,I,X:6:4);
```

The `:6` after the `X` tells the computer that the entire field should be six characters wide. This includes the digits to the left of the decimal point, the decimal point itself, and the characters to the right of the decimal point.

The :4 tells the computer that there are to be four digits to the right of the decimal point.

In this program also, we center the labels horizontally by subtracting half the length of the labels from the desired position for the center of the label.




```
program SinAxes2(output);
import dgl_lib;                         {get graphics routines}
const
  CrtAddr=           3;                 {address of internal CRT}
  ControlWord=       0;                 {device control; 0 for CRT}
type
  RoundType=         (Up, Down, Near);          {used by function Round2}
var
  CharWidth:         real;              {width of char is world coords}
  CharHeight:        real;              {height of char is world coords}
  Text:              string[20];        {temporary holding place for text}
  ErrorReturn:       integer;           {variable for initialization outcome}
  I:                 integer;           {return variable from STRWRITE}
  X:                 integer;
  Y:                 real;
  ClipXmin, ClipXmax:  real;            {soft clip limits in x}
  ClipYmin, ClipYmax:  real;            {soft clip limits in y}
$include 'DGLPRG:DataPoint'$            {function: y:=f(x) }
```

```
$page$ {*********************************************************************}
procedure ClipLimit(Xmin, Xmax, Ymin, Ymax: real);
{---------------------------------------------------------------------------}
{  This procedure defines the four global variables which specify where the  }
{  soft clip limits are.                                                     }
{---------------------------------------------------------------------------}
begin                                  {body of procedure "ClipLimit"}
if Xmin<Xmax then begin                { \                             }
  ClipXmin:=Xmin;                      {  \    Force the minimum soft  }
  ClipXmax:=Xmax;                      {   \   clip limit in X to be   }
end                                    {    \  the smaller of the two  }
else begin                             {    /  X values passed into    }
  ClipXmin:=Xmax;                      {   /   the procedure.          }
  ClipXmax:=Xmin;                      {  /                            }
end;                                   { /                             }
if Ymin<Ymax then begin                { \                             }
  ClipYmin:=Ymin;                      {  \    Force the minimum soft  }
  ClipYmax:=Ymax;                      {   \   clip limit in Y to be   }
end                                    {    \  the smaller of the two  }
else begin                             {    /  Y values passed into    }
  ClipYmin:=Ymax;                      {   /   the procedure.          }
  ClipYmax:=Ymin;                      {  /                            }
end;                                   { /                             }
end;                                   {procedure "ClipLimit"}
$page$ {*********************************************************************}
procedure ClipDraw(X1, Y1, X2, Y2: real);
{---------------------------------------------------------------------------}
{    This procedure takes the endpoints of a line, and clips it.  The soft   }
{  clip limits are the real global variables ClipXmin, ClipXmax, ClipYmin,   }
{  and ClipYmax.  These may be defined through the procedure ClipLimit.      }
{---------------------------------------------------------------------------}
label
  1;
type
  Edges=        (Left,Right,Top,Bottom);        {possible edges to cross}
  OutOfBounds=  set of Edges;                   {set of edges crossed}
var
  Out,Out1,Out2:OutOfBounds;
  X, Y:         real;
{---------------------------------------------------------------------------}
procedure Code(X, Y: real; var Out: OutOfBounds);
begin                                  {nested procedure "Code"}
Out:=[];                               {null set}
if x<ClipXmin then Out:=[left]           {off left edge?}
else if x>ClipXmax then Out:=[right];    {off right edge?}
if y<ClipYmin then Out:=Out+[bottom]     {off the bottom?}
else if y>ClipYmax then Out:=Out+[top];  {off the top?}
end;                                   {nested procedure "Code"}
```

```
{------------------------------------------------------------------------------}
begin                                   {body of procedure "ClipDraw"}
Code(X1,Y1,Out1);                       {figure status of point 1}
Code(X2,Y2,Out2);                       {figure status of point 2}
while (Out1<>[]) or (Out2<>[]) do begin {loop while either point out of range}
  if (Out1*Out2)<>[] then goto 1;       {if intersection non-null, no line}
  if Out1<>[] then Out:=Out1
    else Out:=Out2;                     {Out is the non-empty one}
  if left in Out then begin             {it crosses the left edge}
    y:=Y1+(Y2-Y1)*(ClipXmin-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmin;                        {new x is left edge}
  end  {left in Out?}
  else if right in Out then begin       {it crosses right edge}
    y:=Y1+(Y2-Y1)*(ClipXmax-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmax;                        {new x is right edge}
  end  {right in Out?}
  else if bottom in Out then begin      {it crosses the bottom edge}
    x:=X1+(X2-X1)*(ClipYmin-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmin;                        {new y is bottom edge}
  end  {bottom in Out?}
  else if top in Out then begin         {it crosses the top edge}
    x:=X1+(X2-X1)*(ClipYmax-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmax;                        {new y is top edge}
  end;  {top in Out?}
  if Out=Out1 then begin
    X1:=x;   Y1:=y;   Code(x,y,Out1);   {redefine first end point}
  end  {Out=Out1?}
  else begin
    X2:=x;   Y2:=y;   Code(x,y,Out2);   {redefine second end point}
  end;  {else begin}
end;  {while}
move(x1,y1);                            {if we get to this point, the line...}
line(x2,y2);                            {...is completely visible, so draw it}
1:   end;                               {procedure "ClipDraw"}
$page$  {**********************************************************************}
function Round2(N, M: real; Mode: RoundType): real;
{------------------------------------------------------------------------------}
{  This function rounds "N" to the nearest "M", according to "Mode".  This    }
{  function works only when the argument is in the range of MININT..MAXINT.  }
{------------------------------------------------------------------------------}
const
  epsilon=       1E-10;                 {roundoff error fudge factor}
var
  Rounded:       real;                  {temporary holding area}
  Negative:      boolean;               {flag: "It is negative?"}
begin                                   {body of "Round2"}
Negative:=(N<0.0);                      {is the number negative?}
if Negative then begin
  N:=abs(N);                            {work with a positive number}
  if Mode=Up then Mode:=Down            {if number is negative, ...}
  else if Mode=Down then Mode:=Up;      {...reverse up and down}
end;
```

```
case Mode of                              {should we round the number...}
  Down: Rounded:=trunc(N/M)*M;            {...left on the number line?}
  Up:   begin
          Rounded:=N/M;                   {...right on the number line?}
          if abs(Rounded-round(Rounded))>epsilon then
            Rounded:=(trunc(Rounded)+1.0)*M
          else
            Rounded:=trunc(Rounded)*M;
        end;
  Near: Rounded:=trunc(N/M+M*0.5)*M;      {...to the nearest multiple?}
end;  {case}
if Negative then Rounded:=-Rounded;       {reinstate the sign}
Round2:=Rounded;                          {assign to function name}
end;                                      {function "Round2"}
$page$  {******************************************************************}
procedure XaxisClip(Spacing, Location: real; Major: integer;
  Majsize,Minsize: real);
{--------------------------------------------------------------------------}
{  This procedure draws an X-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                     }
{    Spacing:   The distance between tick marks on the axis.                }
{    Location:  The Y-value of the X-axis.                                  }
{    Major:     The number of tick marks to ge before drawing a major tick  }
{               mark.  If Major=5, every fifth tick mark will be major.     }
{    Majsize:   The length, in world units, of the major tick marks.        }
{    Minsize:   The length, in world units, of the minor tick marks.        }
{--------------------------------------------------------------------------}
var
  X:              real;                   {X position of tick marks}
  SemiMajsize:    real;                   {half of major tick size}
  SemiMinsize:    real;                   {half of minor tick size}
  Counter:        integer;                {Keeps track of when to do major ticks}
begin                                     {body of procedure "XaxisClip"}
SemiMajsize:=MajSize*0.5;                 {calculate half of major tick size}
SemiMinsize:=MinSize*0.5;                 {calculate half of minor tick size}
Counter:=0;                               {start with a major tick}
ClipDraw(ClipXmin,Location,ClipXmax,Location);  {draw the X-axis itself}
X:=Round2(ClipXmin,Spacing*Major,Down); {round to next lower major}
while X<=ClipXmax do begin                {loop until greater than ClipXmax}
  if Counter=0 then                       {do a major tick mark?}
    ClipDraw(X,Location-SemiMajsize,X,Location+SemiMajsize)
  else
    ClipDraw(X,Location-SemiMinsize,X,Location+SemiMinsize);    {do minor tick}
  Counter:=(Counter+1) mod Major;         {Keep track of which length tick to do}
  X:=X+Spacing;                           {go to next tick position}
end;  {while}
end;                                      {procedure "XaxisClip"}
```

```
$page$ {*********************************************************************}
procedure YaxisClip(Spacing, Location: real; Major: integer;
  Majsize, Minsize: real);
{---------------------------------------------------------------------------}
{  This procedure draws an Y-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                      }
{    Spacing:   The distance between tick marks on the axis.                 }
{    Location:  The X-value of the Y-axis.                                   }
{    Major:     The number of tick marks to ge before drawing a major tick   }
{               mark.  If Major=5, every fifth tick mark will be major.      }
{    Majsize:   The length, in world units, of the major tick marks.         }
{    Minsize:   The length, in world units, of the minor tick marks.         }
{---------------------------------------------------------------------------}
var
  Y:             real;                    {Y position of tick marks}
  SemiMajsize:  real;                     {half of major tick size}
  SemiMinsize:  real;                     {half of minor tick size}
  Counter:      integer;                  {Keeps track of when to do major ticks}
begin                                     {body of procedure "YaxisClip"}
SemiMajsize:=Majsize*0.5;                 {calculate half of major tick size}
SemiMinsize:=Minsize*0.5;                 {calculate half of minor tick size}
Counter:=0;                               {start with a major tick}
ClipDraw(Location,ClipYmin,Location,ClipYmax);
Y:=Round2(ClipYmin,Spacing*Major,Down); {round to next lower major}
while Y<=ClipYmax do begin                {loop until greater than Ymax}
  if Counter=0 then                       {should we do a major tick?}
    ClipDraw(Location-SemiMajsize,Y,Location+SemiMajsize,Y)
  else
    ClipDraw(Location-SemiMinsize,Y,Location+SemiMinsize,Y);
  Counter:=(Counter+1) mod Major;         {keep track of which size tick to do}
  Y:=Y+Spacing;                           {go to next tick position}
end;  {while}
end;                                      {procedure "YaxisClip"}
```

```
$page$  {*****************************************************************}
begin                                       {body of program "SinAxes2"}
graphics_init;                              {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin                 {output device initialization OK?}
  set_aspect(511,389);                      {use the whole screen}
  CharWidth:=2*0.04;                        {char width: 4% of screen width}
  CharHeight:=2*0.08;                       {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);      {install character size}
  Text:='VOLTAGE VARIANCE';                 {define text to be labelled}
  for X:=-3 to 3 do begin                   {make "bold" label}
    move(-(strlen(Text)*CharWidth)/2+X*0.002,0.9);      {center label}
    gtext(Text);                            {label the text}
  end;
  set_text_rot(0,1);                      . {vertical labels}
  CharWidth:=2*0.025;                       {char width: 2.5% of screen width}
  CharHeight:=2*0.04;                       {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);      {install char size}
  Text:='Voltage';                          {define text to be labelled}
  move(-0.97,-(strlen(Text)*CharWidth)/2);       {start point of centered label}
  gtext(Text);                              {label the text}
  Text:='Time (seconds)';                   {define text to be labelled}
  set_text_rot(1,0);                        {horizontal labels}
  move(-(strlen(Text)*CharWidth)/2,-0.92);       {start point of centered label}
  gtext(Text);                              {label the text}
  set_viewport(0.1,0.99,0.12,0.7);          {define subset of the screen}
  move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);              {scale the window for the data}
  ClipLimit(0,100,0.16,0.18);               {define the soft clip limits}
  XaxisClip(1,0.16,5,0.0008,0.0004);        {draw the clipped X-axis}
  YaxisClip(0.0005,0,5,2,1);                {draw the clipped Y-axis}
  CharWidth:=1.3;                           {char width: 1.3 user X units wide}
  CharHeight:=0.0008;                       {char height: .0008 user Y units high}
  set_char_size(CharWidth,CharHeight);      {install character size}
  Text:='';                                 {erase previous definitions of string}
  for X:=0 to 10 do begin                   {eleven X labels}
    strwrite(Text,1,I,X*10:0);              {convert number to string}
    move(X*10-(strlen(Text)*CharWidth)/2,0.1593);      {center the label}
    gtext(Text);                            {label the text}
  end;  {for x}
  Y:=0.16;                                  {starting Y position for Y labels}
  repeat
    strwrite(Text,1,X,Y:6:4);               {convert number to string}
    move(-8,Y-0.0002);                      {center the text vertically}
    gtext(Text);                            {label the text}
    Y:=Y+0.0025;                            {next Y position}
  until Y>0.18;                             {terminating condition}
  for X:=1 to 100 do begin                  {100 points total}
    Y:=DataPoint(X);                        {get a point from the function}
    if X=1 then move(X,Y)                   {move to the first point...}
    else line(X,Y);                         {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                              {terminate the graphics package}
end.                                        {program "SinAxes2"}
```

Notice that even though the clip limits were still active when the axis labels were written, the text (whose characters are merely a series of short lines) was *not* clipped. This is because the GTEXT procedure does not call the user-defined clipping routine CLIPDRAW, it calls the DGL procedures MOVE and LINE. Thus clipping on labelled text is only done at the hard clip limits—the edges of the plotting surface.

This is the final version of our progressive example. It is the version which created the initial display at the beginning of the chapter.

Miscellaneous Graphics Concepts

Chapter 2

In the last chapter we discussed the more elementary graphics operations. In this chapter, we will discuss how to use some of those concepts more fluently, along with several other graphics operations.

As in the last chapter, the demonstration programs in this chapter are stored for your convenience on the `DGLPRG:` disc which was shipped with this manual. You are encouraged to run these programs while you are reading the manual, as they will make understanding the concepts much easier.

Setting the Display Limits

It is possible to define a subarea of the entire display surface by calling the DGL procedure SET_DISPLAY_LIM. The area thus defined is the area in which a subarea can be specified by the SET_ASPECT procedure.

The parameters passed to SET_DISPLAY_LIM are expressed in millimeters. An example call would be:

```
set_display_lim(40.5,100,30,99,Error);
```

This would set the logical limits of the display device to an area whose:

- left edge is 40.5 millimeters from the physical left edge of the display device;
- right edge is 100 millimeters from the physical left edge of the display device;
- bottom edge is 30 millimeters from the physical bottom edge of the display device;
- top edge is 99 millimeters from the physical bottom edge of the display device.

If the integer variable `Error` comes back with a value of 0, no error occurred. An error occurs if either the minimum X or Y is greater than the maximum X or Y, or if the requested area is even partially outside the physical display limits. In either case, the call is ignored and the variable `Error` is returned non-zero.

More on Defining a Viewport

In the last chapter it was mentioned that the SET_VIEWPORT procedure defined a subset of the screen in which to plot. More precisely, the SET_VIEWPORT procedure *defines a rectangular area into which the SET_WINDOW coordinates will be mapped.* That is, the left edge of the window will be placed upon the left edge of the viewport, the right edge of the window will be placed upon the right edge of the viewport, and the same will happen with the bottom and the top edges.

Assuming that the SET_ASPECT procedure has been invoked to make use of the entire screen, the screen has default edge values in the virtual display coordinates of 0.0 through 1.0 in the X direction, and 0.0 through 299/399 $\approx$ 0.75 (for the Models 216, 220 and 226), 0.0 through 389/511 $\approx$ 0.76 (for the Models 217 and 236), or 0.0 through 767/1023 $\approx$ 0.75 (for the Model 237) in the Y direction. The length of a unit in virtual coordinates is defined as "the length of one of the longer edges of the plotting area." To recap the important characteristics of virtual coordinates:

- The lower left of the plotting area is 0,0.
- Virtual coordinates are isotropic; that is, one unit in the X direction is the same distance as one unit in the Y direction.
- Virtual coordinates are limited to the range 0 through 1. The maximum coordinate on one side is 1, and the maximum coordinate on the other side is less than or equal to 1.

As we mentioned in the last chapter, it is trivial to determine the longer edge of the screen in virtual coordinates, but substantially more involved to calculate the length of the shorter edge in virtual coordinates. Since the height of the screen is shorter than the width of the screen, the longer edge is in the X direction; therefore, the maximum X in virtual coordinates is 1.0. If the screen had been higher than it is wide, the maximum *Y* in virtual coordinates would have been 1.0. Now for the interesting part.

Remember that virtual coordinates are isotropic: X and Y units are the same length. This means that the length in virtual coordinate units of the shorter edges of the plotting surface can be determined from the **aspect ratio** of the plotting surface. The aspect ratio is the ratio of width to height of the plotting surface. Thus, if the plotting area is wider than it is high, the ratio would be greater than one. If the plotting area is higher than it is wide, the ratio would be less than one, and if the plotting area were perfectly square, the ratio would be 1. You can determine the aspect ratios of both the virtual display and the logical limits of the plotting surface by calling the INQ_WS procedure with operation selector 254:

```
const
  AspectRatio=          254;        {mnemonic better than magic number}
type
  RatioTypes=           (VirtualDisplay,LogicalLimits);
  RatioType=            array [RatioTypes] of real;
var
  Pac:          packed array [1..1] of char;  { \   These are the sundries  }
  Iarray:       array [1..1] of integer;      {  \  needed by the call to   }
  Ratios:       RatioType;                    {  /  "inq_ws".               }
  Error:        integer;                      { /                           }
     .
     .
     .
inq_ws(AspectRatio,0,0,2,Pac,Iarray,Ratios,Error); cr/if Error<>0
then cr/if writeln('Error ',Error:0, ' in determining aspect ratio.');
```

The user can now use `Ratio[VirtualDisplay]` and `Ratio[LogicalLimits]` to determine what values are used to set the aspect ratio. (For more information on the INQ_WS procedure, look up this procedure in Appendix B.)

Usually, however, the user knows the aspect ratio because he explicitly set it at the beginning of the program, using the SET_ASPECT procedure.

Using the value for the aspect ratio, we can derive a statement which is almost indispensible when writing a general-purpose statement for calling the SET_VIEWPORT procedure. Assuming the aspect ratio is contained in a variable called `AspectRatio`:

```
if AspectRatio>1.0 then begin
  MaxVirtX:=1.0;
  MaxVirtY:=1/AspectRatio
end
else begin
  MaxVirtX:=AspectRatio;
  MaxVirtY:=1.0
end;
```

These statements define the maximum X and maximum Y in virtual coordinate units. This will work no matter what plotting device you are using. Now that we have `MaxVirtX` and `MaxVirtY` defined, we have complete control of the subset we want on the plotting surface. Suppose we want:

- the left edge of the viewport to be 10% of the hard clip limit[1] width from the left edge,
- the right edge of the viewport to be 1% of the hard clip limit width from the right edge,
- the bottom edge of the viewport to be 15% of the hard clip limit height from the bottom, and
- the top edge of the viewport to be 10% of the hard clip limit height from the top.

We would specify:

```
LeftEdge:=0.1*MaxVirtX;
RightEdge:=0.99*MaxVirtX;
BottomEdge:=0.15*MaxVirtY;
TopEdge:=0.9*MaxVirtY;
SET_VIEWPORT(LeftEdge,RightEdge,BottomEdge,TopEdge);
```

1 Hard clip limits are those limits set by the SET_DISPLAY_LIM procedure.

Calculating Window Limits

In our progressive example in the last chapter, we were using the sometimes unrealistic practice of using constants in the SET_WINDOW procedure call. Often you don't know until the program is running what the values to be passed to SET_WINDOW are. The X values which were used in the SET_WINDOW procedure call (0 and 100) came from the fact that there were 100 data points. The Y values (for this type of plot) must be determined either by you or by the computer itself. If you want the computer to determine the X or Y minimum and maximum, you could do it in the following manner. Assuming that the X values are in a real array called `X`:

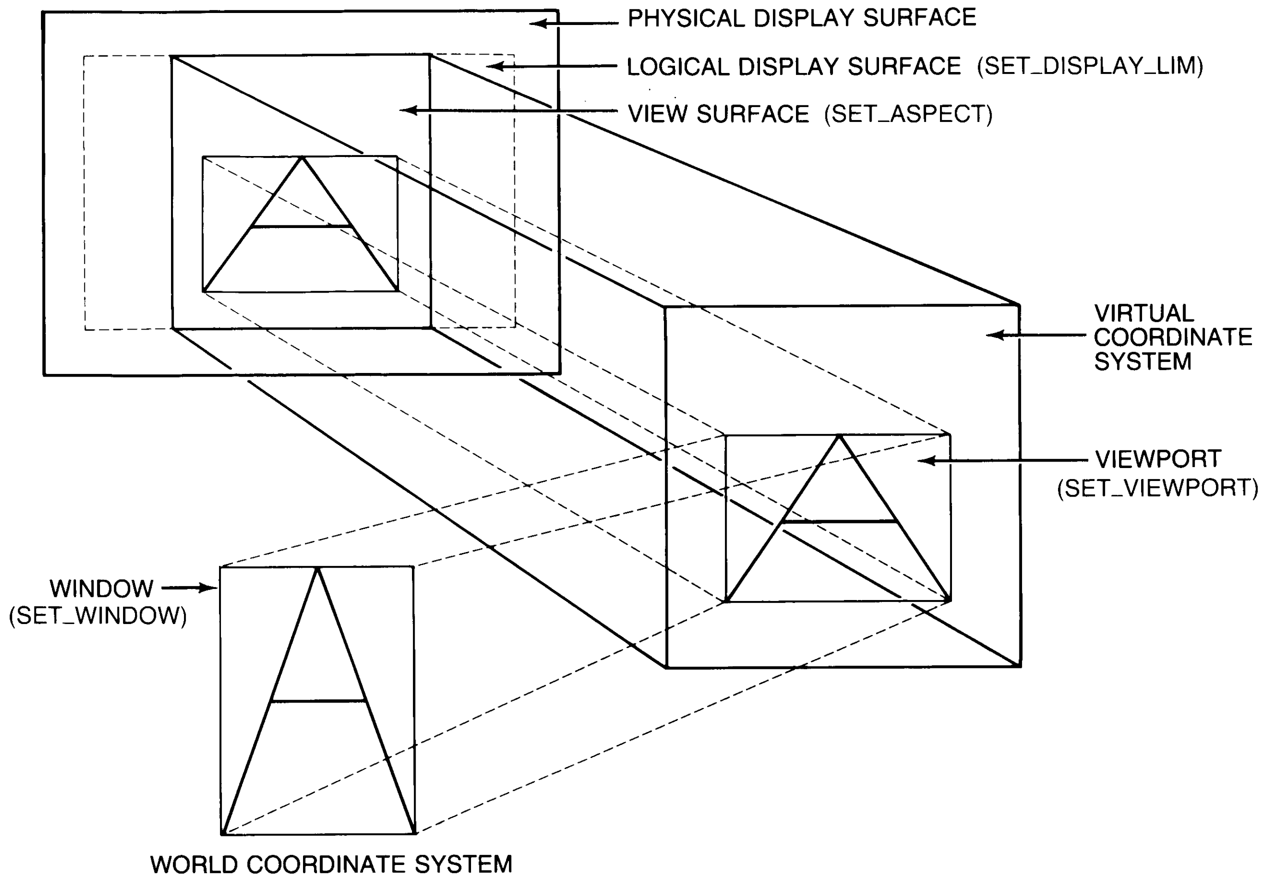
```
const
  MaxReal=        1.79769313486231E308;
     .
     .
     .
Xmax:=-MaxReal;    {Smaller than smallest possible value in array}
for I:=1 to N do   {N is the number of elements in the array}
  if X[I]>Xmax then Xmax:=X[I];
```

A similar method can be used for figuring the minimum value of the X array: First, assign `Xmin` to be `+MaxReal`. The reason this is done is to ensure that at least the first value in the array is used. Then, check through the array of X values, and if the value of any element is smaller than the current minimum, it becomes the new minimum.

Of course, the minimum and maximum Y values can be found in the same manner.

Drawing a Window Frame

The SET_VIEWPORT procedure specifies where in the logical display to put the plot—the subarea of the plotting surface in which to plot. This is the area which the SET_WINDOW procedure affects.



Quite often, a frame is desired around the current window to set it apart from the labels outside the window, and so forth. If the window limits are known (or it is convenient to find out), you can just do a MOVE and four LINEs, as was done in the last chapter. The way it was done in the last chapter was to draw the frame *after* the SET_VIEWPORT call, but *before* the SET_WINDOW call. Since we had not yet set our own window, the default window limits were -1 to 1 in both directions. Therefore, we could say:

```
move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);
```

This is not always the case, however. If you do not know the current window limits, you can interrogate the system through the DGL procedure INQ_WS. The values returned from there can be used to draw the frame. The following lines of code demonstrate how to do this. First, the INQ_WS routine is accessed to determine the current window limits, and then a box is drawn around those limits.

```
const
  WindowLimits= 450;                    {mnemonic better than magic number}
type
  LimitOrder=   (Xmin, Xmax, Ymin,Ymax);
  LimitType=    array [LimitOrder] of real;
var
  Pac:          packed array [1..1] of char;    { \  These are the sundries  }
  Iarray:       array [1..1] of integer;        {  \ needed by the call to   }
  Window:       LimitType;                      {  / the DGL procedure       }
  Error:        integer;                        { /  "inq_ws".               }
        •
        •
        •
inq_ws(WindowLimits,0,0,4,Pac,Iarray,Window,Error);
if Error=0 then begin
  move(Window[Xmin],Window[Ymin]);              {move to lower left corner}
  line(Window[Xmin],Window[Ymax]);              {draw to upper left corner}
  line(Window[Xmax],Window[Ymax]);              {draw to upper right corner}
  line(Window[Xmax],Window[Ymin]);              {draw to lower right corner}
  line(Window[Xmin],Window[Ymin]);              {draw to lower left corner}
end  {Error=0?}
else writeln('Error ',Error:0,' occurred in "Frame"');
```

Turning Displays On and Off

If you ran the last chapter's programs, and do not have a bit-mapped display, you probably noticed that the graphics screen was turned on automatically to show you what was being plotted, but the alpha screen was *not* turned *off* at the same time. If you do have a bit-mapped display (e.g., Model 237), both alpha and graphics occupy the same screen; the screen is either on or it isn't.

In the case of nonbit-mapped displays, as soon as the program ended, the Main Command Level prompt appeared at the top of the screen, obstructing the view of the top portion of the graphics image. This can be mildly annoying as it is, having to turn off the alpha raster by pressing the (GRAPHICS) key, but it rapidly gets more annoying if your program generates printed output *and* plotted output which are not intended to be viewed simultaneously.

What is needed is a way to turn either the alpha raster or the graphics raster on or off at will. There is a way to do this, by calling the OUTPUT_ESC procedure with operation selectors 1050 or 1051. Or, if you prefer a more readable method, the you could write a procedure to do the operations. Assume that there has been an enumerated type declared:

```
type
  DisplayStates=    (Off,On);
```

Here is an example section of code to show you how to turn the displays on or off. The parameter used is assumed to be of the type declared above.

```
{****************************************************************************}
procedure Alpha(State: boolean);
{----------------------------------------------------------------------------}
{  This procedure turns the alpha raster on or off (true=on, false=off).     }
{----------------------------------------------------------------------------}
const
  AlphaRaster=  1051;                    {mnemonic better than magic number}
var
  AlphaOn:      array [1..1] of integer;        { \   This is all stuff that  }
  Rarray:       array [1..1] of real;           {  >  is needed by the        }
  Error:        integer;                        { /   "output_esc" procedure. }
begin                                    {procedure "Alpha"}
if State=On then AlphaOn[1]:=1           {"On" is a boolean constant: true}
else AlphaOn[1]:=0;
output_esc(AlphaRaster,1,0,AlphaOn,Rarray,Error);
if Error<>0 then writeln('Error ',Error:0,' in procedure "Alpha".');
end;                                     {procedure "Alpha"}
```

Similar code could be generated for turning the graphics display on and off. The references to "Alpha" should be changed to "Graphics" just to avoid confusion, and the operation selector should be changed to 1050.

Conversion Between Coordinate Systems

Many times, you'll probably want the ability to convert back and forth between virtual display coordinates and world coordinates. One of the most-used areas where this is desired is where you want to specify some parameter in units relative to the display device, *not* the graphical model currently in use. For example, it is often desirable to specify character sizes as, say, 6% of the screen height. Or, you want to draw an X axis whose tick marks are 1% of the screen height. These, and other places, the values could be specified in world coordinates, but it is an inconvenience to have to specify a constant-sized line or character in units which are varying all over the place. For example, if you have a general-purpose plotting routine which gets it data from an external source, it doesn't know until it gets the data what the window limits are to be. It is only *after* the window limits are known that the character sizes would be specified.

If we could specify these things in virtual display coordinates, we could have the computer do the dirty work of converting from virtual coordinates to whatever the current world coordinates are.

To convert from one coordinate system to another, there are three steps involved:

1. Determine, as a fraction, how far into the old system the point of interest is. For example, if the old system goes from 10 to 20 in X (calculations for Y proceed with identical steps), and you want to find out how far 13 is into that range, you take:

   ```
   OldFraction:=(X-OldXmin)/(OldXmax-OldXmin);
   ```

   or, using our numbers,

   ```
   OldFraction:=(13-10)/(20-10);
   ```

   This evaluates to 0.3, and, sure enough, 13 is three tenths of the way between 10 and 20.
2. Take the fraction found in the previous step, and go the same distance into the new coordinate system. For example, say our new coordinate system goes from 300 to 400. To go into this new range the same fraction of the way, you take:

   ```
   NewDistance:=OldFraction*(NewXmax-NewXmin);
   ```

   Again, putting our numbers into the expression,

   ```
   NewDistance:=0.3*(400-300);
   ```

   This evalutes to 30, and, sure enough, we have to go thirty units into the new coordinate system.
3. To "go into" the new coordinate system means that we have to add the new coordinate system's minimum value to the distance into the new system so that the distance into the new system is relative to the same starting point as the system itself.

   ```
   NewPoint:=NewDistance+NewXmin;
   ```

   or, in our units,

   ```
   NewPoint:=30+300;
   ```

   And 330 is the desired point in the new coordinate system.

The "old" coordinate system and the "new" coordinate system can have any maxima and minima (you are not restricted to converting between the world coordinate system and the virtual coordinate system), and the point of interest may be inside the range, one of the end points, or outside the range; it make no difference to the mathematics.

Following are two routines which convert between virtual display coordinates and world coordinates.

```
{****************************************************************************}
procedure ConvertVirtualToWorld(VirtualX, VirtualY: real;
                            var WorldX,   WorldY:   real);
{----------------------------------------------------------------------------}
{    This routine converts any point in virtual coordinates, whether on the  }
{  plotting surface or not, into world coordinates.                          }
{----------------------------------------------------------------------------}
const
  WindowLimits=         450;       {mnemonic better than magic number}
  ViewportLimits=       451;       {...here, too.                    }
type
  LimitOrder=   (Xmin, Xmax, Ymin, Ymax);
  LimitType=    array [LimitOrder] of real;
var
  Pac:          packed array [1..1] of char;    { \   These are the sundries  }
  Iarray:       array [1..1] of integer;        {  \  needed by the call to   }
  Window:       LimitType;                      {   > the DGL procedure       }
  Viewport:     LimitType;                      {  /  "inq_ws".               }
  Error:        integer;                        { /                           }
begin                              {body of procedure "ConvertVirtualToWorld"}
inq_ws(WindowLimits,0,0,4,Pac,Iarray,Window,Error);
if Error<>0 then writeln('Error ',Error:0,
  ' in determining window limits in "ConvertVirtualToWorld".');
inq_ws(ViewportLimits,0,0,4,Pac,Iarray,Viewport,Error);
if Error<>0 then writeln('Error ',Error:0,
  ' in determining viewport limits in "ConvertVirtualToWorld".');
WorldX:=(VirtualX-Viewport[Xmin])   { \  Calculate X distance from left...  }
  /(Viewport[Xmax]-Viewport[Xmin])  {  \ ...convert to a fraction...        }
  *(Window[Xmax]-Window[Xmin])      {  / ...so same fraction into world...  }
  +Window[Xmin];                    { /  ...add Xmin to get value.          }
WorldY:=(VirtualY-Viewport[Ymin])   { \  Calculate Y distance from bottom...}
  /(Viewport[Ymax]-Viewport[Ymin])  {  \ ...convert to a fraction...        }
  *(Window[Ymax]-Window[Ymin])      {  / ...so same fraction into world...  }
  +Window[Ymin];                    { /  ...add Ymin to get value.          }
end;                                {procedure "ConvertVirtualToWorld"}
```

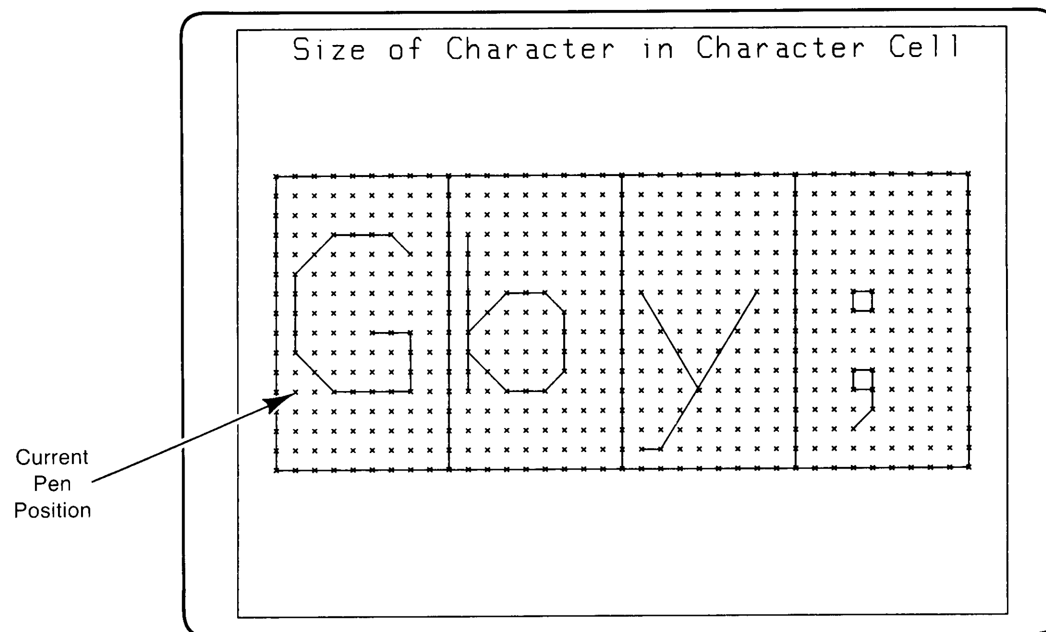
```
{*****************************************************************************}
procedure ConvertWorldToVirtual(WorldX,   WorldY:   real;
                            var VirtualX, VirtualY: real);
{-----------------------------------------------------------------------------}
{    This routine converts any point in world coordinates, whether on the    }
{  plotting surface or not, into virtual coordinates.                         }
{-----------------------------------------------------------------------------}
const
  WindowLimits=         450;        {mnemonic better than magic number}
  ViewportLimits=       451;        {...here, too.                    }
type
  LimitOrder=   (Xmin, Xmax, Ymin, Ymax);
  LimitType=    array [LimitOrder] of real;
var
  Pac:          packed array [1..1] of char;    { \   These are the sundries  }
  Iarray:       array [1..1] of integer;        {  \  needed by the call to   }
  Window:       LimitType;                      {   > the DGL procedure       }
  Viewport:     LimitType;                      {  /  "inq_ws".               }
  Error:        integer;                        { /                           }
begin                           {body of procedure "ConvertWorldToVirtual"}
inq_ws(WindowLimits,0,0,4,Pac,Iarray,Window,Error);
if Error<>0 then writeln('Error ',Error:0,
  ' in determining window limits in "ConvertWorldToVirtual".');
inq_ws(ViewportLimits,0,0,4,Pac,Iarray,Viewport,Error);
if Error<>0 then writeln('Error ',Error:0,
  ' in determining viewport limits in "ConvertWorldToVirtual".');
VirtualX:=(WorldX-Window[Xmin])       { \  Calculate X distance from left...  }
  /(Window[Xmax]-Window[Xmin])        {  \ ...convert to a fraction...        }
  *(Viewport[Xmax]-Viewport[Xmin])    {  / ...so same fraction into world...  }
  +Viewport[Xmin];                    { /  ...add Xmin to get value.          }
VirtualY:=(WorldY-Window[Ymin])       { \  Calculate Y distance from bottom...}
  /(Window[Ymax]-Window[Ymin])        {  \ ...convert to a fraction...        }
  *(Viewport[Ymax]-Viewport[Ymin])    {  / ...so same fraction into world...  }
  +Viewport[Ymin];                    { /  ...add Ymin to get value.          }
end;                                  {procedure "ConvertWorldToVirtual"}
```

More on Labelling a Plot

To help you get a better grasp of the concept of labelling, there will be four small sections, each of which demonstrates something more about the concept of labelling a graph.

The Character Cell

The first program deals with the relationship between the size of the character, per se, and the size of the **character cell**—that rectangle in which the character is placed. This program is on file "CharCell" on the `DGLPRG:` disc.




```
program CharCell(output);                {program name same as file name}
import dgl_lib, dgl_inq;                 {access the necessary procedures}
const
  Crt=              3;                   {device address of graphics raster}
  Control=          0;                   {device control word; ignored for CRT}
type
  LorgType=         1..9;                {the valid values to pass the "Lorg"}
  Str255=           string[255];         {for the procedure "Glabel"}
var
  Error:            integer;             {display_init return variable; 0 = ok}
  I, X, Y:          integer;             {loop control variables}
```

```
$page$  {*********************************************************************}
begin                                      {body of program "CharCell"}
graphics_init;                             {initialize graphics library}
display_init(Crt,Control,Error);           {initialize CRT}
if Error=0 then begin                      {if no error occurred...}
  set_aspect(511,389);                     {use the whole screen}
  move(-1,-1);  line(-1,1);  line(1,1);  line(1,-1);  line(-1,-1);
  set_window(-2,38,-7.5,22.5);             {define appropriate window}
  set_char_size(1,2);                              { \                     }
  move(1,21);                                      {  >  Do main label.    }
  gtext('Size of Character in Character Cell'); { /                     }
  for X:=0 to 36 do begin                  { \                     }
    for Y:=0 to 15 do begin                {  \                    }
      move(X-0.1,y+0.1);                   {   \   Draw the four 9x15  }
      line(X+0.1,Y-0.1);                   {    \  character cells. Make }
      move(X+0.1,Y+0.1);                   {    /  a frame around each, }
      line(X-0.1,Y-0.1);                   {   /   and an X at every   }
    end;  {for y}                          {  /    point.              }
  end;  {for x}                            { /                     }
  for I:=0 to 3 do begin                   {draw a frame around each char cell}
    move(I*9,0); line(I*9,15); line(I*9+9,15); line(I*9+9,0); line(I*9,0);
  end;
  set_char_size(9,15);                     {big characters}
  move(1,4);                               {go to starting position}
  gtext('Gby;');                           {label some characters}
end;  {Error=0?}                           {end of conditional code}
graphics_term;                             {terminate graphics library}
end.  {program "CharCell"}                 {end of program}
```

As the diagram shows, a character is drawn inside a rectangle, with some space on all four sides. Both the rectangle's width and height are specified by the values passed to the DGL procedure SET_CHAR_SIZE, and are measured in world coordinates. This rectangle is subdivided into a grid of 9 wide by 15 high. Characters are drawn in this framework.

The current pen position—that position moved to before writing a label—is one unit to the right and four units up from the lower left-hand corner of the character cell. For example, when labelling a lower-case "b", the bottom of the longer vertical line would end up at the point moved to before labelling. Also note that there doesn't have to be any part of the character at the current pen position, as in the upper-case "G" in the plot. For characters which have descenders (lines which go below the "baseline" of the character cell), the current position is still relative to the lower left corner of the character *cell*, not the character.

Of course, the little ×s in the plot above are not drawn when you label a string of text; they are there solely to show the position of the characters within the character cell.

The DGL procedure SET_CHAR_SIZE specifies the height of the character *cell*, not the character itself.

Setting Character Size

In a previous section, we discussed translation of points between coordinate systems. And as it was mentioned before, often it is desirable to be able to specify character sizes in screen-dependent units, rather than model-dependent units.

As we saw in the last chapter, there is a DGL procedure called SET_CHAR_SIZE which sets an attribute of all subsequent characters, namely the width and height of the character cells. When using SET_CHAR_SIZE, the characters are scaled using the same scaling as the objects drawn.

In other cases, however, the text size should be related to the display device, rather than the user's graphics model. For example, when a general-purpose display routine gets data from a file, or some other source, it probably does not know until the data is actually received what the range of the data is. Thus, the window limits are calculated in the program. To get the title of the plot of a consistent size, you would have to convert the actual size of the label relative to the display device to the same size expressed in world coordinates so they can be sent to SET_CHAR_SIZE.

The following piece of code shows you how to define character cell height in virtual coordinates, and the width is defined as a fraction of the height; thus, it is an aspect ratio. The reason that the aspect ratio is desired, rather than the character cell width, is that if you want characters with a constant shape, you would just have to take your first parameter, and multiply it by a constant. Thus, in effect, you have just specified the aspect ratio.

The values passed into the routine are converted into character cell width and character cell height in world coordinates, which the DGL procedure SET_CHAR_SIZE needs. SET_CHAR_SIZE is called and the converted values are passed to it. The converted values are retrievable by invoking the INQ_WS procedure with operation selector 250. The character cell height and width are needed by another piece of code (which actually does the labelling) covered shortly.

Here is how to specify character size in virtual coordinates, with an aspect ratio, and convert it into parameters appropriate for the SET_CHAR_SIZE routine. Notice that the conversion routine covered a few sections back is used:

```
var
  Width:        real;                {temporary spot for width}
  X0, Y0:       real;                {0,0 (virtual) in world}
  X1, Y1:       real;                {1,1 (virtual) in world}
        •
        •
        •
ConvertVirtualToWorld(0,0,X0,Y0);    {convert 0,0 in virtual to world}
ConvertVirtualToWorld(1,1,X1,Y1);    {convert 1,1 in virtual to world}
Height:=Height*(Y1-Y0);              {convert height in virtual to world}
Width:=Height*AspectRatio*(X1-X0)/(Y1-Y0);  {convert width in virtual to world}
set_char_size(Width,Height);         {invoke the parameters}
```

Here is the graphical output of a program which demonstrates the use of the procedure CHARSIZE, and then the program itself:

```
program CsizeProg(output);
import dgl_lib, dgl_inq;                      {get graphics routines}
const
  Crt=                  3;                    {address of internal CRT}
  Control=              0;                    {device control; 0 for CRT}
var
  Error:                integer;              {variable for initialization outcome}
  I, J:                 integer;              {utility variables}
  Strng:                string[10];           {temporary holding place for strings}
$include 'DGLPRG:ConvVtoW'$                   {virtual-to-world conversion}
$page$  {*****************************************************************}
procedure CharSize(Height, AspectRatio: real);
{-----------------------------------------------------------------------------}
{    This procedure defines character cell size and the puts the Width and   }
{  Height values into global variables for later use.  The arguments passed  }
{  in are the height of the character cell in VIRTUAL coordinates, and the   }
{  aspect ratio of the character cell.  The values for the window limits     }
{  may be anything; they are taken into account and do not affect the size   }
{  of the characters, since they are defined in virtual coordinates.  This   }
{  procedure, along with Lorg and Ldir, define global variables for use by   }
{  Glabel.                                                                    }
{-----------------------------------------------------------------------------}
```

```
var
  Width:        real;                   {temporary spot for width}
  X0, Y0:       real;                   {0,0 (virtual) in world}
  X1, Y1:       real;                   {1,1 (virtual) in world}
begin                                   {body of procedure "CharSize"}
ConvertVirtualToWorld(0,0,X0,Y0);       {convert 0,0 in virtual to world}
ConvertVirtualToWorld(1,1,X1,Y1);       {convert 1,1 in virtual to world}
Height:=Height*(Y1-Y0);                 {convert height in virtual to world}
Width:=Height*AspectRatio*(X1-X0)/(Y1-Y0);  {convert width in virtual to world}
set_char_size(Width,Height);            {invoke the parameters}
end;                                    {procedure "CharSize"}
$page$  {***************************************************************}
begin                                   {body of program "CsizeProg"}
graphics_init;                          {initialize the graphics system}
display_init(Crt,Control,Error);        {which output device?}
if Error=0 then begin                   {output device initialization OK?}
  set_aspect(511,389);                  {use the whole screen}
  set_window(1,2,100,0);                {scale the window for the data}
  for I:=1 to 6 do begin                {six different character sizes}
    CharSize(I*I*0.01,0.6);             {install character size}
    move(1,I*I*I*0.4+I);                {move to a appropriate place}
    strwrite(Strng,1,J,I*I:0);          {convert number to string}
    gtext(Strng+'%');                   {label the string}
  end;  {for i}
end;  {Error=0?}
graphics_term;                          {terminate the graphics package}
end.                                    {program "CsizeProg"}
```

The FOR loop writes lines of text on the screen with different character sizes. Incidentally, notice also the SET_WINDOW procedure. It specifies a Ymin *larger* than the Ymax. This causes the top of the screen to have a lesser Y-value than the bottom. This is perfectly legal.

Again, character cell height, when using the algorithm above, is measured in virtual coordinates, and the definition of aspect ratio for a character is identical to the definition of aspect ratio for the hard clip limits mentioned earlier: the width divided by the height. Thus, if you want short, fat letters, use an aspect ratio of 1.5 or larger. If you want tall, skinny letters, use an aspect ratio less than about 0.5. If you call the above routine:

`CharSize(0.03,0.6);`	Cell 3% virtual coordinate units high, aspect ratio 0.6.
`CharSize(0.06,0.3);`	Cell 6% virtual coordinate units high, aspect ratio 0.3 (tall and skinny).
`CharSize(0.1,2);`	Cell 10% virtual coordinate units high, aspect ratio 2 (short and fat).

Setting the Label's Direction

We saw in the last chapter that label could be rotated by using the DGL procedure SET_TEXT_ROT, which specifies angles in a run/rise format. Many people, however, deal with angles more easily than run/rise ratios. Again, the angular value is converted to run/rise numbers by taking the cosine and sine of the angle, respectively:

```
set_text_rot(cos(Angle),sin(Angle));
```

You could define a procedure for which the angle could be specified in degrees, radians, or grades[1], depending on the value of the units parameter, which, being an enumerated type, can have the value DEG (degrees), RAD (radians), or GRAD (grades):

```
AngleType=        (Deg, Rad, Grad);
```

The value passed in, in the unit of measure defined by the units parameter, must be converted to radians. Radians are the only units understood by the trigonometric functions in Pascal. Conversion is accomplished by a simple division. (The division could be changed to a multiply by the reciprocal. This would increase the speed with little loss of understandability.)

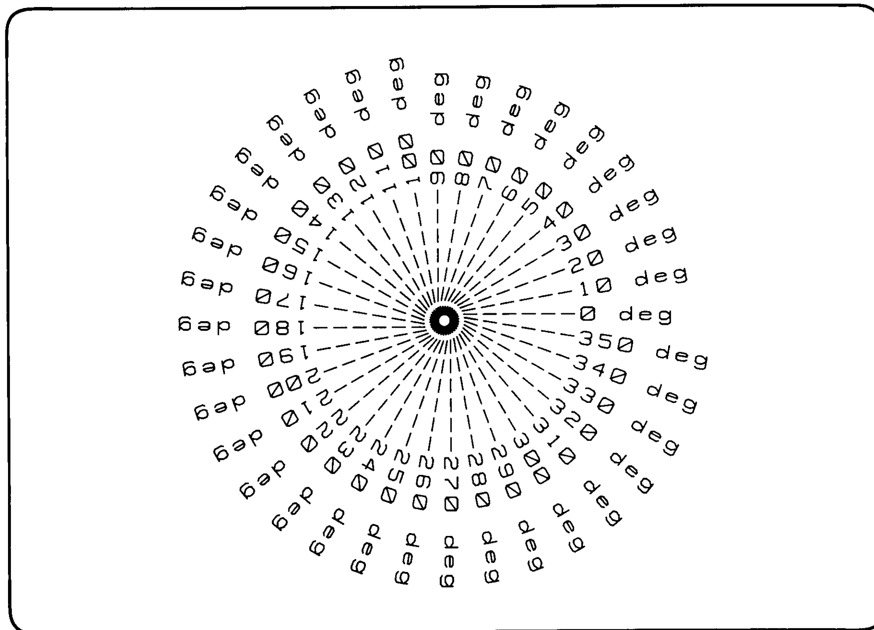
```
const
  Deg_per_rad=  57.2957795131;  {180/pi: for converting degrees to radians}
  Grad_per_rad= 63.6619772368;  {200/pi: for converting grads to radians}
        •
        •
        •
case Units of
  Deg:  Direction:=Direction/Deg_per_rad;      {degrees to radians}
  Rad:  ;                                      {correct units already}
  Grad: Direction:=Direction/Grad_per_rad;     {grads to radians}
end;  {case}
set_text_rot(cos(CharTheta),sin(CharTheta));   {invoke the new text direction}
```

For example, assuming you call the routine `LabelDirection`, and that there is a constant called "Pi" which has a value of 3.1415926535897:

`LabelDirection(0,Deg);`	Writes label horizontally to the right.
`LabelDirection(Pi/2,RAD);`	Writes label vertically, ascending.
`LabelDirection(14,Grad);`	Writes label ascending a gentle slope, up and right.
`LabelDirection(Pi,Rad);`	Writes label upside down.
`LabelDirection(270,DEG);`	Writes label vertically, descending.

[1] One revolution = 360° = 2π radians = 400 grades.

Here is a plot demonstrating the specification of a label's direction by a genuine angle:




```
program LdirProg;                          {program name same as file name}
import dgl_lib;                            {access the necessary procedures}
const
  Crt=                3;                   {device address of graphics raster}
  Control=            0;                   {device control word; ignored for CRT}
type
  AngType=            (Deg,Rad,Grad);      {used by procedure LabelDirection}
var
  Error:              integer;             {display_init return variable; 0 = ok}
  I,J:                integer;             {loop control variable and spare}
  Strng:              string[50];          {string to label}
  CharTheta:          real;                {global variable for label direction}
$page$ {*********************************************************************}
procedure LabelDirection(Direction: real; Units: AngType);
{--------------------------------------------------------------------------}
{   This procedure is used in conjunction with LabelOrigin, CharSize and    }
{  Glabel.  It sets the labelling direction to be used, and places the      }
{  direction into a global variable so Glabel can use it.                   }
{--------------------------------------------------------------------------}
const
  Deg_per_rad=  57.2957795131;  {180/pi: for converting degrees to radians}
  Grad_per_rad= 63.6619772368;  {200/pi: for converting grads to radians}
begin                                      {procedure "LabelDirection"}
case Units of
  Deg:  Direction:=Direction/Deg_per_rad;    {degrees to radians}
  Rad:  ;                                    {correct units already}
  Grad: Direction:=Direction/Grad_per_rad;  {grads to radians}
end;  {case}
CharTheta:=Direction;                      {put into a global variable}
set_text_rot(cos(CharTheta),sin(CharTheta));    {invoke the new text direction}
end;                                       {procedure "LabelDirection"}
```

```
$page$  {*******************************************************************}
begin                                       {body of program "LdirProg"}
graphics_init;                              {initialize graphics library}
display_init(Crt,Control,Error);            {initialize CRT}
if Error=0 then begin                       {if no error occurred...}
  set_aspect(511,389);                      {use the whole screen}
  set_window(-1,1,-1,1);                    {define appropriate window}
  set_char_size(0.05,0.08);                 {set the size for the characters}
  for I:=0 to 35 do begin                   {every ten degrees}
    Strng:='';                              {empty the string}
    strwrite(strng,1,J,I*10:0);             {convert the loop variable to degrees}
    Strng:='-------'+Strng+' deg';          {attach prefix and suffix}
    LabelDirection(I*10,Deg);               {specify label direction}
    move(0,0);                              {move to the center of the screen}
    gtext(Strng);                           {label the text}
  end;  {for I}
end;  {Error=0?}
graphics_term;                              {terminate graphics library}
end.                                        {program "LdirProg"}
```

When a character size is selected whether through the DGL routine SET_CHAR_SIZE or through the utility routine CHARSIZE, the width and height associated with a character cell are defined for an unrotated character cell. Thus, when a character is rotated, its shape does not change, even though its width (measured along the X axis) and height (measured along the Y axis) are not the same directions as the display device's axes.

In the preceding plot, you may have noticed that the hyphens do not precisely meet in the middle. This brings up another point: when you move to a point and then write a label, which part of the label ends up at that point? In other words, how is the label justified?

Justifying Labels

On a label written by the GTEXT procedure, the label is always justified at the lower left-hand corner of the label. Unfortunately, this does not lend itself to centering text, which is often a very desirable thing. It would be nice if we could programmatically select how the label should be justified. For the progressive example we were working on in the last chapter, the main title needed to be as far toward the top of the graph as it can be, and at the same time, centered horizontally. The following addresses just this kind of need.

For horizontal centering, there are three possible choices: left-justified, centered, and right-justified. For vertical centering, there are also three choices: bottom-justified, centered, and top-justified. Thus, there are nine possible combinations of values which can be sent to the LABELJUSTIFY routine: left, centered, and right for the X direction, and for each of these, bottom, centered, and top for the Y direction.

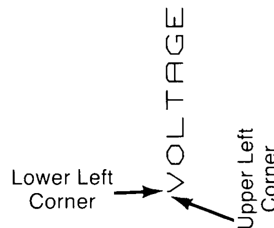
Assume there are two enumerated types declared:

```
HJustifyType=(Left,HCentered,Right);
VJustifyType=(Bottom,VCentered,Top);
```

Label justification is relative to the *label*, not the plotting surface, and it is independent of the current label direction. For example, if you have specified:

- upper left label justification,
- and label direction of 90°,
- a move to point (6,8),

and then write the label, it is written going straight up, not horizontally:



Therefore, it is the upper left corner of the label which is at point 6,8 *relative to the rotated label.* However, it is the lower left corner of the label which is at 6,8 *relative to the plotting device* because the label has been rotated.

Note that two things are obtained by calls to the INQ_WS procedure: the current pen position, and the current character size (in world coordinates).

If you are going to use the label justification scheme just described, you will need to write your own labelling routine which takes into account the current justification values. Label justification gets a little tricky when dealing with user-definable label direction, as you can see in the section of code below.

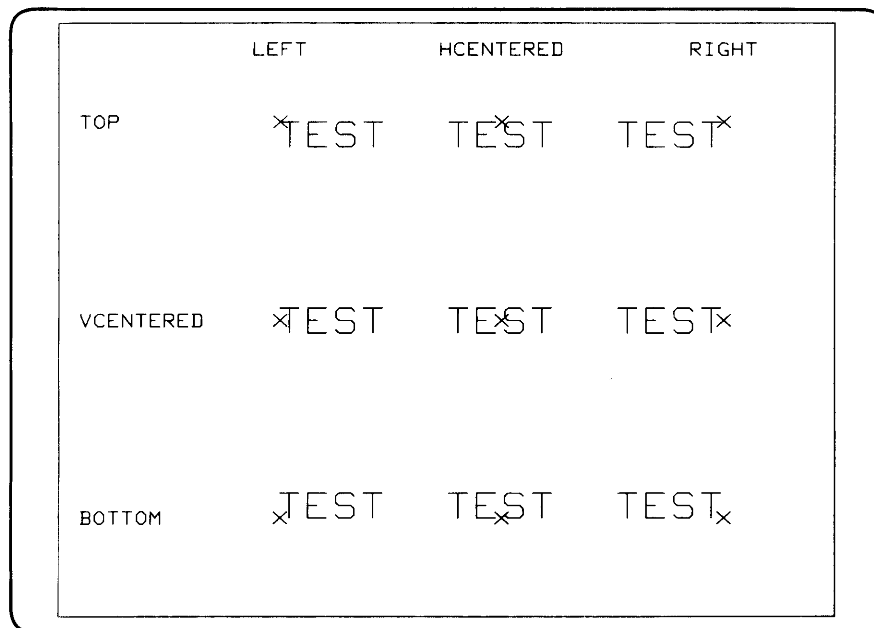
The following three global variables are assumed to exist:

- `HJustification`: The currently-defined horizontal justification. This is of the previously-mentioned type `HJustifyType`.
- `VJustification`: The currently-defined vertical justification. This is of the previously-mentioned type `HJustifyType`.
- `CharTheta`: This real variable is the current label direction, expressed in radians. We need to keep this in a global variable because there is no operation selector we can send to INQ_WS to determine it.

```
const
  CharSizeCode=         250;          {mnemonic better than magic number}
  CurrentPosition=      259;          {ditto}
type
  Positions=            (X,Y);
  PositionType=         array [Positions] of real;
  CharAttributes=       (Width,Heighth);
  CharAttrType=         array [CharAttributes] of real;
```

```
var
  Chars:              integer;
  Charsize:           CharAttrType;
  Len,Height:         real;           {length and height of character string}
  Dx,Dy:              real;
  R,Theta:            real;           {for rectangular-to-polar conversion}
  Pac:                packed array [1..1] of char;    {  \  These are the   }
  Iarray:             array [1..1] of integer;        {   \ sundry items    }
  Position:           PositionType;                   {   / needed for the  }
  Error:              integer;                        {  /  call to "inq_ws"}
        •
        •
        •
inq_ws(CharSizeCode,0,0,2,Pac,Iarray,Charsize,Error);     {get pen position}
if Error<>0 then writeln('Error',Error:0,' in "Glabel".');
Chars:=strlen(text);
Len:=Charsize[Width]*(7*Chars+2*(Chars-1))/9;   {length minus inter-char gap}
Height:=Charsize[Heighth]*8/15;                 {height minus inter-line gap}
Dx:=Len*(-ord(HJustification)/2);
Dy:=Height*(-ord(VJustification)/2);
R:=sqrt(Dx*Dx+Dy*Dy);                 { \  Convert to polar coordinates so  }
Theta:=Atan(Dy,Dx);                   { /  rotation is easy.                }
Theta:=Theta+CharTheta;               {add the LabelDirection angle}
Dx:=R*cos(Theta);                     { \  Convert R and the new Theta back }
Dy:=R*sin(Theta);                     { /  to rectangular coodinates.       }
inq_ws(CurrentPosition,0,0,2,Pac,Iarray,Position,Error);     {get pen position}
if Error=0 then begin
  move(Position[X]+Dx,Position[Y]+Dy);  {move to the new starting point}
  gtext(text);
end  {Error=0?}
else writeln('Error',Error:0,' in "Glabel".');
```

And here is a program using all the label-related algorithms mentioned above.



```
program JustProg(output);
import dgl_lib,dgl_inq;                  {get graphics routines}
const
  CrtAddr=            3;                 {address of internal CRT}
  ControlWord=        0;                 {device control; 0 for CRT}
type
  HJustifyType=       (Left,HCentered,Right); {horizontal justification}
  VJustifyType=       (Bottom,VCentered,Top); {vertical justification}
  AngType=            (Deg,Rad,Grad); {used by procedure "LabelDirection"}
  Str255=             string[255];    {for the procedure "Glabel"}
var
  ErrorReturn:        integer;        {variable for initialization outcome}
  HJust:              HJustifyType;   {horizontal justification variable}
  VJust:              VJustifyType;   {vertical justification variable}
  I:                  integer;        {for the strwrite statement}
  Strng:              str255;         {labelled text holder}
  CharWidth,CharHeight: real;         { \  These are global variables    }
  HJustification:     HJustifyType;   {  \ needed by the LabelJustify/   }
  VJustification:     VJustifyType;   {  / LabelDirection/CharSize       }
  CharTheta:          real;           { /  series of procedures.         }
$include 'DGLPRG:ConvVtoW'$              {needed by procedure "CharSize"}
        •
        •
        •
```

Procedures `Frame`, `CharSize`, `LabelDirection`, `LabelJustify`, `Atan`, and `Glabel` go here.

```
        •
        •
        •
begin                                    {body of program "JustProg"}
graphics_init;                           {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin              {output device initialization OK?}
  set_aspect(511,389);                   {use the whole screen}
  set_window(-1,2.5,-0.5,2.5);           {scale the window for the data}
  Frame;                                 {draw a frame around the screen}
  CharSize(0.03,0.6);                    {width=3% screen width; asp. ratio=.6}
  LabelDirection(0,Deg);                 {horizontal labels}
  {===== Labels at the top ===============================================}
  LabelJustify(HCentered,Top);           {label's reference point: top middle}
  for HJust:=Left to Right do begin      {horizontal loop}
    Strng:='';                           {null the string so nothing left over}
    strwrite(Strng,1,I,HJust);           {convert enumerated type to string}
    move(ord(HJust),2.4);                {move to the appropriate place}
    Glabel(Strng);                       {label the string}
  end;  {for HJust}
  {===== Labels on the left edge =========================================}
  LabelJustify(Left,VCentered);          {label's reference point: left middle}
  for VJust:=Top downto Bottom do begin  {vertical loop}
    Strng:='';                           {null the string so nothing left over}
    strwrite(Strng,1,I,VJust);           {convert enumerated type to string}
    move(-0.9,ord(VJust));               {move to the appropriate place}
    Glabel(Strng);                       {label the string}
  end;  {for VJust}
```

```
  {===== Labels ("TEST") with different justifications =====================}
  CharSize(0.06,0.6);                     {characters a bit bigger}
  for HJust:=Left to Right do begin       {horizontal loop}
    for VJust:=Top downto Bottom do begin        {vertical loop}
      LabelJustify(HJust,VJust);                 {set label justification}
      move(ord(HJust)+0.03,ord(VJust)+0.03);     { \                    }
      line(ord(HJust)-0.03,ord(VJust)-0.03);     {  \  Make the "x" at   }
      move(ord(HJust)-0.03,ord(VJust)+0.03);     {  /  the appropriate   }
      line(ord(HJust)+0.03,ord(VJust)-0.03);     { /   place.            }
      move(ord(HJust),ord(VJust));        {move to label's starting position}
      Glabel('TEST');                     {label the text}
    end;  {for VJust}
  end;  {for HJust}
 end;  {ErrorReturn=0?}
 graphics_term;                           {terminate the graphics package}
 end.                                     {program "JustProg"}
```

The ×s indicate where the pen was moved to before labelling the word "TEST". What this diagram means is that, for example, if `LabelJustify(Left,Bottom)` is in effect, and you move to 4,5 to write a label, the lower left of that label would be at 4,5. This automatically compensates for the character size, label direction, and label length. It makes no difference whether there is an odd or even number of characters in the label. If `LabelJustify(Center,Top)` had been in effect, and you had moved to 4,5, the center of the top edge of the label would be at 4,5. You can readily see how useful this concept is in centering labels, both horizontally and vertically.

Monochromatic CRT Drawing Modes

On a monochromatic CRT, there are three different drawing modes available[1]:

- Drawing dominant lines. This is the most obvious drawing mode; pixels are turned on. It is the mode the graphics package is in by default. White lines are drawn on a dark background, and dark lines are drawn on a white background.
- Erasing lines. In this mode, pixels are turned off. If a line is erased on a background which is already dark, there is no effect. This is the method for making sure a line is gone after it may or may not have been drawn.
- Complementing lines. When this type of line is drawn, pixels which are on are turned off, and pixels which are off are turned on. This is for drawing something which will be visible no matter what the background is; e.g., a graphics cursor.

The drawing modes are selected by calling the OUTPUT_ESC procedure. This DGL procedure allows you to control device-dependencies of output devices. The operation selector which controls drawing modes is 1052. Following is an algorithm which takes care of all the necessary variables, declarations, and all-around "housekeeping" involved in selecting a drawing mode. This implementation of the algorithm assumes the existence of the following type declaration:

```
DrawingModeType=  (Dominant,Erase,Complement);
```

Here is the section of code for selecting drawing modes on a monochromatic CRT:

```
const
  SetDrawingMode=        1052;           {mnemonic better than magic number}
var
  DrawMode:      array [1..1] of integer;        { \   This is all stuff that   }
  Rarray:        array [1..1] of real;           {  >  is needed by the         }
  Error:         integer;                        { /   "output_esc" procedure.  }
       •
       •
       •
case Mode of                          { \                                    }
  Erase:         DrawMode[1]:=2;      {  \   Convert DrawingMode enumerated  }
  Dominant:      DrawMode[1]:=0;      {   >  type into the appropriate       }
  Complement:    DrawMode[1]:=3;      {  /   value for OUTPUT_ESC procedure. }
end;  {case}                          { /                                    }
output_esc(SetDrawingMode,1,0,DrawMode,Rarray,Error);          {set it}
if Error<>0 then writeln('Error ',Error:0,' in procedure "DrawingMode".');
```

A characteristic of drawing with drawing mode `Dominant` or drawing mode `Erase` is that if a line crosses a previously-drawn line, the intersection will be the same "color" as the lines themselves. When drawing with drawing mode `Complement`, and a line crosses a previously-drawn line, the intersection becomes the opposite state of the lines. In other words, the pixels being defined by the line being drawn are exclusively-ORed with the pixels already on the screen. For example, assume a black background (like right after calling CLEAR_DISPLAY[2]).

1 There are actually four drawing modes that you can select; however, two of them, dominant and non-dominant, are identical on monochromatic displays. See the section called Writing Modes and Color in the Color Graphics chapter for a description of using non-dominant mode on color displays.

2 There is a way to clear the screen to white, also. Set entry number 0 in the color table (use the SET_COLOR_TABLE procedure) to anything which has a luminosity greater than 0.5.

You invoke a drawing mode `Complement`, then draw a pair of intersecting lines. When the first line is drawn, all pixels are off, so the line being drawn causes all pixels to be turned on along its length. However, when the second line is drawn, it will turn on pixels until it intersects the first line. At that point, the pixel is on, so it gets turned *off*. After that, the rest of the pixels are off, so they are again turned on.

This concept is illustrated by the program DrawMdPrg (found on file "DrawMdPrg" on the `DGLPRG:` disc). The listing is given in the appendix so you can see how it works, but since it is a dynamic display, and constantly changing, it makes little sense to show a snapshot of it. The first statement of the main program (`DrawMode:=Dominant;`) defines the type of operation the program will exhibit. If `DrawMode` equals `Complement`, all lines will complement, because the two lines in the infinite loop (the `while true` loop) which select drawing modes only modify the drawing mode is if is `Dominant` or `Erase`. Otherwise, the drawing mode is not changed. When you wish to change the program to the drawing/erasing mode, change the first statement of the main program to say `DrawMode:=Dominant;`. Then the two drawing-mode-selecting lines will select drawing modes `Erase` and `Dominant`, respectively.

In complementing mode, a pixel is on only if it has been acted upon by an odd number of line segments.

Faster Drawing Procedures

In the previous section, *CRT Drawing Modes*, the routines INT_MOVE and INT_LINE were used for moving and drawing, rather than the MOVE and LINE procedures used previously. The reason for the existence of these routines is that they exhibit higher execution speed. This increase in speed is obtained because the procedures do integer arithmetic, which is much faster than real arithmetic. The only restriction on parameters is that they must be 16-bit signed integers; that is, a subrange of INTEGER whose range is −32 768 through 32 767. There is a TYPE defined in the module `DGL_TYPES` called GSHORTINT which is this subrange of INTEGER.

Depending on the application, they may be up to three times faster than their counterparts which deal with real numbers. However, the increase in speed will only take place if the following three conditions are met:

- The display must be a raster device;
- The window bounds must be within the range of −32 768 through 32 767; and
- The window must be less than 32 767 units wide and high.

There are some more `INT-` routines available also. They are identical to the same routines without the `INT_` at the beginning of their names except for the restriction mentioned above.

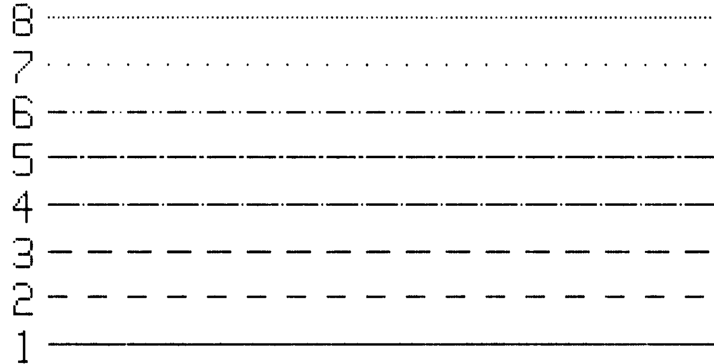
MOVE	→	INT_MOVE
LINE	→	INT_LINE
POLYGON	→	INT_POLYGON
POLYGON_DEV_DEP	→	INT_POLYGON_DD
POLYLINE	→	INT_POLYLINE

Selecting Line Styles

When a graph is attempting to convey several different kinds of information, colors are often used: the red curve signifies one thing, the blue curve signifies another thing, etc. But when only one color is available, as on a monochromatic CRT, this method cannot be used. Something that *can* be used, however, is different line styles. Even on a monochrome CRT, it makes sense to say that the solid line signifies one thing, the dotted line signifies another thing, and the dashed line signifies still another.

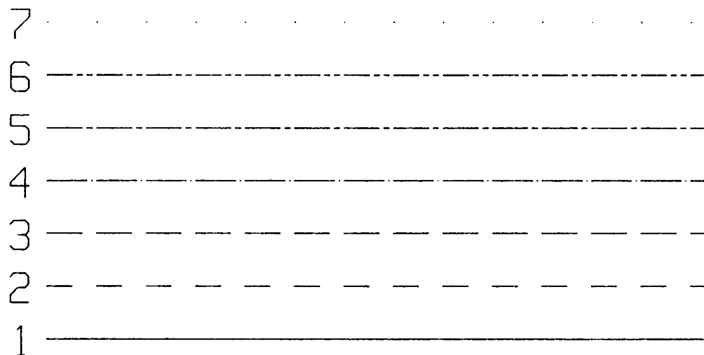
The DGL procedure SET_LINE_STYLE is used to select from the available line styles. The single argument is an integer whose value is 1 through the number of line styles supported on the device currently being used. If using an HP-GL plotter, look under the LT (Line Type) instruction to determine how many line styles are supported.

The CRT supports eight line styles:



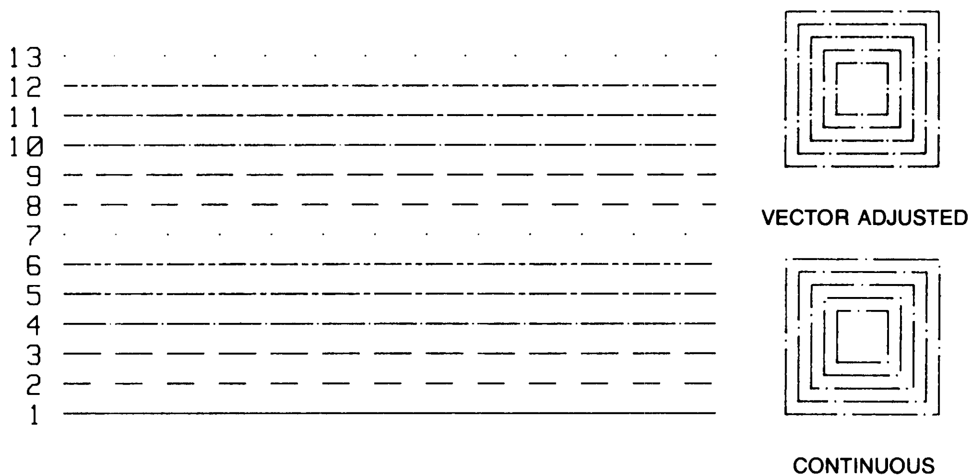
As you can see, line style 1 draws a solid line. Line styles 2 through 8 are patterned sequences of on and off. For all line styles, the computer remembers where in the pattern a line segment ended. Therefore, when you start drawing another line segment, the line pattern will continue from where it left off. If you want the pattern to start over, just re-execute the line style procedure.

Plotters also have different line styles to select from. For example, the following line styles are available on the HP 9872 and HP 7470 plotters.



HP 9872 and 7470 Line Styles

As another example, the HP 7580 and HP 7585 plotters have two different ways of plotting most of their line styles: *continuous* and *vector-adjusted*. Lines drawn with a continuous line style are drawn such that every line segment drawn continues the pattern from where the previous segment left off. If a line segment is short enough and the next section of the pattern is the space between marks, there may be nothing at all drawn for a particular line segment. Vector-adjusted lines are forced to have the middle of the main drawn section at each endpoint of the line segment. See the line segments below.



HP 7580 and 7585 Line Styles

Isotropic Scaling

It was mentioned in the last chapter that there were two different types of scaling: **isotropic** and **anisotropic**. Isotropic scaling means that one unit in the X direction is equal in length to one unit in the Y direction. Anisotropic means that one unit in the X direction does not *necessarily* equal to one unit in the Y direction.

We dealt with anisotropic scaling in the last chapter by calling the DGL procedure SET_WINDOW. For the task we were working on at that time, anisotropic scaling was the best choice. However, when drawing a picture of a object, or drawing a map, is is very desirable to have isotropic scaling, so the representation of the object is not distorted.

There is a way to cause isotropic scaling to be invoked. First, comparisons of the aspect ratios of the viewport limits and the window limits must be made. Then some extra room is allowed in either the X direction or the Y direction (but not both). The amount of extra room is just the precise amount to cause the requested window to be isotropically scaled into the viewport.

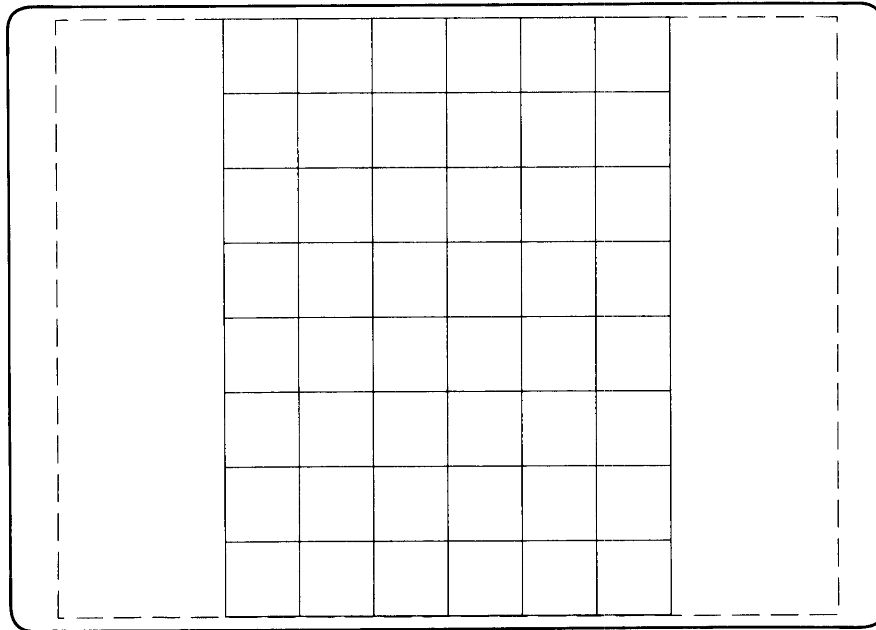
Following is the listing of an algorithm to set a window isotropically.

```
const
  ViewportLimits=        451;            {mnemonic better than magic number}
type
  LimitOrder=    (Vxmin,Vxmax,Vymin,Vymax);
  LimitType=     array [LimitOrder] of real;
var
  Pac:           packed array [1..1] of char;    { \   ...sundry variables      }
  Iarray:        array [1..1] of integer;        {  \ needed by the "inq_ws"    }
  Viewport:      LimitType;                      {  / procedure, called to set }
  Error:         integer;                        { /  window limits.            }
  Wxrange, Wyrange:      real;   {X/Y range in window (world) coordinates}
  Vxrange, Vyrange:      real;   {X/Y range in viewport (virtual) coordinates}
  Wratio, Vratio:        real;   {aspect ratios of window and viewport}
  Wxmid, Wymid:          real;   {X/Y midpoints of window}
  WVratio, VWratio:      real;   {ratios of the ratios}
  Multiplier:            real;   {the amount to multiply the semirange by}
        •
        •
        •
inq_ws(ViewportLimits,0,0,4,Pac,Iarray,Viewport,Error); {set viewport limits}
if Error<>0 then
  writeln('Error ',Error:0,' in procedure "Show".');
Wxrange:=Wxmax-Wxmin;                     {range of X in desired window}
Wyrange:=Wymax-Wymin;                     {range of Y in desired window}
Wratio:=Wxrange/Wyrange;                  {aspect ratio of desired window}
Vxrange:=Viewport[Vxmax]-Viewport[Vxmin];      {range of X in current viewport}
Vyrange:=Viewport[Vymax]-Viewport[Vymin];      {range of Y in current viewport}
Vratio:=Vxrange/Vyrange;                  {aspect ratio of viewport}
if abs(Vratio)<abs(Wratio) then begin     {need more room on top and bottom}
  Wymid:=Wymin+Wyrange*0.5;               {Y midpoint in desired window}
  WVratio:=abs(Wratio/Vratio);            {ratio of aspect ratios}
  Multiplier:=Wyrange*0.5*WVratio;        {what the Y range must be extended by}
  Wymin:=Wymid-Multiplier;                {new minimum Y for window}
  Wymax:=Wymid+Multiplier;                {new maximum Y for window}
end
```

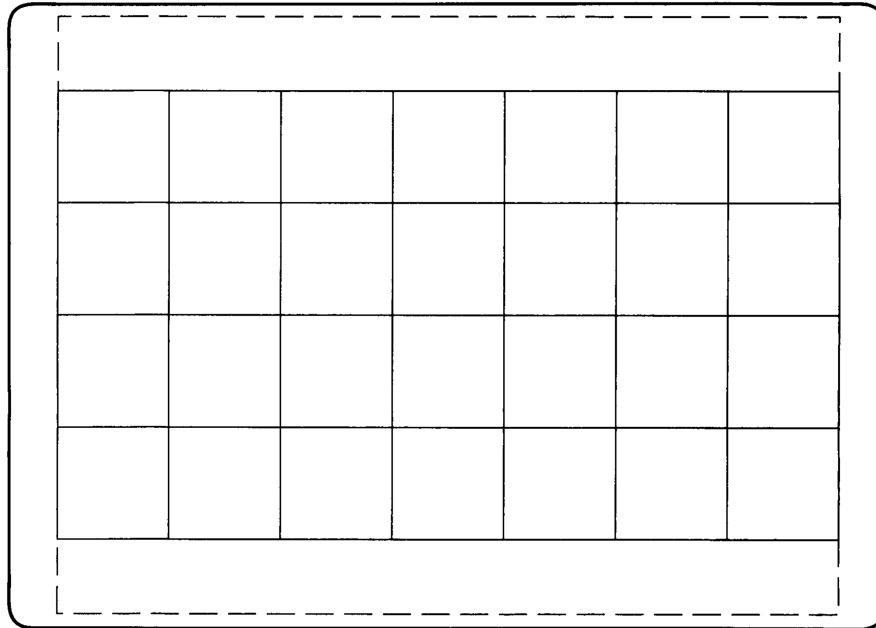
```
else begin                                          {need more room on right and left}
  Wxmid:=Wxmin+Wxrange*0.5;                         {X midpoint in desired window}
  VWratio:=abs(Vratio/Wratio);                      {ratio of aspect ratios}
  Multiplier:=Wxrange*0.5*VWratio;                  {what the X range must be extended by}
  Wxmin:=Wxmid-Multiplier;                          {new minimum X for window}
  Wxmax:=Wxmid+Multiplier;                          {new maximum X for window}
end;  {vratio<wratio?}
set_window(Wxmin,Wxmax,Wymin,Wymax);                {set window with twiddled parameters}
```

Following are two example outputs from the program "IsoProg" (found on a file of the same name on the DGLPRG: disc) which demonstrates the isotropic scaling routine. The user is asked to specify Xmin, Xmax, Ymin, and Ymax for the isotropic units. The specified area is mapped into the viewport area isotropically, adding extra space to either the X or Y direction, whichever is needed. There is a dotted-line frame around the screen limits, and the requested limits are shown in a solid-line grid. The space added is outside the solid-line grid. In both cases, the whole screen was used for the viewport.

In the first example, the requested values were 0 to 6 in X, and 0 to 8 in Y. Since the aspect ratio of this window is less than the aspect ratio of the viewport, some extra room is needed in the X direction, as shown.



In the next example, the requested values were 0 to 7 in X, and 0 to 4 in Y. Since the aspect ratio of this window is greater than the aspect ratio of the viewport, some extra room is needed in the Y direction, as shown.



The program that produced the two preceding outputs is listed in the appendix.

Axes and Grids

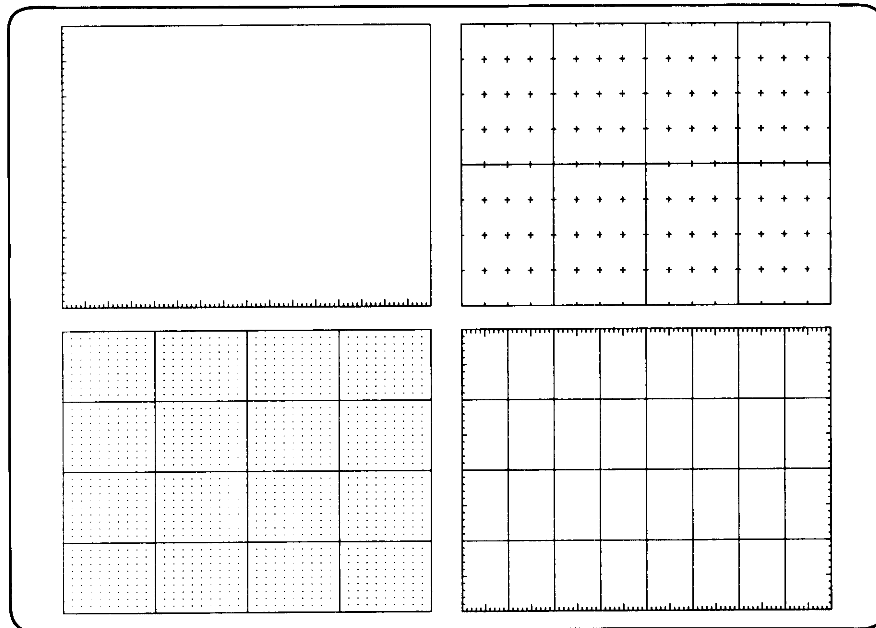
For many data-display graphs, axes along the edges are sufficient to get the message across. But if your graph needs to be read with more precision than axes afford, you can use a grid. A grid is a logical extension to axes, with some differences. The differences are:

- The major tick marks extend all the way across the clip limits, and
- The minor tick marks intersect in small crosses over the entire surface of the soft clip limits.

There is a program called "AxesGrid" on the DGLPRG: disc which will help you understand how to write your own grid-drawing routine. It is similar to the axis procedures, except for the two differences noted above: the major ticks extend across the entire soft clip area (it calls CLIPDRAW), and the minor ticks for X and Y intersect in little crosses between the grid lines.

The following program shows the differences between:

- a pair of axes by themselves,
- a sparse grid,
- a dense grid, and
- a sparse grid with two pair of axes.



Note that some care must be taken to ensure that the minor tick marks in a grid are smaller than the distance between them. If they are not, the minor tick crosses drawn by the grid procedure would have overlapped. The end result would have been a grid with even the minor ticks extending all the way across the soft clip area.

As the lower left graph shows, there is a way to get the best of both worlds—accurate interpolation and lack of clutter. If you want to be able to estimate the data values very accurately from the finished plot, but also want to prevent the plot from appearing too "busy", or cluttered, it can be done. The grid drawn has somewhat sparse major tick marks, but very many minor tick marks. The point of interest is that the minor tick length parameter is reduced to virtually zero. This causes the tick crosses (the little plus signs) to be reduced to mere dots. Using this strategy allows easy interpolation of data points (to the same accuracy as typically used in axes), but does not clutter the graph nearly as much as normal ticks would. In fact, had we used the previous minor tick length, the length of the lines making up the tick crosses would have been greater than the distance between the ticks. Thus, they would have merged together to make solid lines, extending all the way across the graph. This would greatly clutter the graph.

Be aware when using this strategy of making huge numbers of degenerate tick crosses that the computer still thinks of them as crosses, which means that both the horizontal and vertical components must be drawn. This looks to you like drawing and then redrawing each dot. Therefore, when sending this type of grid to a hard-copy plotter, do not be averse to starting your plot, and then going on vacation.

In the lower right quarter of the plot, there is another way to reach a compromise between ease of interpolation and lack of clutter. Axes are used on all four edges, and a sparse grid is drawn with major tick marks every second of the axes' major tick marks.

Note that *two* pairs of axes were drawn. The parameters are identical save for the position of the intersection. The first pair of axes intersect at the lower left corner of the soft clip area. The second pair of axes intersect at the upper right corner of the soft clip area.

Also note that when a grid is drawn, the frame around the window can usually be removed (depending on the Major Tick Count); the lines around the soft clip limits were being drawn by grid procedure anyway.

All of the above have advantages; there is no one approach which is always best. On many occasions, an application is defined such that there is no question as to which procedure to use. Other times, however, it is not such a cut-and-dried situation and you want to weigh the alternatives carefully before setting your program in concrete. To aid you in the decision, here are some pros and cons to the approaches above.

Advantages to axes:

- Axes execute much faster than grids. This is for two reasons. First, there is much less calculating the computer must do, and second (and more important), there is much less actual drawing of lines the computer must do. This becomes especially evident when sending a plot to a hard-copy plotting device where physical pen must be hauled around.
- It does not clutter the plot as much. Reference points are available at the axes, but there is no question about where the data curve is. When using a grid, it is possible to lose the data curve among the reference lines if it is close to being horizontal or vertical.

Advantages to grids:

- Interpolation and estimation are much more accurate due to the great number of reference ticks and lines; one need not estimate horizontal and vertical lines to refer back to the axis labels.
- Usually there is no need to explicitly draw a frame around the grid area to completely enclose the soft clip limits, as is often desired, because the major tick marks from the GRID procedure would probably redraw the lines. Of course, this is dependent upon the Major Tick count.

Logarithmic Plotting

In many fields, there are ranges of valid values which are so large that not only is isotropic scaling out of the question, but any kind of linear scaling—even anisotropic—is virtually useless. To successfully depict these kinds of data, one or both of the axes can be logarithmic scales; that is, the data points themselves are not plotted, but the *logarithm* of each data point is plotted. For example:

- In seismology, earthquake intensity is measured in the logarithmic Richter scale.
- In acoustics, both sound intensity (decibels) and frequency (octaves) are dealt with in logarithmic scales.
- In astronomy, a Hertzsprung-Russell diagram graphs both the luminosities and surface temperatures of stars logarithmically.
- Also in astronomy, black-body radiation curves are plotted logarithmically.

For logarithmic plots, logarithms (from here on referred to as *logs*) to the base 10 are most often used[1].

Homemade Mathematical Functions

To deal with logs, we need to write two mathematical routines which are not provided in the language.

Taking a Number to a Power

First, we need to be able to exponentiate—take an arbitrary number to an arbitrary power. We can use an identity function of logarithms to do this:

$$x^y = e^{y\ ln\,(x)}$$

This is easily done since Pascal does have functions to return the log and antilog[2] in the Napierian[3] base e. The function to return the natural log is `LN`, and the function for returning the natural antilog—e to a power— is `EXP`.

1 An exception to this is the frequency example in acoustics mentioned above, in which octaves deal with powers of two.

2 The $\log_{10} 1000 = 3$ because $10^3 = 1000$. The $\text{antilog}_{10} 3 = 1000$ because $10^3 = 1000$.

3 The Napierian base e is the base of natural logarithms. Its value is $1/0! + 1/1! + 1/2! + 1/3! + 1/4! \ldots$ and equals approximately 2.718 281 828.

The Logarithm to Any Base

The next function is slightly more complex. We needed a function to calculate the common logarithm (log to the base 10). We used another identity function of logarithms which allows one to calculate the log of any positive number to *any* positive base not equal to 1—even fractional ones. We used a special case of this to calculate the common logarithm, or log to the base 10:

$$y = ln(x)/ln(b)$$

Since this allows us to calculate the log of any (positive) number to any (positive) base not equal to 1, we will define the base to be 10. Now we can deal with common logarithms.

Back to Logarithmic Axes...

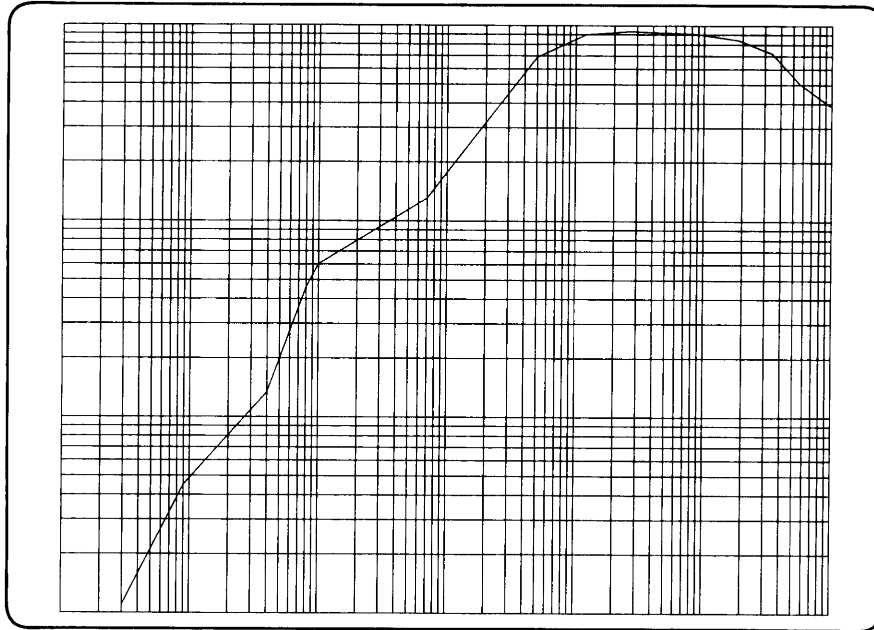
When you are doing logarithmic axes using logs to the base 10, you need to specify the minimum and maximum in *decades*. For example, say you want to make logarithmic axes from 0.01 to 1000. This is 10^{-2} to 10^{3}, therefore, the will be five decades represented. To draw a logarithmic X axis:

```
for Decade:=-2 to 3 do begin
  if Decade<3 then UnitMax:=9
  else UnitMax:=1;
  for Units:=1 to UnitMax do begin
    X:=Decade+Log10(Units);
    move(X,Ymin);
    draw(X,Ymax);
  end;  {for Units}
end;  {for Decade}
```

The statement starting "if Decade<3" is there because we want the units to go from 1 to 9[1] for every decade except the last one, for which we only want the integral power of ten.

1 Each decade goes from 1 to 9, not from 1 to 10, because 10 will be covered by the first iteration on the next decade.

Following is a short program (found on file "LogPlot" on the DGLPRG: disc) which draws logarithmic grid, and plots a curve on it. A logarithmic grid is merely a logarithmic axis with long tick marks.



```
program LogPlot(Keyboard,output);
import dgl_lib;
const
  Xmin=          -4;                  { \                    }
  Xmax=          2;                   {  \  Decade minima    }
  Ymin=          0;                   {  /  and maxima.      }
  Ymax=          3;                   { /                    }
  Crt=           3;                   {device address of graphics raster}
  Control=       0;                   {device control word; ignored for CRT}
type
  RDataType=     array [1..15] of real;
const
  Xvalues=       RDataType[0.0003, 0.0009, 0.004, 0.008, 0.01, 0.07, 0.22, 0.5,
                      1.2, 2.6, 8.9, 18.6, 34, 56, 97];
  Yvalues=       RDataType[1.1, 4.5, 13.38, 45.9, 60.33, 130.7, 346, 690.4,
                      899, 933, 903, 841, 720, 505, 390];
var
  Error:                  integer;        {display_init return variable; 0 = ok}
  Decade:                 integer;
  Units, UpperLimit:      integer;
  X, Y:                   real;
  I:                      integer;
```

```
$page$  {*****************************************************************}
function Log10(X: real): real;
{-----------------------------------------------------------------------}
{  This function returns the logarithm to the base ten of a number.      }
{-----------------------------------------------------------------------}
const
  Log_10=       2.30258509299;          {log to the base e of 10}
begin                                   {function "Log10"}
Log10:=ln(X)/Log_10;
end;                                    {function "Log10"}
$page$  {*****************************************************************}
begin                                   {body of program "LogPlot"}
graphics_init;                          {initialize the graphics system}
display_init(Crt,Control,Error);
if Error=0 then begin
  set_aspect(511,389);
  set_window(Xmin,Xmax,Ymin,Ymax);
  {===== Draw and label logarithmic X-axis grid ============================}
  for Decade:=Xmin to Xmax do begin     {one decade equals one mantissa cycle}
    if Decade=Xmax then UpperLimit:=1
    else UpperLimit:=9;
    for Units:=1 to UpperLimit do begin         {do 2-9 if not last cycle}
      X:=Decade+Log10(Units);
      move(X,Ymin);
      line(X,Ymax);
    end;  {for units}
  end;  {for decade}
  {===== Draw and label logarithmic Y-axis grid ============================}
  for Decade:=Ymin to Ymax do begin     {one decade equals one mantissa cycle}
    if Decade=Ymax then UpperLimit:=1
    else UpperLimit:=9;
    for Units:=1 to UpperLimit do begin         {do 2-9 if not last cycle}
      Y:=Decade+Log10(Units);
      move(Xmin,Y);
      line(Xmax,Y);
    end;  {for units}
  end;  {for decade}
  {===== Draw the logarithmic data curve ===================================}
  for I:=1 to 15 do begin
    if I=1 then move(Log10(XValues[I]),Log10(Yvalues[I]))
    else line(Log10(XValues[I]),Log10(Yvalues[I]));
  end;  {for i}
end;  {Error=0?}                        {end of conditional code}
graphics_term;                          {terminate graphics library}
end.                                    {program "LogPlot"}
```

Storing and Retrieving Images

If a picture on the screen takes a long time to draw, or the image is used often, it may be advisable to store the image itself—*not* the commands used to draw the image—in memory or on a file.

Note

Because the location of the Model 237's frame buffer may vary, storing and retrieving images on the Model 237 is somewhat more complex and exceeds the scope of this manual. Therefore, application of the GSTORE procedure to the Model 237 is not discussed here.

Image transfer from the graphics memory to a user array can be done by overlaying an array directly on top of the graphics memory, i.e., forcing the starting address of a user array to be the same as the starting address of graphics memory. The user array is also the same size as the graphics memory. First, you must have an INTEGER array (32-bit integers) of sufficient size to hold all the data in the graphics raster. This amounts to an array size of 7500[1] on the Models 216, 220 and 226, 6240 on the Models 217 and 236, and 24 960 on the Model 236 Color Computer. This array holds the picture itself, and it doesn't care how the information got to the screen, or in what order the different parts of the picture were produced.

In the following program, the image is drawn with normal plotting commands, and then, *after the fact*, the image is read from the graphics area in memory, and placed into the user array, using the procedure GSTORE. After the array is filled by the GSTORE procedure, a curve is plotted on top of the image already there. Then, turning the knob changes the value of a parameter, and a different curve results. *But we do not have to replot the grid, axes and labels.* We merely need to copy the data containing the image (which has everything but the curve and the current parameter value) back into graphics memory by calling the inverse procedure, GLOAD. This allows the curve to be changed almost in real time. This program is contained in file "GstorProg" on the `DGLPRG:` disc. Note that only the size of the data array must be decreased if this is to work on a Model 216, 220, or 226. If this is to work on a Model 236 Color Computer, both the array size must be increased and (because of the increased array size) it must be accessed dynamically–the `NEW` statement and pointers.

Note that the $SYSPROG ON$ compiler directive must be in the program. The reason for this is that we are using the compiler's ability to force an array to be in a particular area in memory. We declare an integer array whose location in memory is exactly that of the graphics raster memory. Thus, when we deal with the array, we are dealing with the graphics memory, which has the current image in it.

1 The reason the the lower-resolution displays require more memory for image storage than the higher-resolution displays is that the Models 216, 220, and 226 use only the odd bytes of the words. Thus, only the least significant eight bits of each sixteen-bit word are used; the most significant eight bits are zeroes.

To write a program such as this, which stores a graphical image and reloads it, there are several housekeeping things which must be done. Two constants must be defined. First, you must know where in the physical memory of the machine the graphics memory resides[1]:

```
const
  GRasterAddr=        hex('530000');
```

Next, you must know how large the graphics memory is (the size of the graphics raster is expressed in 32-bit words):

```
const
  GRasterSize=        6240;
```

Now, we must declare a type of which the variable being overlaid on the graphics memory will be:

```
type
  GRasterType=        array [1..GRasterSize] of integer;
```

Next, overlay a variable directly on top of the graphics memory. The constant in the brackets immediately after the variable name forces the address of that variable to the specified location in memory. This can only be done if the `$SYSPROG ON$` compiler directive has been encountered. Note that if a Model 236 Color Computer is being used the array size exceeds the total amount of global memory space available, so the variable must be created dynamically; use the `NEW` statement and pointers.

```
var
  GRaster[GRasterAddr]:       GRasterType;
```

And finally, the user's variable into which the graphics memory will be placed. Although it is of the same type as the variable `GRaster` above, we will let the machine figure out where to put it:

```
Screen:                     GRasterType;
```

After all these declaration have been set up, it is a trivial matter to store the graphics image into the user array:

```
Gstore(Screen);
```

Loading a screen image is just as trivial:

```
Gload(Screen);
```

Again, this program is on file "GstorProg" on the `DGLPRG:` disc, and a listing of ther program is in the appendix. It stores and reloads the graphics image to and from a user array. Of course, it also defines the necessary support constants, types, and variables for the GLOAD and GSTORE routines. It draws a blackbody radiation curve for the current temperature.

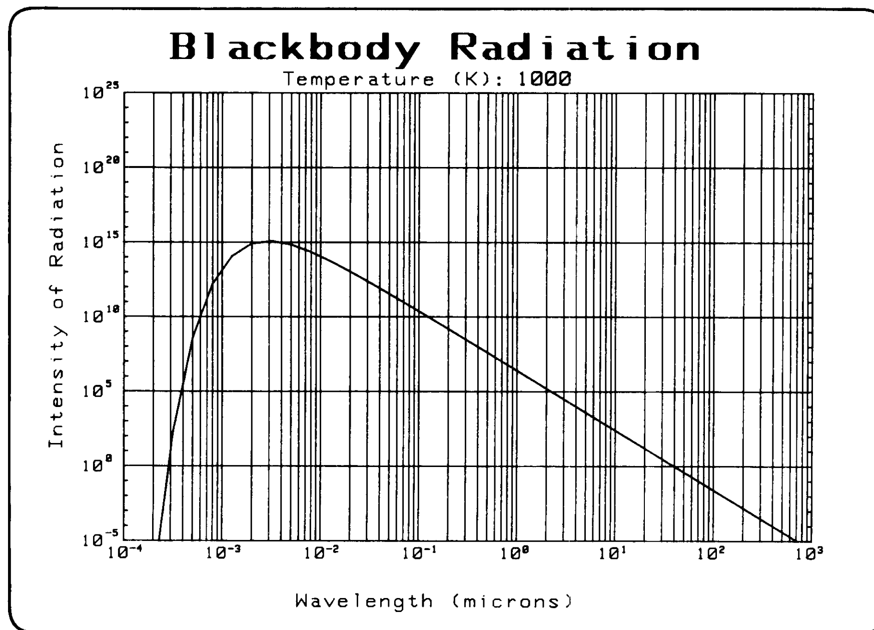
1 Both the size and the address of the graphics memory in your machine is dependent on the model:

	Models 216, 220, and 226	Model 217 and 236	Model 236 Color
Graphics memory address	$530000	$530000	$520000
Graphics memory size	7500	6240	24960

The addresses are expressed in hexadecimal and the sizes are expressed in 32-bit integers.

Note that this program puts into use many of the concepts previously discussed in this chapter:

- Conversion from virtual coordinates to world coordinates;
- Specifying character size with a size in virtual coordinates and an aspect ratio, angular specification of label direction, and label justification;
- Turning the alpha raster off (nonbit-mapped displays)
- Logarithmic axes and grid;
- Image storage and retrieval.

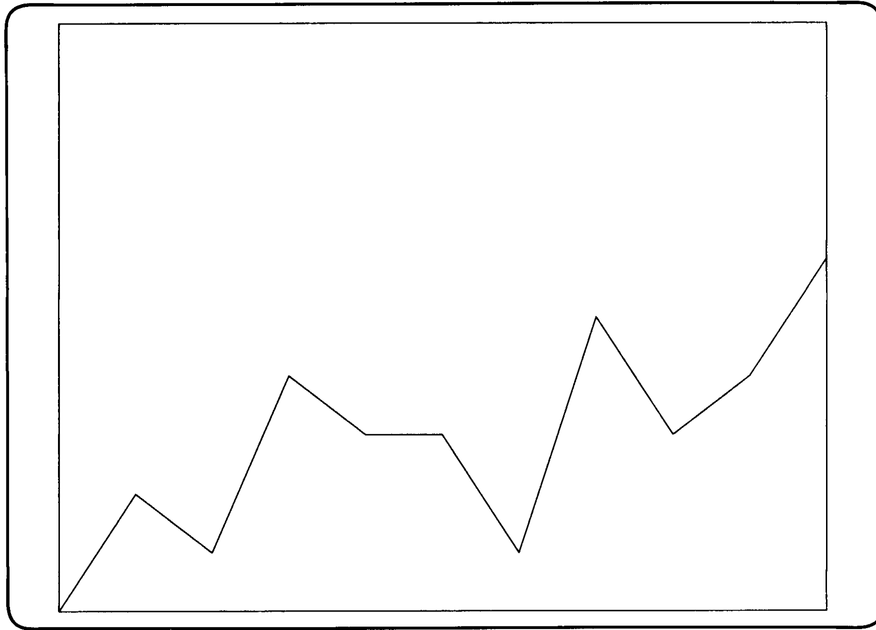


The first time the curve is displayed, it will look like the preceding display. Every time you hit a digit key, a new curve will be drawn, based on the current value of `Temperature`.

Data-Driven Plotting

Many Lines in One Step

In the cases where the data to be plotted is in arrays, it can plotted in one statement by using the POLYLINE procedure. The X data must be in one array, and the Y data in another array. Both arrays must be one-dimensional arrays of reals. Below is a program showing how to plot an X data array versus a Y data array.



```
program PLineProg(output);
import dgl_lib,dgl_inq;
const
  CrtAddr=              3;
  ControlWord=          0;
type
  RDataType=            array [0..10] of real;
const
  Xvalues=              RDataType[0,1,2,3,4,5,6,7,8,9,10];
  Yvalues=              RDataType[0,2,1,4,3,3,1,5,3,4,6];
var
  ErrorReturn:          integer;
  X, Y:                 RDataType;
$page$ {*********************************************************************}
begin                                   {program "PLineProg"}
graphics_init;
display_init(CrtAddr,ControlWord,ErrorReturn);
if ErrorReturn=0 then begin
  set_aspect(511,389);
  set_window(0,10,0,10);
  move(0,0);  line(0,10);  line(10,10);  line(10,0);  line(0,0);
  X:=Xvalues;  Y:=Yvalues;
  polyline(11,X,Y);
end;  {ErrorReturn=0?}
graphics_term;
end.                                    {program "PLineProg"}
```

Note that the X data need not be steadily increasing values so as to make a broken-line chart like above. It could just as easily be used for drawing pictures of objects where both X and Y vary in an unpredictable way. However, if both X and Y are going to change, you'll probably want to be able to control the pen status—you'll want to lift the pen and drop the pen at will.

Drawing Multi-Line Objects

Often, when plotting data points, they do not form a continuous line like the broken-line charts we've seen before. One must have the ability to control the pen's status (up or down), to allow drawing of several different, disconnected parts of an image in one step. For this need, there is a DGL procedure called POLYGON. The fourth parameter in the POLYGON procedure is the operation selector, and its function is to tell the computer to draw or not draw a particular line. It also specifies where individual polygons start.

When plotting an entire array with the polygon statement, the fourth parameter is defined in the following manner. It must be of type INTEGER. The resultant action for the various operation selectors are:

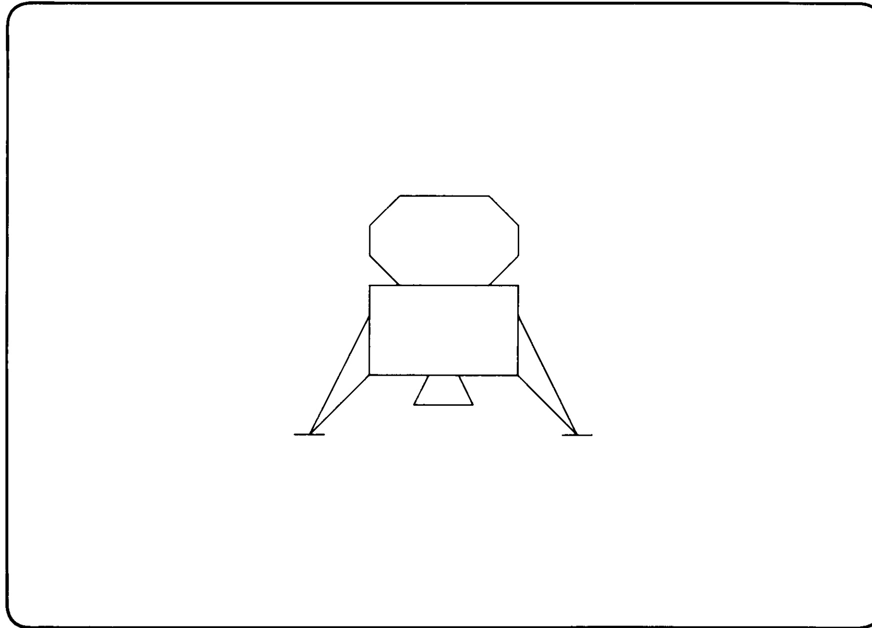
Polygon Operation Selectors

Operation Selector	Resultant Action
0	Do not draw the edge extending from the last vertex to this one.
1	Draw the edge extending from the last vertex to this vertex.
2	This vertex is the first vertex of a member polygon.

Note

Although the POLYGON procedure heading declares the incoming arrays to be of type GREAL_LIST or GSHORTINT_LIST, you cannot declare your own variables this way. Declare your variables a your own data type: arrays of the appropriate size of reals and short integers (16-bit integers: i.e., `-32768..32767`). If you import the module `DGL_TYPES`, you can use the type `GSHORTINT`.

Following is a program (file "PolyProg" on `DGLPRG:` disc) which uses the POLYGON procedure. It draws a LEM (Lunar Excursion Module). The first parameter specifies how many points there are in the arrays. There are three arrays used: two one-column REAL arrays for the X and Y data, and a one-column INTEGER array containing opcodes.



```
program PolyProg(output);                       {program name same as file name}
import
  dgl_lib,dgl_types,dgl_poly,dgl_inq;           {access the necessary procedures}
const
  MaxPoints=            27;                     {number of points in arrays}
  Crt=                  3;                      {device address of graphics raster}
  Control=              0;                      {device control word; ignored for CRT}
type
  Reals=         array [1..MaxPoints] of real;   {to contain X and Y values}
  Word=          -32768..32767;                  {16-bit word}
  Integers=      array [1..MaxPoints] of Word;      {to contain op. selectors}
const
  Xvalues=       Reals[ 1.5, 2.5, 2.5, 1.5,-1.5,-2.5,-2.5,-1.5,  {Octagon}
                       -2.5, 2.5, 2.5,-2.5,-2.5,                 {Box}
                       -2.5,-4.5,-2.5,-5.0,-4.0,                 {Left leg}
                        2.5, 4.5, 2.5, 5.0, 4.0,                 {Right leg}
                       -0.5,-1.0, 1.0, 0.5];                     {Nozzle}
  Yvalues=       Reals[ 1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0,  {Octagon}
                        1.0, 1.0,-2.0,-2.0, 1.0,                 {Box}
                       -2.0,-4.0, 0.0,-4.0,-4.0,                 {Left leg}
                       -2.0,-4.0, 0.0,-4.0,-4.0,                 {Right leg}
                       -2.0,-3.0,-3.0,-2.0];                     {Nozzle}
  OpCodes=       Integers[2,1,1,1,1,1,1,1,                       {Octagon}
                         2,1,1,1,1,                              {Box}
                         2,1,1,2,1,                              {Right leg}
                         2,1,1,2,1,                              {Left leg}
                         2,1,1,1];                               {Nozzle}
var
  Error:                integer;        {display_init return variable; 0 = ok}
  I:                    integer;        {loop control variable}
  LemX, LemY:           Reals;          {so we can pass it to "polygon"}
  OpSelectors:          Integers;       {ditto}
  Points:               integer;        {ditto}
```

```
$page$  {*********************************************************************}
begin                                 {body of program "PolyProg"}
LemX:=Xvalues;                        { \  Put into variable array so  }
LemY:=Yvalues;                        {  > it can be passed by         }
OpSelectors:=OpCodes;                 { /  reference into the DGL proc.}
Points:=MaxPoints;                    {put constant into an array variable}
graphics_init;                        {initialize graphics library}
display_init(Crt,Control,Error);      {initialize CRT}
if Error=0 then begin                 {if no error occurred...}
  set_aspect(511,389);                {use the whole screen}
  set_window(-13,13,-10,10);          {invoke isotropic units}
  polygon(Points,LemX,LemY,OpSelectors);    {draw the lines}
end;  {Error=0?}                      {end of conditional code}
graphics_term;                        {terminate graphics library}
end.  {program "PolyProg"}            {end of program}
```

What's in a Polygon?

That's a good question, and brings up the crucial difference between POLYGON and POLYGON_DEV_DEP (as well as the INT versions of the same). The key to understanding the two classes of polygon is the concept of device independence. Polygons generated by procedures that lack the DEV_DEP (or DD) suffix are device independent, and will always be filled, with as close to the fill specified by the polygon table (lines or crosshatched lines at some specified density) as the device they are being drawn on is capable of producing. Thus the lines used for a fill on a CRT may have visible jaggies, and the lines used on a 7580 plotter will not, but both of them will produce polygons filled with lines.

So what happens with POLYGON_DEV_DEP? The "DEV_DEP" calls specify a device dependent polygon. The fastest, most appropriate fill possible on the device is used to fill a polygon. On the CRT, this is a dithered area fill (dithering is discussed in detail in the "Color" chapter). On the plotter, the edge is drawn with the current line color attribute if edge is specifed in the operation selector array and enabled in the polygon table. If the polygon edge attribute is false and the operation selector edge attribute is true, the polygon edge is drawn with the current polygon interior color attribute and polygon linestyle. It is worth noting that in this case, if the current polygon interior color attribute is set to 0 (the background in the color table), the polygon will not be visible.

When to Use Which Polygon?

Why are there two polygon fills? The two polygon calls address different valuable characteristics of the system. The POLYGON call tries to give a consistent representation, regardless of what display device is being used. The POLYGON_DEV_DEP calls are faster. You give up consistency by using the device dependent calls, as well as control of drawing mode (all device dependent polygons are drawn in the dominant writing mode). The choice is yours: if you want speed or control of drawing mode, use the device dependent calls—if you want consistant presentation of the image and/or control of the drawing mode, use the device independent call.

Polygon Filling

When drawing a polygon, whether it is filled or not is an *attribute* of the polygon. The filling attribute itself has other attributes; namely, method (dithered/hatched), density (0-100%), and, if hatched, hatching direction ($-90°$-$90°$) and perpendiculars (true/false).

Polygons can be filled two different ways. Filling allows you to shade the polygons to various shades of gray.

The first method of filling is to draw lines across the polygons; crosshatching. This is selected with the SET_PGN_STYLE procedure. Various densities of shading can be achieved through crosshatching, depending on both of the following:

- The amount of space between the crosshatching lines; and
- Whether or not there are perpendiculars.

The other method of shading on a monochromatic CRT is called **dithering**. Dithering is a more accurate way to get shades of gray on a black-and-white CRT whose electron gun is either fully on or completely off. Dithering is accomplished through selecting small groups of pixels[1], a four-by-four square of them on the Series 200 computers. Various pixels in the dithering box are turned on and off to arrive at an "average" shade of gray. There are only seventeen possible shades because out of sixteen pixels (the 4 × 4 box); you can have none of them on, one of them on, two of them on, and so forth, up to all sixteen of them on. And it makes no difference *which* pixels are on; the pattern for each level is chosen to minimize the striped or polka-dotted pattern inherent to a dithered image.

Crosshatching is accomplished by drawing many lines, and lines are drawn taking into account the current drawing mode (dominant, erase, or complement). One reason that this is important is that you can draw a complementing cursor with a call to the POLYGON procedure. Dithering does not deal with lines, therefore, the current drawing mode is ignored when doing a dithered fill.

Note

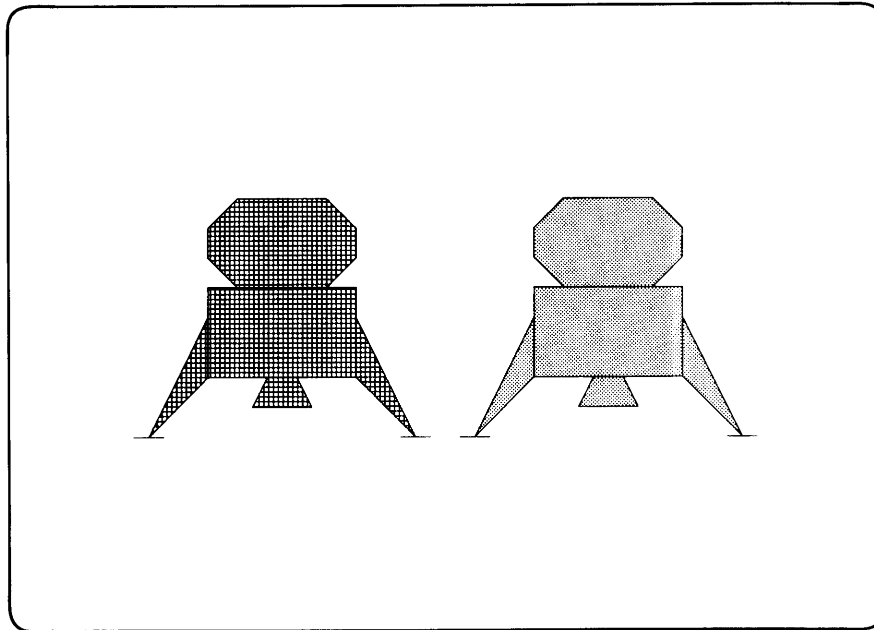
Polygons to be filled which extended over the edge of the plotting surface are completely filled-including all the area off the plotting surface. If a great deal of the polygon is invisible, then, it may appear that the machine is hung[2], but in reality, it is merely doing a lot of calculations which do not affect the visible image at all.

---

1 The word "pixel" is a blend of the two words "picture element," and it is the smallest addressable point on a plotting surface. A Model 236 computer has 512 × 390-pixel resolution; thus there can be no more than 512 dots drawn on any row of the CRT, or 390 dots drawn in any column.

2 "Hung," in this context, is short for "hung up." It is a computerese term which means that the machine has entered a state, usually unanticipated, in which the machine becomes unresponsive, and drastic measures are often required to correct it.

Here is another program which draws the LEM, and fills the polygons in two different ways. On the left, it is filled by crosshatching; on the right, it is filled by dithering.

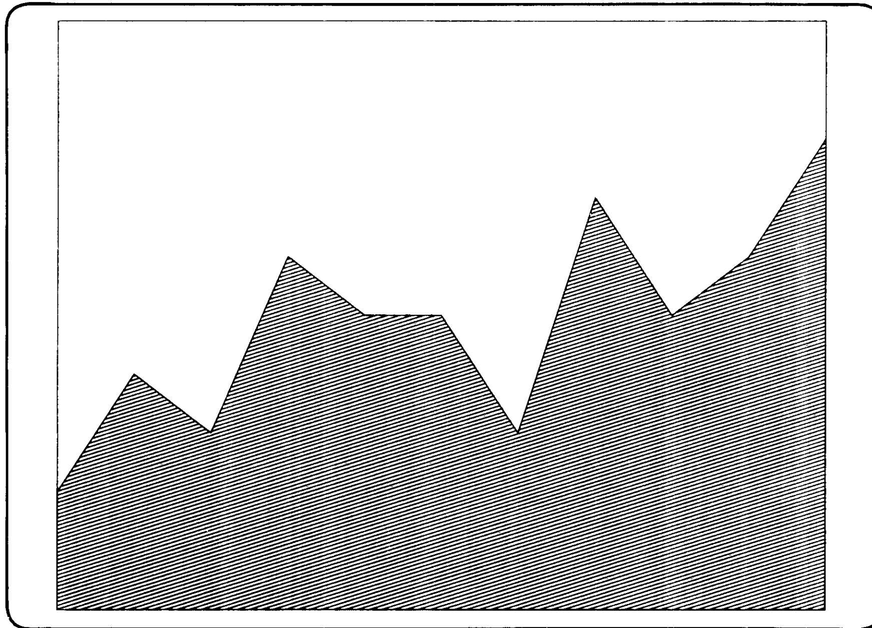


```
program FillProg(output);                  {program name same as file name}
import
  dgl_lib,dgl_types,dgl_poly,dgl_inq;     {access the necessary procedures}
const
  MaxPoints=          27;                  {number of points in arrays}
  Crt=                3;                   {device address of graphics raster}
  Control=            0;                   {device control word; ignored for CRT}
type
  Reals=       array [1..MaxPoints] of real;    {to contain X and Y values}
  Word=        -32768..32767;                   {16-bit word}
  Integers=    array [1..MaxPoints] of Word;      {to contain op. selectors}
const
  Xvalues=     Reals[ 1.5, 2.5, 2.5, 1.5,-1.5,-2.5,-2.5,-1.5,  {Octagon}
                     -2.5, 2.5, 2.5,-2.5,-2.5,                  {Box}
                     -2.5,-4.5,-2.5,-5.0,-4.0,                  {Left leg}
                      2.5, 4.5, 2.5, 5.0, 4.0,                  {Right leg}
                     -0.5,-1.0, 1.0, 0.5];                      {Nozzle}
  Yvalues=     Reals[ 1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0,  {Octagon}
                      1.0, 1.0,-2.0,-2.0, 1.0,                  {Box}
                     -2.0,-4.0, 0.0,-4.0,-4.0,                  {Left leg}
                     -2.0,-4.0, 0.0,-4.0,-4.0,                  {Right leg}
                     -2.0,-3.0,-3.0,-2.0];                      {Nozzle}
  OpCodes=     Integers[2,1,1,1,1,1,1,1,                        {Octagon}
                        2,1,1,1,1,                              {Box}
                        2,1,1,2,1,                              {Right leg}
                        2,1,1,2,1,                              {Left leg}
                        2,1,1,1];                               {Nozzle}
var
  Error:              integer;        {display_init return variable; 0 = ok}
  I:                  integer;        {loop control variable}
  LemX, LemY:         Reals;          {so we can pass it to "polygon"}
  OpSelectors:        Integers;       {ditto}
  Points:             integer;        {ditto}
```

```
$page$  {*******************************************************************}
begin                                         {body of program "FillProg"}
LemX:=Xvalues;                                { \  Put into variable array so  }
LemY:=Yvalues;                                {   > it can be passed by         }
OpSelectors:=OpCodes;                         { /  reference into the DGL proc.}
Points:=MaxPoints;                            {put constant into an array variable}
graphics_init;                                {initialize graphics library}
display_init(Crt,Control,Error);              {initialize CRT}
if Error=0 then begin                         {if no error occurred...}
  set_aspect(511,389);                        {use the whole screen}
  set_window(-7.5,18.5,-10,10);               {invoke isotropic units}
  set_pgn_style(14);                          {crosshatched fill}
  polygon(Points,LemX,LemY,OpSelectors);        {draw the lines}
  set_window(-18.5,7,-10,10);                {invoke isotropic units}
  set_pgn_table(14,0.51,0,1);                 {set the "do a fill" flag}
  set_color_table(1,0.125,0.125,0.125);       {specify 12.5% gray scale}
  set_pgn_color(1);                           {use specified "color"}
  polygon_dev_dep(Points,LemX,LemY,OpSelectors);        {draw the lines}
end;  {Error=0?}                              {end of conditional code}
graphics_term;                                {terminate graphics library}
end.  {program "FillProg"}                    {end of program}
```

Shading Graphs

Two previously-mentioned concepts can be combined to make broken-line charts which are filled. That is, you can consider the curve on the graph as edges of a polygon (along with the lower corners of the viewport), and fill the area with shading. Below is a short program which demonstrates the combined concepts. The program is found on file "FillGraph" on the DGLPRG: disc.



```
program FillGraph(output);
import dgl_lib, dgl_types, dgl_poly;
const
  CrtAddr=              3;
  ControlWord=          0;
type
  RDataType=            array [0..12] of real;
  WDataType=            array [0..12] of -32768..32767;
const
  Xvalues=              RDataType[0,1,2,3,4,5,6,7,8,9,10,10,0];
  Yvalues=              RDataType[2,4,3,6,5,5,3,7,5,6,8,0,0];
  OperationSelectors=   WDataType[2,1,1,1,1,1,1,1,1,1,1,1,1];
var
  ErrorReturn:          integer;
  X, Y:                 RDataType;
  OpSel:                WDataType;
```

```
$page$  {*******************************************************************}
begin                                   {program "FillGraph"}
graphics_init;
display_init(CrtAddr,ControlWord,ErrorReturn);
if ErrorReturn=0 then begin
  set_aspect(511,389);
  set_window(0,10,0,10);
  move(0,0);  line(0,10);  line(10,10);  line(10,0);  line(0,0);
  X:=Xvalues;    Y:=Yvalues;    OpSel:=OperationSelectors;
  set_pgn_table(1,0.333,17.34,1);
  set_pgn_style(1);
  polygon(13,X,Y,OpSel);
end;  {ErrorReturn=0?}
graphics_term;
end.                                    {program "FillGraph"}
```

Note that the two lower corners of the graph must be included in the definition of the polygon. The shading is done with hatching lines, and the angle of those lines is deliberately a strange angle to point out that you are not restricted to multiples of 45° for the hatching lines. If the plot is to come out on a CRT, dithering may be used instead.

If the shading is going to be done with hatching lines, you may want to perform a linear regression on the data points. Then, you can indicate the overall trend of the data by defining the slope of the hatching lines to be the angle determined by the linear regression.

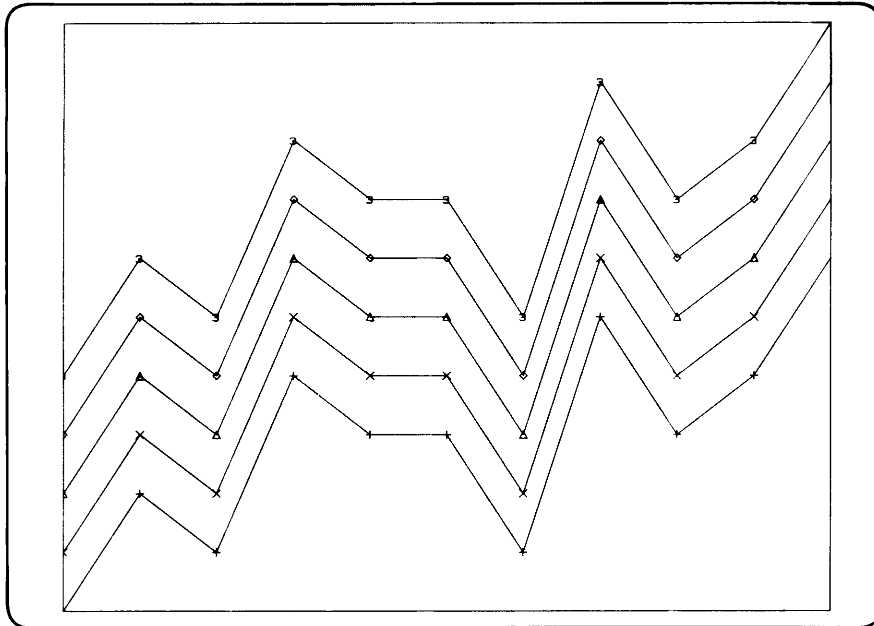
Highlighting Data Curves

You can note the location of the starting points of line segments by using the MARKER procedure. When the procedure is called, it outputs a marker of whatever type you selected. The valid values and what types of markers they output are listed below:

Marker Number	Resulting Shape	Marker Number	Resulting Shape
<1	•	10	0
1	•	11	1
2	+	12	2
3	*	13	3
4	O	14	4
5	X	15	5
6	△	16	6
7	□	17	7
8	◇	18	8
9	⊞	19	9

Marker numbers greater than 20 are device dependent. If the specified marker is larger than the number of marker the device supports, a dot (marker selector 1) will be used.

Below is a program and its output which shows how to use the MARKER procedure. The program can be found on the file "MarkrProg" on the DGLPRG: disc.



```
program MarkrProg(output);
import dgl_lib,dgl_inq;
const
  CrtAddr=            3;
  ControlWord=        0;
type
  MarkerNumType=      array [0..4] of integer;
  DataType=           array [0..10] of integer;
const
  MarkerNumber=       MarkerNumType[2,5,6,8,13];
  Data=               DataType[0,2,1,4,3,3,1,5,3,4,6];
var
  ErrorReturn:        integer;
  I, J:               integer;
$page$  {*****************************************************************}
begin                                  {program "MarkrProg"}
graphics_init;
display_init(CrtAddr,ControlWord,ErrorReturn);
if ErrorReturn=0 then begin
  set_aspect(511,389);
  set_window(0,10,0,10);
  move(0,0);  line(0,10);  line(10,10);  line(10,0);  line(0,0);
  for I:=0 to 4 do begin
    for J:=0 to 10 do begin
      if J<>0 then marker(MarkerNumber[I]);
      if J=0 then move(J,Data[J]+I)
      else line(J,Data[J]+I);
    end;  {for j}
  end;  {for i}
end;  {ErrorReturn=0?}
graphics_term;
end.                                   {program "MarkrProg"}
```

External Graphics Displays and Plotters

Chapter 3

In this chapter, we will be discussing the selection of external plotting devices. The DISPLAY_INIT procedure will be more thoroughly covered, in addition to dumping graphics images from a CRT to a printer. External CRTs (cathode-ray tubes), which may be connected to your computer through a 98627A interface card, and plotters, which may be connected through the built-in HP-IB (Hewlett-Packard Interface Bus) port in the back of your computer, will also be discussed.

Selecting a Plotter

In the previous two chapters, the program listings contained a line which said:

```
display_init(CrtAddr,ControlWord,ErrorReturn);
```

Because the value contained in the variable `CrtAddr` was 3–specifying the current console– the computer activated the internal CRT graphics raster as the plotting device, and all subsequent graphics output was directed to this display. If you want a plotter to be the output device, only the value of the variable `CrtAddr` need be changed. (You may also want to change the name of the variable. It is somewhat misleading to have the address of a plotter in variable named `CrtAddr`.) If your plotter is at interface select code 7 and address 5 (the factory settings), the modification would be:

```
CrtAddr:=705;
```

Dumping Raster Images

In addition to generating a hard-copy plot with a plotter, as described above, you can dump a CRT's raster image to a printer. This method is called a *graphics dump* or *screen dump*. It is accomplished by copying data from the frame buffer to a printer to be printed dot for dot.

First, the image must be drawn on a CRT. Either the internal CRT or a color monitor connected by an HP98627A interface card may be used. Since this technique dumps a raster-type image, it prints only dots. Thus, it cannot draw a line, per se, but only the approximation of a line from the screen, made up of dots. The dump device "takes a snapshot" of the graphics screen at some point in time, and doesn't care *how* the dots came to be turned on or off. Thus, filled areas can be dumped to the printer; indeed, all CRT graphics capabilities (except color) are available.

If your printer is an HP 9876, HP 2631G, HP 2671G, HP 2673A or any other printer which conforms to the HP Raster Interface Standard, dumping a graphics image is achieved with the OUTPUT_ESC procedure. If your active graphics display device (set with the DISPLAY_INIT procedure call) is monochromatic, a call to OUTPUT_ESC with operation selector 52 will dump the display if:

- The active graphics display is the console (where alpha is displayed), or
- The active graphics display is bit-mapped (i.e., is a Model 237 display or a display connected via the HP 98627A RGB interface).

If you have a color device, all planes in the frame buffer are logically ORed. If you want more control over the output of a color image, an operation selector of 1053 will allow you to select individual planes from the frame buffer. The 1053 operation selector will work with the Model 236C, the Model 237 bit-mapped display, or with a color display connected via the HP 98627A RGB interface. Since the Model 237 has only one plane, the plane designator is ignored.

The exception to producing a desirable image via this method occurs if your active CRT is a bit-mapped display that supports more pixels than your printer has dots. In this case, the dump starts at the upper left-hand corner of the screen and dumps as far to the right and down as there are corresponding dots on the printer.

Both of these operation selectors sent to OUTPUT_ESC would take the image in the currently active CRT graphics frame buffer (the internal CRT by default) and send it to volume `PRINTER:`. By default the printer is assumed to be at select code 7, bus address 1. This can be changed by modifying the `CTABLE.TEXT` program on the `CONFIG` disc. Find the line:

```
local_printer_default_dav = dav[sc: 7, ba:1, du:-1, dv:-1];
```

This sets the DAV (device address vector) for the printer to be at select code (`sc`) 7 and bus address (`ba`) 1. By changing this line, you can alter the destination of data sent to the volume `PRINTER:`[1]. 701 is the default factory setting for printers.

If a graphics dump operation is aborted with the (STOP) key, the printer may or may not terminate its graphics mode.

1 For an in-depth coverage of how to modify the `CTABLE.TEXT` program, see the *Special Configurations* chapter of the *Pascal 3.0 Workstation System Manual*.

If you have a printer which does not conform to the HP Raster Interface Standard, all is not lost. It must, however, be capable of printing raster-image bit patterns. There are two main methods by which printers output bit sequences. The first is: when a printer receives a series of bits, it prints them in a one-pixel-high line across the screen. The paper then advances one pixel's distance, and the next line is printed. The other method (which lends itself to user-defined *characters* more than graphics image dumping) takes a series of bits, breaks it up into 8-bit chunks, and prints them as vertical bars 8 pixels high and one pixel wide. The next eight bits compose the next 1 × 8-pixel bar, and so forth.

This latter method is that used by the HP82905 printer. The image (which is printed out sideways) takes a GSTOREd image and breaks the 16-bit integers into two 8-bit bytes, and sends them to the printer one row at a time. Writing your own routine to dump a graphics image to a non-comforming printer should not be difficult, given the ability of taking the graphics image and placing into your own data array (referred to in the last chapter).

Note that on a CRT, an "on" pixel is light on an otherwise dark background, and on a printer, an "on" pixel is dark on an otherwise light background. Thus, the hard copy is a negative image of that on the screen. To dump light images on a dark background, you can invert every bit in the stored image. To invert the bits in a 32-bit integer, you can execute the following code segment:

```
if N=minint then
  N:=maxint
else
  N:=-N-1;
```

The reason for the subtraction is that Series 200 computers use twos-complement representation of integers. Also, you must consider MININT[1] as a special case because you cannot negate MININT in an integer; +2 147 483 648 cannot be represented in a signed thirty-two bit twos-complement number.

1 MININT and MAXINT are two standard constants in HP Pascal. MININT = -2^{31} = −2 147 483 648, and MAXINT = $2^{31}-1$ = +2 147 483 647.

External Color Displays

The HP 98627A RGB interface allows you to connect a color monitor to your computer, whether the computer's internal CRT supports color or not. The HP 98627A does not, as mentioned before, support color map operations; thus, you cannot change the color of an area on the screen without redrawing the area. Nor can you define your own color-addition scheme as you can with a color-mapped device (see the Color Graphics chapter). In addition to this, there are only eight pure colors[1]; to get others, you must go to dithering.

There are many types of color monitors which you can connect to your computer through an HP 98627A color monitor interface. In the `ControlWord` variable which is passed to the DISPLAY_INIT procedure, you must specify accordingly:

Desired Display Format	Description	Bits 10-8
Standard Graphics		
512 by 390 pixels, 60 Hz, non-interlaced	U.S. Standard	001 (256)
512 by 390 pixels, 50 Hz, non-interlaced	European Standard	010 (512)
TV Compatible Graphics		
512 by 474 pixels, 60 Hz, interlaced (30 Hz refresh rate)	U.S. Television	011 (768)
512 by 512 pixels, 50 Hz, interlaced (25 Hz refresh rate)	European Television	100 (1024)
High-Resolution Graphics		
512 by 512 pixels, 46.5 Hz, non-interlaced	High Resolution	101 (1280)
HP Use Only	Internal	110 (1536)

Out of range values are treated as if `ControlWord` = 256, as is `ControlWord` = 0 (except Model 237, where 0 keeps the type-ahead buffer, and 256 removes it).

1 Only eight pure colors can be created on an external color monitor. This is because there is no control over the intensity of each color gun. Each color can be either off or on, and there are three colors: red, green, and blue. Two states, three colors: $2^3 = 8$.

External Plotter Control

There are many device-dependent operations you can do through calling the OUTPUT_ESC procedure. See Appendix B for details on all the things you can do.

Controlling Pen Speed

To improve the quality of the lines drawn by a plotter pen, you may want to make them draw more slowly. There are other factors, too, which can affect line quality. For example, humidity can alter the line quality of a fiber-tipped pen. To accomplish this, you can call the OUTPUT_ESC procedure with the appropriate parameters. Or, the following procedure will do it.

```
{*************************************************************************}
procedure PenSpeed(Speed: integer);
{-------------------------------------------------------------------------}
{  This procedure selects a pen speed for ah HPGL plotter.                }
{-------------------------------------------------------------------------}
const
  SetPenSpeed=  2050;          {a mnemonic is better than a magic number}
var
  Iarray:       array [1..2] of integer;        { \  These are variables    }
  Rarray:       array [1..1] of real;           {  > needed by the DGL      }
  Error:        integer;                        { /  procedure "output_esc" }
begin                                  {procedure "PenSpeed"}
Iarray[1]:=Speed;                      {use the passed parameter}
Iarray[2]:=0;                          {affect all pens}
output_esc(SetPenSpeed,2,0,Iarray,Rarray,Error);
if Error<>0 then                       {error?}
  writeln('Error ',Error:0,' in procedure PenSpeed".');
end;                                   {procedure "PenSpeed"}
```

The first element of the integer array specifies the pen speed; the range and resolution of pen speeds, and default maximum speed depend on the plotter. The second element of the array specifies the pens to be affected. One through eight specifies pens one through eight, respectively. Any value outside of this range is taken to mean, "Affect all pens."

Selecting a pen speed specifies a *maximum* speed rather than an *only* speed, because on short line segments, the pen does not have time to accelerate to the specified speed before the midpoint of the line segment is reached and deceleration must begin.

This procedure also provides a skeleton for making other special-purpose routines. For most operations dealing with OUTPUT_ESC, one need only change the name of the procedure and the parameters being passed to the OUTPUT_ESC procedure.

Controlling Pen Acceleration

On the HP 7580, HP 7585 and HP 7586 drafting plotters, you can specify the amount of acceleration the pen is to undergo when starting or ending a line. On any particular line, positive acceleration (speeding up) will occur until one of two things happens:

- The midpoint of the line is reached, and negative acceleration (slowing down, or deceleration) must begin, to ensure that the pen will reach a speed of zero precisely at the second endpoint of the line it's drawing; or
- The specified maximum speed is reached. In this case, that speed will be maintained until the pen is at a particular distance from the second endpoint of the line. At that distance, which depends on the specified maximum speed and the specified acceleration, the pen will start to smoothly decelerate such that it will reach zero velocity at the second endpoint.

The first element of the integer array passed to OUTPUT_ESC specifies the pen acceleration; it may range from one through four gees[1]. The second specifies the pens to be affected. One through eight specifies pens one through eight, respectively. Any value outside of this range is taken to mean, "Affect all pens."

Controlling Pen Force

On many drafting plotters (e.g., HP 7580, HP 7585, HP 7586), you can specify the amount of force pressing the pen tip to the drawing medium. This is useful when matching a pen type (ball-point, fiber-tip, drafting pens, etc.) to a drawing medium (paper, vellum, mylar, etc.). Again, if a pen is partially dried out, it may help line quality to adjust the pen force.

The PenSpeed procedure mentioned above can be modified slightly to control pen force. The operation selector should be 2051. The first element of the integer array specifies the pen force; the second specifies the pens to be affected. One through eight specifies pens one through eight, respectively. Any value outside of this range is taken to mean, "Affect all pens."

The force number is translated into a force in grams. If, for example, you have an HP7580A plotter, the force number is converted to force as follows:

1 = 10 grams	5 = 42 grams
2 = 18 grams	6 = 50 grams
3 = 26 grams	7 = 58 grams
4 = 34 grams	8 = 66 grams

This is not by any means an exhaustive list of the things you can do with OUTPUT_ESC, but it serves to acquaint you with the concept of using the procedure for controlling device-dependent operations. A thorough understanding of its use can only be gotten by combining information from the DGL Language Reference with actual hands-on experience.

1 One "gee," or one [earth] "gravity," is the acceleration due to gravity at sea level. Its value is approximately 9.8 m/sec^2 or 32 ft/sec^2.

Interactive Graphics

Chapter 4

Introduction

It has already been pointed out that graphics is a very powerful tool for communication. The high speed available from Series 200 computers makes possible a powerful mechanism for communicating with the computer: *Interactive Graphics.*

The best way to understand interactive graphics is to see it in action. Compile and execute the program "BAR_KNOB", from your "DGLPRG:" disc. If you turn the knob clockwise, the bar graph displayed on the screen will indicate a larger value. At the same time, the numeric readout underneath the bar will increase it's value. Turning the knob counterclockwise has the opposite effect. (If your computer has no knob, the arrow keys or mouse will work, but may not feel as "natural.") This is an effective demonstration of all the key characteristics of an interactive graphics system. They are:

- A data structure. (The value displayed underneath the bar is the contents of a variable that we are modifying. The internal variable containing the value is a degenerate case of a data structure.)
- A graphic display that represents the contents of the data structure. (The bar graph and the numeric display represent the value of the internal variable.)
- An input mechanism for interacting with the displayed image (the knob, in this case.)

This is the minimum set of requirements for an interactive graphics system. A key feature of interactive graphics is that it is a *closed loop* system. This means that the operator can immediately see the effect of his action on the system, and thus base his next action not only on the state of the system, but also on the effect his last action had on the system. A few points are worth noting about this system:

- The knob is used because it is *functionally appropriate.* While we could have entered numeric values to control the bar graph, the knob "feels" right. We are used to using knobs to control metered readouts.
- Control of the value with the knob is fairly intuitive. The normal range markings make it readily apparent when the value is in range. Little explanation is needed, due to the immediate feedback from the displayed image.
- A system is "modeled." The user's input has a well defined relation to the output of the system.

Thus, interactive graphics can be as simple as representing a single value on the screen and providing the user a method for interacting with it. It can also be as complex as a Printed Circuit layout system. This chapter will not tell you how to build a Printed Circuit layout system, but it will provide some hints on implementing interactive graphics systems that work.

Characterizing Graphic Interactivity

One of the most important things in designing a good interactive graphics system is characterizing the interaction with the system correctly. Properly characterizing the interactivity allows selection of the most appropriate device for interaction with the system. Three things have to be considered in characterizing the interaction:

- The number of *degrees of freedom* in the system. This is the number of ways in which a system can be changed.
- The *quality* of each of the degrees of freedom. This describes how the input to a degree of freedom can be changed.
- The *separability* of the degrees of freedom.

Once again, the best way to understand the characterization of interaction is to see an example in action. Compile and execute "BAR_KNOB2" from your "DGLPRG:" disc. This program is very similar to "BAR_KNOB", but it has several bars, instead of one. This introduces another degree of freedom to the model. The original program had a single degree of freedom, the value indicated by the bar graph. The quality of this degree of freedom is continuous. The new program has the same continuous input (which is still handled by the knob) but has added a second degree of freedom, the selection of the bar graph you want to modify. This degree of freedom is quantizable, and is handled by the numeric keys. (Softkeys would be even better, but require digging into the operating system.) The degrees of freedom are also separable, since you don't need to interact with both of them at once.

The degrees of freedom are not separable in freehand drawing -you want to change X and Y simultaneously. They are only partially separable in laying out images on a screen - you can get by with moving along one axis at a time, but it's easier if you can interact with both of them at once.

Selecting Input Devices

The purpose of the discussion on characterization of graphic interaction was to lay the groundwork for discussing when various input devices are appropriate. There are several available to the Series 200 computers, and choosing the correct one is critical to the design of a highly productive human interface for an interactive graphics program.

Single Degree of Freedom

Many interactive graphics programs need deal only with a single degree of freedom. The appropriate control device for such programs depends on whether continuous control or quantized control is needed.

The program "BAR_KNOB" is a good example of a single degree of freedom that is continuous. The knob is ideal for controlling a program like this. If "fine tuning" is needed, the shift key can be used as a multiplier to change the interpretation of the knob. The knob is read through the KEYBOARD file. The knob generates forward and back spaces for clockwise and counterclockwise motion, or line-feeds and up-spaces if the shift key is held down while the knob is turned. The following program ("BAR_KNOB" from the "DGLPRG:" disc) shows how to interpret the knob for a continuous, single degree of freedom, as well as how to update the display to show the results of the interaction.

```
$ucsd,debug$
program Test (Keyboard,output);
import dgl_vars,dgl_types,dgl_lib,dgl_inq;
type
  States=               (On,Off);
  DrawMode=             (Draw,Erase,Comp,NonDom);
const
  FS=                   chr(28);
  BS=                   chr(8);
  US=                   chr(31);
  LF=                   chr(10);
  CR=                   chr(13);
  Q=                    'Q';
  Q1=                   'q';
  Underline=            chr(132);
  Ind_off=              chr(128);
  Inv_On=               chr(129);
  MinBarY=              0;
  MaxBarY=              100;
  MinBarX=              180;
  MaxBarX=              220;
  IncDelta=             0.1;
var
  Error_num:            integer;
  I,TempInt:            integer;
  Level,LastLevel:      real;
  Delta:                real;
  CharWidth,CharHeight: real;
  Character:            char;
  Done:                 boolean;
  Keyboard:             text;
  TempString:           Gstring255;
```

```
$page$ {*****************************************************************}
procedure GraphicsDisplay(State:States {On/Off});
const
  GraphicsDisp=         1050;
var
  Error:integer;
  SwitchArray:integer;
  Dummy:real;
begin                                   {procedure GraphicsDisplay}
case State of
  On:SwitchArray:=1;
  Off:SwitchArray:=0;
end; {case State of}
output_esc(GraphicsDisp,1,0,SwitchArray,Dummy,Error) ;
if Error <> 0 then
  writeln ('Error ',Error:1,' encountered in GraphicsDisplay');
end;                                    {procedure GraphicsDisplay}
$page$ {*****************************************************************}
procedure AlphaDisplay(State:States {On/Off});
const
  AlphaDisp=1051;
var
  Error:integer;
  SwitchArray:integer;
  Dummy:real;
begin                                   {procedure AlphaDisplay}
case State of
  On:SwitchArray:=1;
  Off:SwitchArray:=0;
end; {case State of}
output_esc(AlphaDisp,1,0,SwitchArray,Dummy,Error) ;
if Error <> 0 then
  writeln ('Error ',Error:1,' encountered in AlphaDisplay');
end;                                    {procedure AlphaDisplay}
$page$ {*****************************************************************}
begin                                   {Main Program}
Level:=0;                               {current height of bar}
LastLevel:=Level;                       {previous height of bar}
graphics_init;                          {initialize the graphics system}
display_init(3,0,Error_Num);            {which output device?}
if Error_Num=0 then begin               {output device initialization OK?}
  AlphaDisplay(Off);                    {turn off alpha display}
  GraphicsDisplay(On);                  {turn on graphics display}
  set_aspect(511,389);                  {use whole screen}
  set_window(0,400,-30,120);            {scale the window for the data}
  set_color(1);                         {color number 1: white}
  CharWidth:=(0.035*400);               {char width: 3.5% of screen width}
  CharHeight:=(0.05*150);               {char height: 5% of screen height}
  set_char_size(CharWidth, CharHeight); {install character size}
  {---- Outline the Bar -----------------------------------------------}
  move(MinBarX-0.5,MinBarY-0.5);        {move to lower left corner...}
  line(MinBarX-0.5,MaxBarY+0.5);        {...draw to upper left corner...}
  line(MaxBarX+0.5,MaxBarY+0.5);        {...draw to upper right corner...}
  line(MaxBarX+0.5,MinBarY-0.5);        {...draw to lower left corner...}
  line(MinBarX-0.5,MinBarY-0.5);        {...and draw to lower left corner.}
  {---- Label the bar (numeric labels) --------------------------------}
```

```
for I:=0 to 10 do begin
  strwrite(TempString,1,TempInt,I*10:3,'-');
  move (179-strlen(TempString)*CharWidth,I*10-0.24*CharHeight);
  gtext (TempString);
end; {for I:=1 to 10 }
{---- Label the bar (textual labels) ------------------------------------}
move (221, 80-CharHeight/2);
gtext ('-High Normal');
move (221, 60-CharHeight/2);
gtext ('-Low Normal');
{---- How about some instructions ----------------------------------------}
CharWidth:=(0.02*400);              {char width: 2% of screen width}
CharHeight:=(0.035*150);            {char height: 3.5% of screen height}
set_char_size(CharWidth, CharHeight); {install character size}
move (0, 5);
TempString:='Use the Knob to'+CR+LF;
gtext (TempString);
TempString:='Adjust the value.'+CR+LF;
gtext (TempString);
TempString:=' '+CR+LF;
gtext (TempString);
TempString:='SHIFT with the Knob '+CR+LF;
gtext (TempString);
TempString:='speeds it up.'+CR+LF;
gtext (TempString);
TempString:='';
{---- Set a good character size ------------------------------------------}
CharWidth:=(0.035*400);             {char width: 3.5% of screen width}
CharHeight:=(0.05*150);             {char height: 5% of screen height}
set_char_size(CharWidth, CharHeight); {install character size}
repeat
  read(Keyboard,Character);         {get character without echo to screen}
  Delta:=0;                         {start by assuming no motion}
  case Character of                 {what's the character?}
    FS:   Delta:=IncDelta;          {right arrow?}
    BS:   Delta:=-IncDelta;         {left arrow (backspace)?}
    LF:   Delta:=10*IncDelta;       {down arrow?}
    US:   Delta:=-10*IncDelta;      {up arrow?}
    Q,Q1: Done:=TRUE;               {or Quit?}
  otherwise                         {if none of the above, ignore it}
  end; {case ord(Character)}
  if Delta>0 then begin             {Going Up}
    set_color(1);                   {we want to draw lines}
    while (Level<LastLevel+Delta) and (Level<MaxBarY-IncDelta) do begin
      Level:=Level+IncDelta;        {new top of bar}
      move(MinBarX,Level);          {move to left edge...}
      line(MaxBarX,Level);          {...and draw to right edge}
    end  {while (Level<LastLevel) and (Level<MaxBarY)}
  end  {if (Delta>0) and (Level<100) }
  else begin                        {Going Down}
    if (Delta<0) and (Level>=0.5*IncDelta) then begin
      set_color(0);                 {we want to erase lines}
      repeat
        move(MinBarX, Level);       {move to the left edge...}
        line(MaxBarX, Level);       {...and draw to the right edge}
        Level:=Level-IncDelta;      {new top of bar}
      until (Level<=LastLevel+Delta) or (Level<=MinBarY)
    end;  {if (Delta<0) and (Level>0)}
  end;
```

```
    {---- How about some numbers? -----------------------------------------}
    set_color(0);                       {we want to erase lines}
    strwrite(TempString,1,TempInt,LastLevel:5:1);   {convert level to chars}
    move(MinBarX+(MaxBarX-MinBarX)/2-strlen(TempString)*CharWidth/2,
      MinBarY-2*CharHeight);
    gtext(TempString);                  {erase the old number}
    set_color(1);                       {we want to erase lines}
    strwrite(TempString,1,TempInt,Level:5:1);
    move (MinBarX+(MaxBarX-MinBarX)/2-strlen(TempString)*CharWidth/2,
      MinBarY-2*CharHeight);
    gtext(TempString);                  {write the new}
    LastLevel:=Level;                   {remember the old number}
  until Done;                           {repeat until user hits [Q]}
  GraphicsDisplay (Off);                {turn off graphics display}
  AlphaDisplay (On);                    {turn on alpha display}
  display_term;                         {clean up loose ends}
end;
graphics_term;                          {terminate the graphics package}
end.                                    {main program}
```

Keys can be used for quantizable control of a degree of freedom. It is also possible to use keyboard entry of numeric values for quantizable information.

Non-separable Degrees of Freedom

One characteristic of multiple, non-separable degrees of freedom is that they are generally continuous. The most common operation of this type is free-hand drawing. This is most easily accomplished with the 9111A graphics tablet.

Separable Degrees Of Freedom

In many programs, the degrees of freedom are completely separable. In fact, for some operations, it is definitely preferable to have totally independent control of the degrees of freedom of the model.

All Continuous

If all the degrees of freedom in a model are continuous, then the selection of the degree of freedom to operate on becomes another degree of freedom, and is quantizable. A good choice is using the keyboard to select the degree of freedom and then using the knob to control the input to that degree of freedom. This is not as effective as a bank of knobs, but adding a bank of knobs means adding hardware (a voltmeter, power supplies, and potentiometers). The program "BAR_KNOB2", on the "DGLPRG:" disc is an example of this type of interaction. Single keystrokes are used to select the degree of freedom you are operating on, and then the knob is used to vary the value along that degree of freedom. The following key interpretation loop is used in "BAR_KNOB2" to allow the user to select the bar to be controlled, as well as controlling the value of the selected knob.

```
READ (KEYBOARD,Character);
   Delta := 0;
   CASE Character OF
      FS       : Delta := IncDelta;
      BS       : Delta :=-IncDelta;
      LF       : Delta := 10*IncDelta;
      US       : Delta :=-10*IncDelta;
      Q,Q1     : Done  := TRUE;
      '1'..'5': BEGIN
                  ClearInd(Bar);
                  Bar := ORD (Character)- ORD('0');
                  SetInd(Bar);
                END;
   OTHERWISE
   END; {CASE Character}
```

All Quantizable
If all the degrees are quantizable, using the keyboard (or using softkeys if you have requisite system design experience to use them) is appropriate.

Mixed Modes
In most sophisticated graphics systems, several degrees of freedom in the system interact with each other. A good example is a graphics editor. In a graphics editor, your primary interaction is with a visual image, and the degrees of freedom (X and Y location) for that operation are partially separable, at best. (They are non-separable if it supports freehand drawing.) There is also a degree of freedom involved in controlling the program. The program control is strongly separable from the image creation operation.

The most appropriate device for supporting mixed modes is the HP 9111A Graphics Tablet. The tablet supports two modes of interaction by partitioning the digitizing surface into two areas. Sixteen small squares along the top of the tablet can be used as softkeys to provide a control menu. The large, framed area underneath the softkeys is the active digitizing area. The active digitizing area is used for interacting with the image you are creating. Other menu/ image area combinations are also possible.

It is possible to combine the quantized, separable control operations with continuous, non-separable image editing. This is done by using the active digitizing area for interacting with the image and using the menu area for controlling the operations available in the editing program. The operator does not have to change control devices to access the different interaction modes.

Echoes

An important part of interactive graphics is letting the operator know "where he is at." This can be done by updating the image. In other operations, such as menu selection, object positioning, and freehand drawing, it is important to show the operator where he is. In many cases, this can be done with AWAIT_LOCATOR.

The Built In Echo

Many graphics applications can be handled using the built-in echo. AWAIT_LOCATOR allows you to access one of the built-in echoes for digitizing. The following program interprets a menu to select one of the built-in echoes, and then draws an appropriate image on the CRT after the call to AWAIT_LOCATOR completes. It is on your DGLPRG: disc, in the file called "LOCATOR." If you have an HP 9111 Graphics Tablet, changing the constant `LocatorAddress` from 2 to 706 will allow you to use the tablet for a locator instead of the knob or mouse.

```
$debug$
program Test(output);
import dgl_vars,dgl_types,dgl_lib,dgl_poly,dgl_inq;
type
  Commands=             0..8;           {nine commands total}
  RealArray=            array [1..5] of real;
const
  FS=                   chr(28);        {right arrow}
  BS=                   chr(8);         {left arrow or backspace}
  US=                   chr(31);        {up arrow}
  LF=                   chr(10);        {down arrow}
  CR=                   chr(13);        {carriage return}
  MinX=                 0;              {minimum X value for screen}
  MinY=                 0;              {minimum Y value for screen}
  MaxX=                 511;            {maximum X value for screen}
  MaxY=                 389;            {maximum Y value for screen}
  Xrange=               MaxX-MinX;      {total range of X}
  Yrange=               MaxY-MinY;      {total range of Y}
  LocatorAddress=       2;              {2 for Knob,706 for 9111}
var
  Error_num:            integer;        {error return variable}
  I,TempInt:            integer;        {utility variables}
  ButtonValue:          integer;        {which button selected?}
  Xin,Yin:              real;           {location of digitized point}
  Xlast,Ylast:          real;           {last digitized point}
  CharWidth,CharHeight: real;           {char size in world coords}
  Done:                 boolean;        {are we supposed to quit?}
  NewLine:              boolean;        {start new line?}
  TempString:           Gstring255;     {utility variable}
  EchoSelect,EchoSelector:     0..9;    {menu selection}
  MenuTop:              real;
  CellWidth:            real;           {width of menu spaces}
  Command:              Commands;       {which command selected?}
$page$  {*******************************************************************}
procedure DrawMenu;
var
  I:                    integer;        {loop-control variable}
  Ylabel:               real;           {Y position of entree label}
  Yarray:               RealArray;
```

```
{---------------------------------------------------------------------------}
procedure MenuCell(I:integer);
var
  TempPitch:            real;          {temporary variable}
  Xlabel:               real;          {X position of entree label}
  Xarray:               RealArray;     {X positions of entree cell}
begin                                  {procedure MenuCell}
case I of
  0: begin
      TempString:='STOP';              {label text}
      Xarray[1]:=0;                    { \                           }
      Xarray[2]:=2*CellWidth;          {  \                          }
      Xarray[3]:=2*CellWidth;          {   > X positions for box     }
      Xarray[4]:=0;                    {  /                          }
      Xarray[5]:=0;                    { /                           }
      Xlabel:=MinX+CellWidth-strlen(TempString)*CharWidth/2;
    end;
  1..10: begin
      TempPitch:=CellWidth*I;          {temporary shorthand variable}
      Xarray[1]:=CellWidth+TempPitch;  { \                           }
      Xarray[2]:=2*CellWidth+TempPitch; {  \                          }
      Xarray[3]:=2*CellWidth+TempPitch; {   > X positions for box     }
      Xarray[4]:=CellWidth+TempPitch;  {  /                          }
      Xarray[5]:=CellWidth+TempPitch;  { /                           }
      TempString:=' ';                 {label text}
      if I<=8 then strwrite(TempString,1,TempInt,I:1);
      Xlabel:=Xarray[1]+CellWidth/2+strlen(TempString)*CharWidth/2;
    end
end; {case I of}
polyline(5,Xarray,Yarray);             {draw perimeter of cell}
move(Xlabel,Ylabel);                   {move to the right place}
gtext(TempString);                     {label the text}
end;                                   {procedure MenuCell}
{---------------------------------------------------------------------------}
begin                                  {procedure DrawMenu}
Yarray[1]:=MinY;                       { \                           }
Yarray[2]:=MinY;                       {  \                          }
Yarray[3]:=MenuTop;                    {   > Y values for box        }
Yarray[4]:=MenuTop;                    {  /                          }
Yarray[5]:=MinY;                       { /                           }
Ylabel:=MinY+(MenuTop-MinY)/2-CharHeight/2;    {Y position of label}
for I:=0 to 10 do MenuCell(I);         {do all the entree cells}
end;  {procedure DrawMenu}
$page$  {******************************************************************}
function CheckMenu(Xin:real):Commands;
begin                                  {function CheckMenu}
if Xin<2*CellWidth then CheckMenu:=0   {X outside of menu?}
else begin
  TempInt:=trunc((Xin-CellWidth)/CellWidth);    {which sell chosen?}
  if TempInt>8 then CheckMenu:=Command
  else CheckMenu:=TempInt
end;
end;   {function  CheckMenu }
```

```
$page$  {*******************************************************************}
begin                                   {Main program}
graphics_init;                          {initialize the graphics system}
display_init(3,0,Error_Num);            {which output device?}
if Error_Num<>0 then begin              {output devic initialization OK?}
  writeln('I failed to initialize the display.');
  writeln('Error number ',Error_Num:2,' was returned.');
end {if Error_Num<>0}
else begin
  LOCATOR_init(LocatorAddress,Error_Num);
  if Error_Num<>0 then begin
    writeln('I failed to initialize the locator.');
    writeln('Error number ',Error_Num:2,' was returned.');
  end  {if Error_Num<>0}
  else begin                            {No errors so far}
    set_aspect(511,389);                {use whole screen}
    set_window(0,511,0,389);            {scale window for data}
    CharWidth:=0.035*511;               {char width: 3.5% of screen width}
    CharHeight:=0.05*389;               {char height: 5% of screen height}
    set_char_size(CharWidth,CharHeight);{install character size}
    MenuTop:=Yrange/13;                 {menu is 1/13 the total screen height}
    CellWidth:=Xrange/12;               {each entree cell 1/12 screen width}
    DrawMenu;                           {draw the menu}
    NewLine:=true;                      {yes, we are starting a new line}
    EchoSelect:=4;                      {start program with default command}
    Command:=4;                         {ditto}
    Done:=false;                        {no, we're not done yet}
    repeat
      if NewLine then                   {starting a new line?}
        EchoSelector:=2
      else
        EchoSelector:=EchoSelect;
      await_locator(EchoSelector,ButtonValue,Xin,Yin);
      if Yin<MenuTop then begin         {user choose menu option?}
        NewLine:=true;                  {start a new line next time}
        Command:=CheckMenu(Xin);        {determine menu selection}
        case Command of                 {which command}
          0: Done:=true;                {yes, we're done with the program}
          1: EchoSelect:=1;             { \                              }
          2: EchoSelect:=2;             {  \                             }
          3: EchoSelect:=3;             {   \                            }
          4: EchoSelect:=4;             {    \  Select the appropriate   }
          5: EchoSelect:=5;             {    /  EchoSelector.            }
          6: EchoSelect:=6;             {   /                            }
          7: EchoSelect:=7;             {  /                             }
          8: EchoSelect:=8;             { /                              }
        end  {case}
      end  {if}
      else begin                        {not  a menu selection}
        if NewLine then begin           {start a new line}
          NewLine:=false;               {now we're in the middle of a line}
          set_echo_pos(Xin,Yin);        {move the graphics cursor}
          move(Xin,Yin);                {cause line-drawing to start there}
          Ylast:=Yin;                   {remember the last X...}
          Xlast:=Xin;                   {...and the last Y}
        end
```

```
        else begin
          set_echo_pos(Xin,Yin);          {move the graphics cursor}
          if (Xin=Xlast) and (Yin=Ylast) then NewLine:=true
          else begin
            case EchoSelect of
              1..7: line(Xin,Yin);        {draw a line}
              8: begin
                  line(Xlast,Yin);
                  line(Xin,Yin);
                  line(Xin,Ylast);
                  line(Xlast,Ylast);
                  NewLine:=true;
                end
              otherwise
            end; {case EchoSelect of}
            Xlast:=Xin;                   {remember the last X...}
            Ylast:=Yin;                   {...and the last Y}
          end
        end;
      end;
    until Done;                           {are we done yet?}
    locator_term;                         {terminate the locator}
    display_term;                         {terminate the display}
  end; {Error trap}
end;
graphics_term;                            {terminate the graphics system}
end.                                      {Main program}
```

Rubber Echoes

If you have run the progam "LOCATOR," you will have seen that several of the echoes are *rubber band* echoes, in other words, they create lines that seem to stretch between various points on the screen. Echoes 4 through 8 require two points to define them. One of these points is the point being tracked with the AWAIT_LOCATOR statement. The other is the *anchor* point, and is set using the SET_ECHO statement. After using one of the rubber band echoes, and drawing the figure it represents, it is necessary to get a new point to anchor the next echo to. This is done in the program "LOCATOR" by the following block of code:

```
IF NewLine THEN BEGIN
   NewLine := FALSE;
   SET_ECHO_POS (Xin,Yin);
   MOVE (Xin,Yin);
   Ylast:= Yin;
   Xlast:= Xin;
END
```

```
ELSE BEGIN
   SET_ECHO_POS (Xin,Yin);
   IF (Xin = Xlast) AND (Yin = Ylast) THEN
      NewLine := TRUE
   ELSE BEGIN
      CASE EchoSelect OF
         1..7: LINE (Xin,Yin) ;
         8   : BEGIN
                  LINE (Xlast, Yin);
                  LINE (Xin, Yin);
                  LINE (Xin, Ylast);
                  LINE (Xlast, Ylast);
                  NewLine := TRUE;
               END
      OTHERWISE
      END; {CASE EchoSelect of}
      Xlast := Xin;
      Ylast := Yin;
   END
END;
```

In the preceding code, the anchor is set to the last digitized point, *unless* the same point was digitized twice, in which case the small croshair cursor can be used to select a new anchor point. Once a new anchor point is selected, the rubber band cursor mode is returned to.

When the knob is being used as a locator, it is also possible to use SET_ECHO to establish the initial position of the locator when AWAIT_LOCATOR is called.

Tablets and Aspect Ratios

If the knob is used as a locator for the CRT, the mapping between the locator device and the display device is isotropic, since the two devices use the same display mechanism. This is not true if an external digitizing device (such as the HP 9111A Graphics Tablet) is used. The default aspect ratio for the 9111A is 0.7234, while the CRT of the Model 236 ≈0.7613 (as set up in "LOCATOR," above). This means that a square area on the graphics tablet does not represent a square area on the CRT. This is not a tremendous problem in many interactive graphics programs, where the tablet is merely used to point at objects. However, in some applications, those in which the tablet is used to copy an existing document into the computer, the distortion is not acceptable. This is easly remedied, through the SET_LOCATOR_LIM procedure. The following addition to the "LOCATOR" program will set the tablet to the same aspect ratio as the CRT, insuring the desired isomorphic transformation.

```
ELSE BEGIN {No errors so far}
   SET_ASPECT (511,389);
   IF LocatorAddress = 706 THEN BEGIN{This is a tablet}
      SET_LOCATOR_LIM(0,(511/389)*217.6,0,217.6,Error_num);
      IF Error_num <> 0 THEN
         WRITELN (Error_num:2,' encountered in SET_LOCATOR_LIM.');
   END; {IF LocatorAddress = 706}
   SET_WINDOW (0,511,0,389);
```

Color Graphics

Chapter 5

Color !

Color can be used for emphasis, for clarity, and just to present visually pleasing images. Color is a very powerful tool, and it follows directly that it is very easy to misuse. Be careful in using color, and it will serve as a valuable tool for communication. Misuse it, and it will garble the communication.

The DGL Color System

In order to create a device independent programming language, it is necessary to model an ideal system, and then create transformations to map that system onto real hardware. This is the way the Device independent Graphics Library (DGL) works. Understanding the ideal color system will make it much easier to understand the actual implementations that are available on Series 200 computers.

In order to understand the color system, it is necessary to understand two concepts:

- Color as an Attribute
- Models for Color Specification

After covering these topics, we will also go into the concept of a color space, which is another way of describing the color models that are used in DGL.

Color As An Attribute

We have already dealt with the attribute of linestyle, and the attributes which describe the fill pattern in a polygon. Color is another primitive attribute. Two procedures in DGL allow you to specify the attribute of color:

- SET_COLOR selects the color used by GTEXT, LINE, INT_LINE, POLYLINE and INT_POLYLINE, as well as the edges generated by POLYGON, POLYGON_DD, INT_POLYGON and INT_POLYGON_DD.
- SET_PGN_COLOR selects the color used for the interior of polygons generated by POLYGON, POLYGON_DD, INT_POLYGON and INT_POLYGON_DD.

Notice that SET_COLOR and SET_PGN_COLOR both *select* a color attribute. The selection is made from the *color table.*

The Color Table

The color table is a repository of color definitions to be used for displaying primitives. It is used to describe both lines and filled areas. The color table is a list of 32 colors, providing 32 colors for the color attribute of graphics primitives.

Default Colors

When DGL is initialized for a color display (the 98627A or the Model 236 Color Computer), the color table is set up with the following values:

Default Color Table Values

Value	Color
0	Black
1	White
2	Red
3	Yellow
4	Green
5	Cyan
6	Blue
7	Magenta
8	Black
9	Olive Green
10	Aqua
11	Royal Blue
12	Maroon
13	Brick Red
14	Orange
15	Brown
16 thru 31	White

The Primary Colors

The lower eight pens are the colors of the default color map; the colors that can be created by turning the guns of a color CRT on or off, in various combinations :

- Black and White (the extremes of no-color)
- Red, Green, and Blue (the additive primaries)
- Cyan, Magenta, and Yellow (the complements of the additive primaries - which happen to be the subtractive primaries)

The Business Colors

The upper 8 colors (8 through 15) were selected by a graphic designer to produce graphs and charts for business applications. The colors are:

- Maroon, Brick Red, Orange, and Brown (warm colors)
- Black, Olive Green, Aqua, Royal Blue (cool colors)

These colors are one designer's idea of appropriate colors for business charts and graphs. They were chosen to avoid clashing with each other. A technique for using them is described under "Color Hard Copy" in the "Color Spaces" section at the end of this chapter.

Monochromatic Defaults

If a monochromatic display device is being used, the color table defaults to a set of dithered gray patterns:

Default Monochromatic Color Table Values

Value	Luminosity
0	0.0000
1	1.0000
2	0.9375
3	0.8750
4	0.8125
5	0.7500
6	0.6875
7	0.6250
8	0.5625
9	0.5000
10	0.4375
11	0.3750
12	0.3125
13	0.2500
14	0.1875
15	0.1250
16	0.0625
17 thru 31	1.0000

If You Don't Like the Defaults

The contents of an entry in the color table can be modified with the procedure SET_COLOR_TABLE. The actual effect of a call to SET_COLOR_TABLE depends on the color model being used. The color model is selected using SET_COLOR_MODEL. Which brings us to color specification.

Models For Color Specification

As mentioned above, SET_COLOR_TABLE is used to control the actual value of entries in the color table. It was also pointed out that the effect of SET_COLOR_TABLE is determined by the current color model, which is controlled by SET_COLOR_MODEL. It follows that it is necessary to understand SET_COLOR_MODEL before it is possible to understand SET_COLOR_TABLE.

SET_COLOR_MODEL selects (if you haven't already guessed) the color model to be used. There are two models available in DGL: the RGB (Red, Green, Blue) and the HSL (Hue, Saturation, Luminosity) models. We will discuss them in order of ascending complexity.

The RGB Model (Red, Green, Blue)

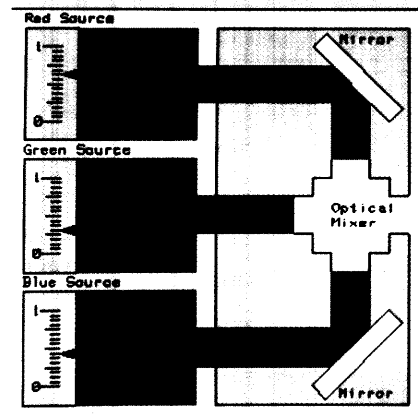
The RGB model can be thought of as mixing the output of three light sources (one each of Red, Green, and Blue). The parameters in the model specify the intensity of each of the light sources. The RGB model is selected by using a model selector of 1:

```
SET_COLOR_MODEL (1);
```

Once the RGB color model has been selected, the parameters sent to SET_COLOR_TABLE represent the percentage of full intensity of the red, green, and blue light sources:

```
SET_COLOR_TABLE (TableEntry, Red, Green, Blue);
```

The following picture illustrates a physical model for the RGB system.



RGB Color Model

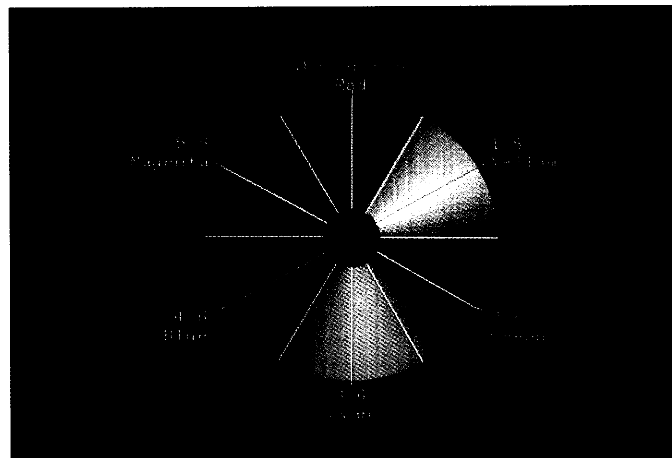
Whenever the red, green, and blue parameters have the same value, the resulting color is a gray tone (i.e. it has no hue component). The RGB model is based on the additive primaries, the colors used for describing mixing light, as opposed to mixing pigments, which are subtractive. It is a good system for interacting with color CRT displays, since it requires little conversion to translate it to a set of signals suitable for driving a color CRT.

The HSL Model (Hue, Saturation, Luminosity)

The HSL model is closer to the intuitive model of color used by artists, and is very effective for interactive color selection. It is similar in concept to the methods used by artists for mixing paints, where pure hues are selected, and then white and black are mixed to dilute the color and/or darken it. The three parameters represent *hue* (the pure color to be worked with), *saturation* (the ratio of the pure color mixed with white), and *luminosity* (the brightness-per-unit area.) To better understand the parameters, let's build a model for the HSL system.

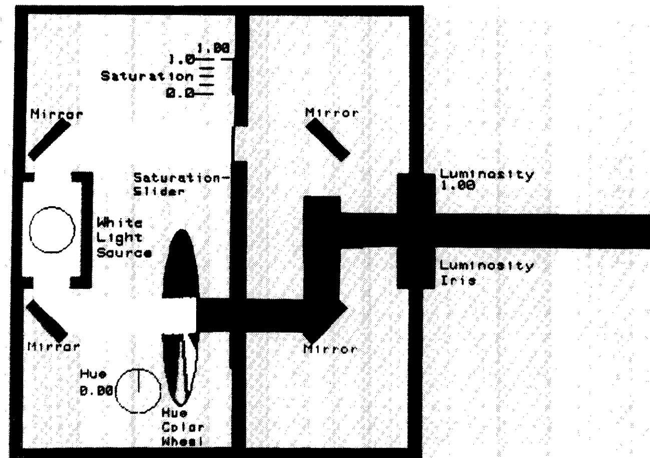
If we start with a white light source we should be able to get any color we want by filtering it. (A perfect white light source contains equal parts of all possible colors.)

The first step is to select the Hue to work with. This can be done with a color filter. In fact, if we take several color filters, and arrange them to form a disk, we could rotate the disk in front of the white light source and choose any of the colors on the filter wheel. Since the model we are working with is a model for understanding rather than one that we actually have to build, we can consider the wheel to consist of an arbitrarily large set of color filters, so that any rotational movement of the wheel will select a different color filter. Now we will provide a mechanism to drive the wheel which will position it angularly, based on a number we send to it, a number between 0 and 1 (inclusive). We will arrange the filters as a conventional color wheel (there are advantages to this, which are discussed under "Effective Use of Color," later in this chapter). Since it is a wheel, it must meet itself somewhere, and Red is as good a place as any, so two parametric values (0 and 1) describe red. Such a color wheel would look something like this:

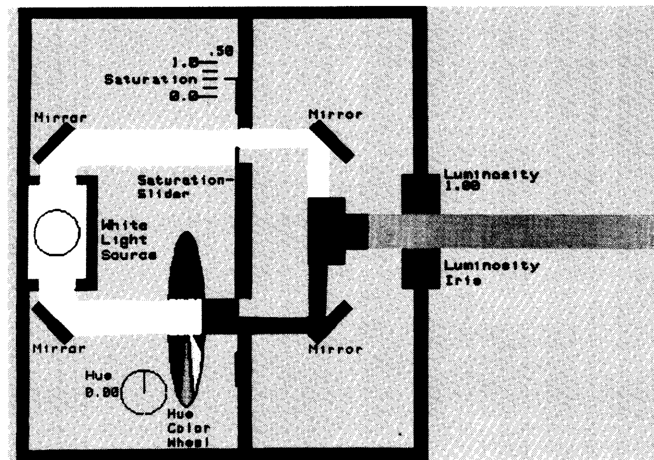


A Color Wheel for the HSL Model

This arrangement is fine for producing highly saturated colors (bright, pure, intense colors), but there are other types of colors, and we need to be able to produce them. For a start, we can mix some white light (remember our white light source?) with the filtered light, to desaturate the color. Combining the filtered and unfiltered lights directly would produce 50% saturation, and would double the luminosity of the resultant color. We want to have variable control of the saturation, and, to keep the model simple, it would be better if the result of the saturation control produced a unit luminosity. If, instead of mixing the two light beams directly, we mix the outputs of two simple optical gates that are linked with a mechanical slider to control the proportions of the colored and filtered light, we can control of the saturation while maintaining a constant luminosity (intensity-per-unit-area). Once again, we will provide a mechanism which takes a number between 0 (no color - pure white) and 1 (fully saturated color) and positions the slider appropriately. The two pictures below show the model we have described, with a fully saturated red in the first one, and a 50% saturated red in the second one.

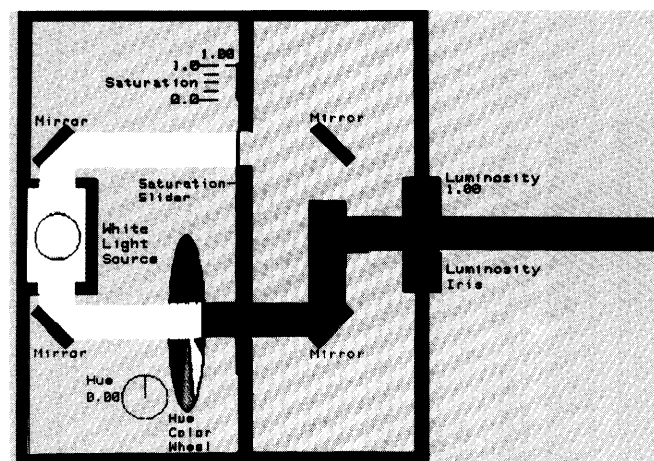


Fully Saturated Red

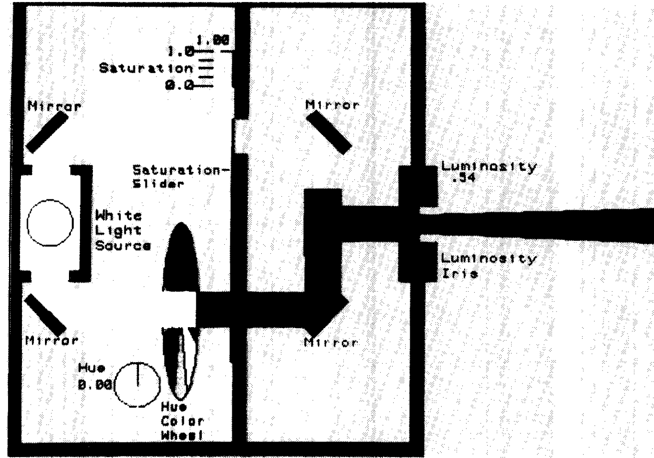


50% Saturated Red

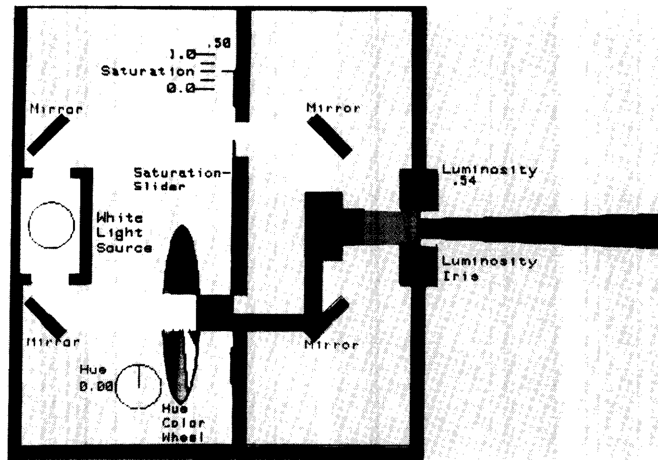
Finally, we may wish to change the luminosity, or brightness of a color (for example, brown is a dark red). This can be accomplished by putting an iris (like the one found on a 35 mm camera) after the mixer that combines the output from the saturation slider. The same 0 through 1 numerical control interface is used to control the iris, and thus the luminosity. The following three pictures show some combinations of the various controls:



Fully Saturated, Fully Luminous red.



Fully Saturated, 50% Luminous Red.



50% Saturated, 50% Luminous Red.

To recap, the Hue parameter rotates a color wheel to select a "pure" color to use. This color is then mixed with white light. The ratio of the pure colored light to the white light is controlled by the Saturation slider. Finally, the output passes through the luminosity iris (think of it as a hole you can adjust the size of) that controls the brightness of the output.

The HSL model is specified by a model selector of 2 in the SET_COLOR_MODEL statement:

```
SET_COLOR_MODEL (2);
```

A program called "COLOR" on the "DGLPRG:" disc uses the HSL model for interactive color selection. ("COLOR" only works correctly on a Model 236 Color Computer.) It produces two arrays for use with the SET_COLOR_TABLE statement, one for INTENSITY and one for COLOR. The program is over 300 lines long, almost all of which is simply a human interface to the following code in the update routine:

```
SET_COLOR_TABLE (TableEntry,
                 HueVal[TableEntry],
                 SatVal[TableEntry],
                 LumVal[TableEntry]);
```

Which Model?

Two models are provided by the DGL color system. If you are working with primaries only, or want gray scale output, the RGB model is great. If, on the other hand, you are trying to deal with pastels and shades, you are better off with a color model that is intuitive in nature, and that is where the HSL model shines.

It is possible to get the best of both worlds by using the HSL model for the human interaction, then reading the color table to get the RGB color values.

The "COLOR" program mentioned above does exactly that to calculate the correct cursor and text color to use when the user changes the background color. This is done by reading in the RGB color table values, calculating which corner of the color cube is farthest from the background color, setting the foreground pen and text displays to that color, and then writing the RGB values back into the color map. Even though the primary interaction is with the HSL model, the RGB model is used because it is more convenient to find distances between colors in it.

```
type
  Colors=           (Red,Yellow,Green,Cyan,Blue,Magenta,White,Black);
  Modes=            (Hue,Sat,Lum,Table,Copy1,Copy2);
  EntryRange=       -1..16;
  FunnyArray=       array [Colors] of char; {array for alpha color}
       •
       •
       •
const
  FunnyChar=        FunnyArray[chr(139),chr(137),   {\  Array for   }
                               chr(136),chr(140),   { \ holding the }
                               chr(142),chr(143),   { / alpha-color }
                               chr(141),chr(138)];  {/  controllers }
       •
       •
       •
```

```
var
  TableEntry:                 EntryRange;
  RedBack,GreenBack,BlueBack:  real;
  LabelColor:                 char;
  BackSum,OldBackSum:         0..7;
      •
      •
      •
    if TableEntry=0 then begin {Background color}
     set_color_model(1);                        {RGB}
     inq_color_table(0,RedBack,GreenBack,BlueBack);     {get RGB values}
     BackSum:=0;                                { \  Calculate the      }
     if RedBack<0.5 then BackSum:=4;            {  \ background color   }
     if GreenBack<0.5 then BackSum:=BackSum+2;  {  / in order to make   }
     if BlueBack<0.5 then BackSum:=BackSum+1;   { /  contrasting text.  }
     if OldBackSum<>BackSum then begin {Color change}
       case BackSum of
         0:  LabelColor:=FunnyChar[Black];      { \                     }
         1:  LabelColor:=FunnyChar[Blue];       {  \                    }
         2:  LabelColor:=FunnyChar[Green];      {   \  Translate the    }
         3:  LabelColor:=FunnyChar[Cyan];       {    \ RGB background   }
         4:  LabelColor:=FunnyChar[Red];        {    / sum to a         }
         5:  LabelColor:=FunnyChar[Magenta];    {   /  complementary    }
         6:  LabelColor:=FunnyChar[Yellow];     {  /   text color.      }
         7:  LabelColor:=FunnyChar[White];      { /                     }
       end; {case BacksSum of}
       MenuLine;                                {print the menu line}
       OldBackSum:=BackSum;                     {store for future comparisons}
       set_color_table(1,1-RedBack,             { \  Make pen one       }
                         1-GreenBack,           {  > complementary,     }
                         1-BlueBack);           { /  too.               }
     end; {if}
     set_color_model(2);                        {HSL}
    end; {if TableEntry=0}
```

One point brought out by the preceding example is that the models can be mixed freely. There is nothing to prevent using the RGB model to set a gray background color and a black pen, and then using the HSL model to produce the rest of the palette. Use whatever is easiest for what you want to do.

If you are interested in pursuing the color models, the RGB model is called a Color Cube and the HSL model is called the Color Cylinder. These models represent idealized color spaces and are discussed next.

Color Spaces

If you ask a broadcast engineer what the primary colors are, he will probably tell you "Red, green, and blue." If you ask a printer what the primary colors are, he will probably tell you "Cyan, magenta, and yellow." If you ask a physicist, he will probably smile and say "That's not the right question." Let's see if we can get enough information about color systems to ask the right question.

Primaries and Color Cubes

The reason for the confusion is that there are two sets of color primaries. Red, green and blue are additive primaries. Cyan, magenta, and yellow are subtractive primaries. Each of these sets of primaries can be used to construct what is referred to as a *color cube*. These are called the RGB color cube and the CMY color cube.

Each of the color cubes can be used to describe a *color space*. Color spaces are mathematical abstractions which are convenient for scientific descriptions of color. This is because the color spaces provide a coordinate system for describing colors. Once you have a coordinate system, you can talk about and manipulate colors mathematically.

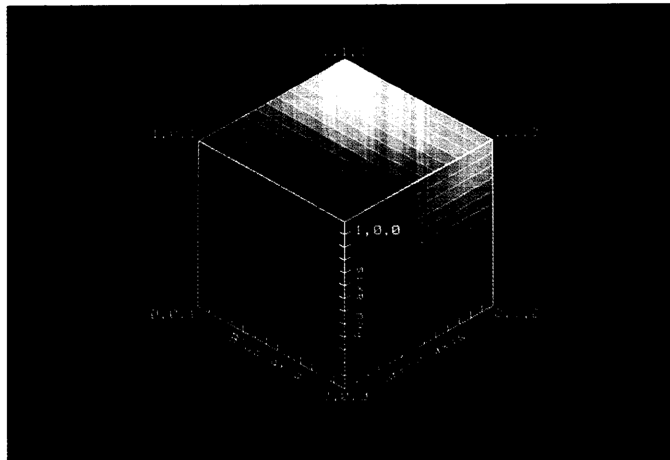
In addition to the color cubes, other color coordinate systems exist. While there are many, we will only look at HSL Color Space, because it is one of the available color models on the Model 236 Color Computer. First, the cubes.

The RGB Color Cube

The RGB color cube describes an *additive* color system. In an additive color system, color is generated by mixing various colored light sources. (Color mixing is discussed in "Effective Use of Color," below.)

The origin (0,0,0) of the RGB color cube is black. Increasing values of each of the additive primaries (Red, Green, and Blue) move towards white (the opposite corner of the cube.) The maximum for all three colors is white (1,1,1).

A diagonal of the cube connecting (0,0,0) and (1,1,1) represents gray shades, which are generated by incrementing all three color axes equally.



The RGB Color Space

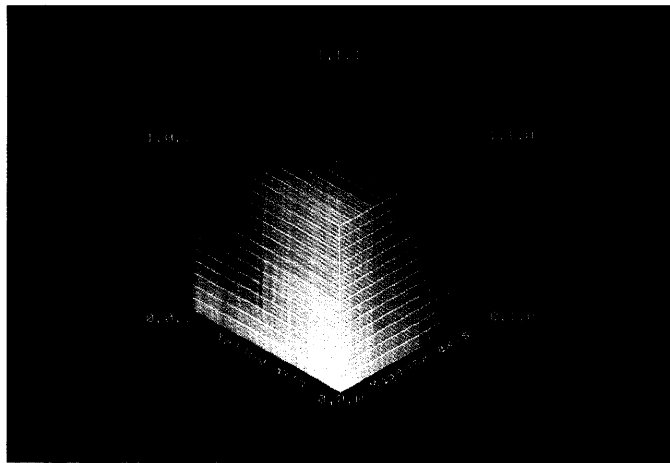
NOTE: This photo is a multiple exposure of Model 236 Color Computer CRT.

The CMY Color Cube

The CMY color cube represents a subtractive color system. In a subtractive color system, colors are created by subtracting colors out of a pure white (containing all colors equally) light source. This most often occurs when light is reflected off of surfaces containing, or coated with, pigments. This happens in printing and painting, among other operations.

The origin (0,0,0) for the CMY color cube is white. This represents all the colors in a perfect white light source being reflected by a white (reflecting all colors) surface. Increasing values of each of the subtractive primaries (Cyan, Magenta, and Yellow) move towards black (the opposite corner of the cube.) The maximum for all three colors is black (1,1,1).

A diagonal of the cube connecting (0,0,0) and (1,1,1) represents gray shades, which are generated by incrementing all three color axes equally. While the CMY color model is not supported by the DGL, it is important to understand when you get to color hard copy.



CMY Color Space

NOTE: This photo is a multiple exposure of Model 236 Color Computer CRT.

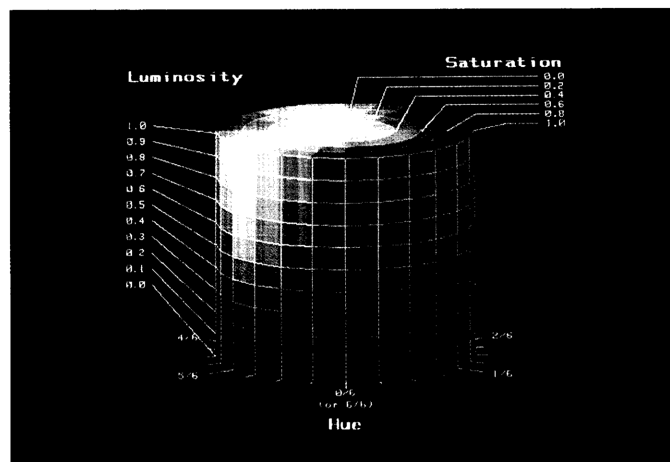
The HSL Color Cylinder

The color cubes are very useful for working with physical systems that are based on color primaries. They are not always intuitive, though.

The HSL color cylinder resides in a cylindrical coordinate system. A cylindrical coordinate system is one in which a polar coordinate system representing the X-Y plane is combined with a Z-axis from a rectangular coordinate system.

- The coordinates are normalized (range from 0 through 1).
- **Hue** (H) is the angular coordinate.
- **Saturation** (S) is the radial coordinate.
- **Luminosity** (L) is the altitude above the polar coordinate plane.

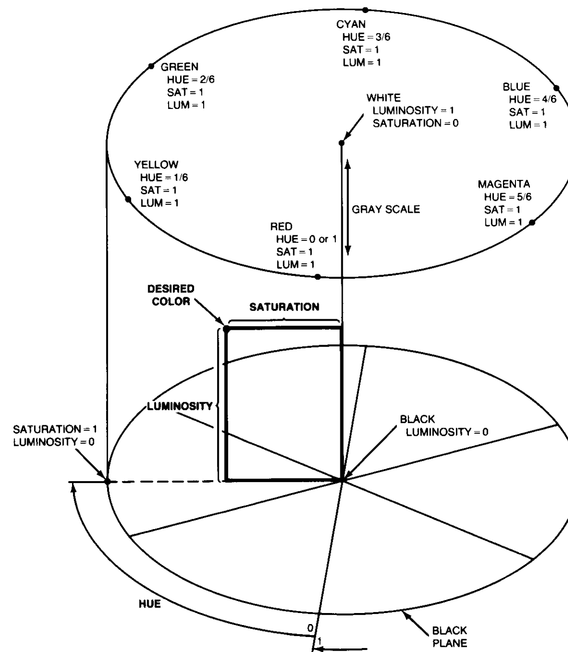
The cylinder rests on a black plane (L = 0) and extends upward, with increasing altitude (Luminosity) representing increasing brightness. Whenever luminosity is at 0, the values of saturation and hue do not matter.



HSL Color Cylinder

NOTE: This photo is a multiple exposure of Model 236 Color Computer CRT.

White is the center of the top of the cylinder (L = 1, S = 0).The center line of the cylinder (S = 0) is a line which connects the center of the black plane (L = 0, S = 0) with white (L = 1, S = 0) through a series of gray steps. (L from 0 to 1, S = 0). Whenever saturation is 0, the value of hue does not matter. The outer edge of the cylinder (S = 1) represents fully saturated color.



HSL Color Specification

Using the above drawing (HSL Color Specification,) hue is the angular coordinate, saturation is the radius, and luminosity is the altitude of the desired color.

Reality Intrudes

It would be fantastic if that were all you needed to understand in order to use the color capabilities in DGL. Unfortunately, "Reality rears its ugly head." HP does not make a piece of hardware capable of supporting the system described above. The Model 236 Color Computer is as close as a Series 200 computer comes to the color modeling system described above, and it only approximates it.

However, now that the idealized color system has been described, we can tackle some real hardware that DGL supports. We will start with the simplest display device (a plotter) and work up to the most complex (the internal color-mapped frame buffer in a Model 236 Color Computer). Along the way, some of the hardware dependencies that make each device unique will be brought out.

Plotters

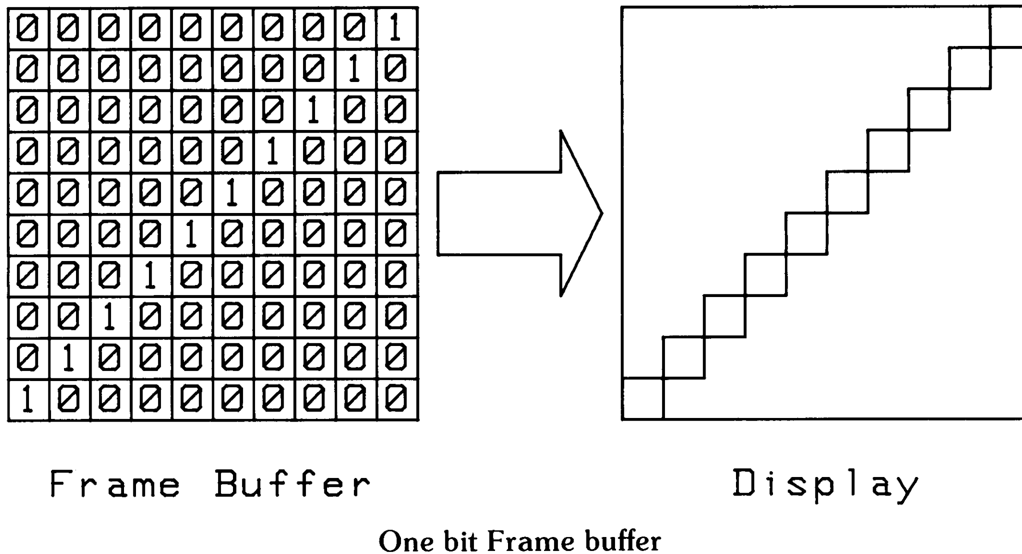
Numerous plotters are supported by DGL. All plotters support color as an attribute of graphics primitives to the extent it is possible with the number of pens available on the plotter. The SET_COLOR and SET_PGN_COLOR procedures select the pen used used to draw the primitives. Using a color selector of 0 will usually put whatever pen is in use away. *Calls to SET_COLOR_TABLE are ignored when a plotter is specified as the display device.* Plotters do not support the color modeling system.

Frame Buffers

The internal displays on Series 200 computers all have bit-mapped graphics, as does the HP 98627A. An area in memory called a *frame buffer* stores a binary description of each pixel location on the display.

Frame Buffer Depth

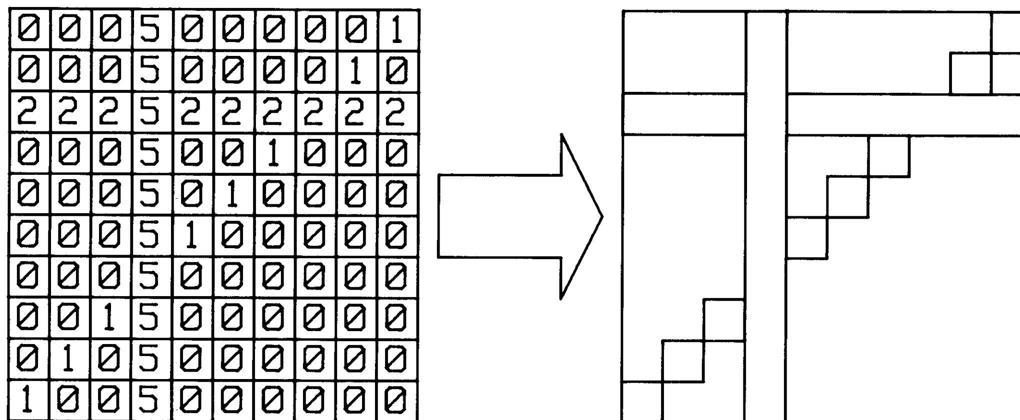
The number of bits available for describing each pixel is called the *depth* of the frame buffer. On all displays except the 98627A Color Output Interface and the Model 236 Color Computer, a single bit is used to describe each pixel location. A single bit allows each pixel to be on or off. This can be thought of as representing one of two colors (black or white, since the CRT is monochromatic). A one-bit frame buffer and the display it produces look something like this:



One bit Frame buffer

The 98627A has a three-bit frame buffer, allowing each pixel to be set to one of 8 colors (Black, Red, Green, Blue, Cyan, Magenta, Yellow, and White). Instead of storing ones and zeros (like a one-bit frame buffer), a number between 0 and 7 can be stored.

Three Bit
Frame Buffer System



Frame Buffer   Display

Three Bit Frame Buffer

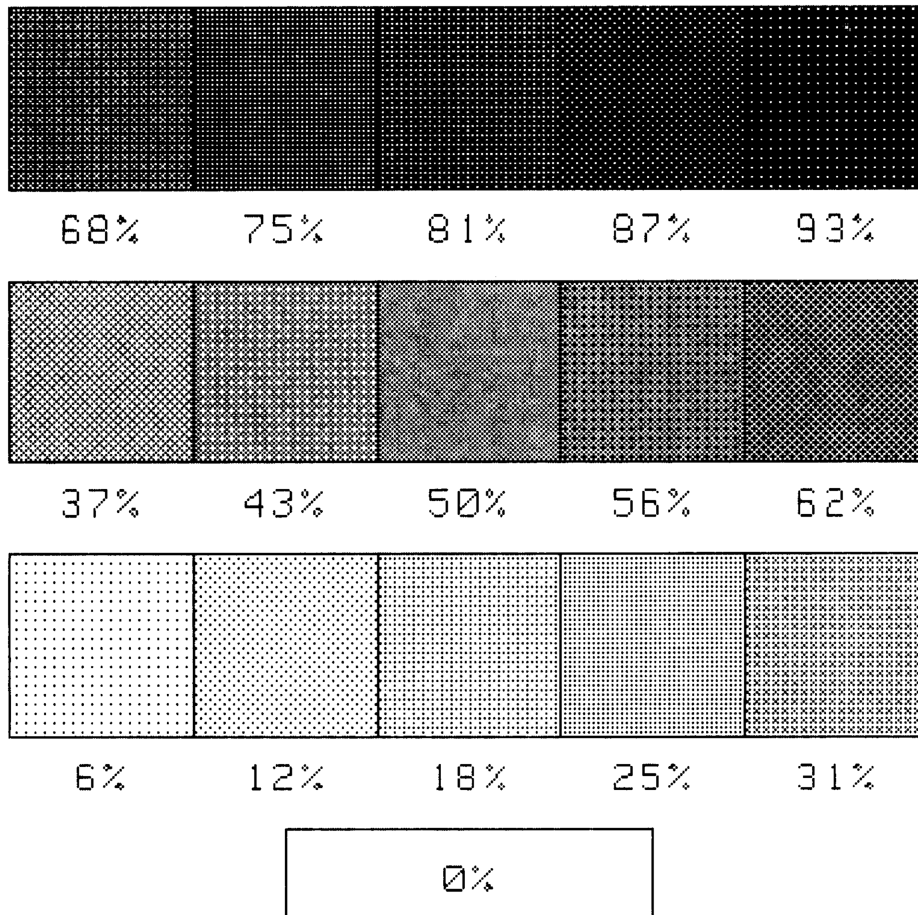
The Model 236 Color Computer has a four-bit frame buffer. A four-bit frame buffer allows each pixel location to contain a number between 0 and 15 (inclusive). Thus the Model 236 Color Computer can set a pixel to any of 16 different colors. The presence of a color map in the Model 236 Color Computer complicates this somewhat, by giving you control over the colors that each of the 16 possible entries in a frame buffer can actually represent (this is a palette of 16 colors out of a gamut of 4016 colors - see the color map description, below). For now, just think of the Model 236 Color Computer as having 16 colors that the user can define.

Faking More Colors From a Frame Buffer

If you have a one-bit frame buffer and need more colors, you can go up to a three- or four-bit frame buffer to solve the problem. If you already have a four-bit frame buffer and need more colors, the problem is more difficult to solve. The same solution that allows you to add more colors to the four-bit frame buffer also allows you to add more colors to a three-bit frame buffer, or even to a one-bit frame buffer. (O.K., it's actually shades of gray in a one-bit frame buffer.) The technique is called *dithering*, and is supported on all Series 200 frame buffers.

Dithering

In early color systems which did not provide control of the intensity of individual pixels, dithering became a very popular method of increasing the number of shades available to the machine. In dithering, *halftoning* is used to create the impression of a larger palette than the system hardware actually supports. This is done by creating patterns of dots of the available colors which the eye will (hopefully) combine into a perceived color different from the colors used to produce the patterns. The effectiveness of this technique depends on the distance from the display, the patterns involved, and the eye of the beholder. For example, if you want to produce a half intensity red, you can turn on half the dots in an area, and it will look half-bright. The 50% pattern fools the eye quite effectively.



Half Tone Color Selection

Thus, by reducing the effective resolution of the system, it is possible to provide a large number of shades of color. On Series 200 computers, this is done by imposing a grid of 4×4 squares on the CRT, that is, each of the squares is 4 pixels square. With a one bit frame buffer, it is possible to get 17 shades of gray in the square (all pixels off, and 1 thru 16 pixels on). On a 3 bit frame buffer (the HP 98627) there are three colors available, providing 4913 (17^3) shades. For a 4 bit frame buffer, there would be 83521 (17^4) shades, *if the colors represented by the frame buffer were fixed.* On the Model 236 Color Computer, however, it is possible to alter the colors represented by the frame buffer value, so the number of colors representable is variable - it could be larger or smaller than 83521 (which is more than the number of dithering squares available on the display, anyway) depending on the contents of the color map.

Creating A Dithered Color

The following discussion gets a little abstract, and it is not absolutely necessary to understand how dithering works to use it. It is interesting information, and can be useful knowledge if dithered areas don't do what you expect.

A color vector is a directed line connecting two points in RGB color space. The dithering process tries to match a *target vector* by constructing a *solution vector* from colors available to the frame buffer. The actual dithered color to be produced will be 16 times the target vector, since 16 points in the dither area will be combined to create it.

The color matching process requires sixteen steps. First, the target vector is compared to the vectors produced by all the colors in the color map. The one which is closest to the target vector is selected as the first component of the solution vector. The distance between the points in the RGB color space is used to determine how far apart the vectors are.

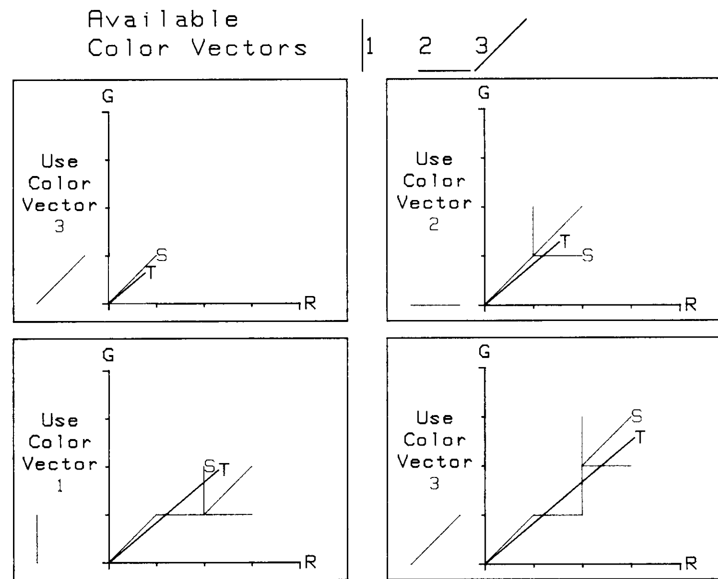
The following process is then repeated 15 times:

1. The target vector is added to itself to produce a new target vector.
2. A trial solution vector is created for each color in the color map by adding the vector for the color map entry to the previous solution vector. The trial solution vector that is closest to the target vector is selected as the new solution vector.

At this point, the target vector is 16 times the original target vector, and the solution vector consists of a summation of color vectors available to the frame buffer that produce, at each iteration, the vector closest to the target vector.

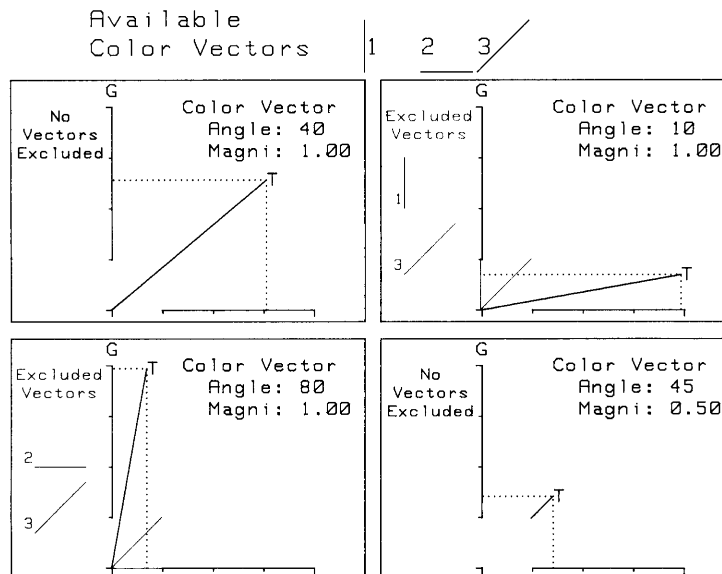
If all this has left your head spinning, let's take a look at a simplified system to see how the process works. Our simplified system will be a two color system (to keep it a two dimensional problem) with a 2×2 dither cell (which means we only have to look at four steps in the total process).

We will use green and red (let's not get "tangled up in blues") for the two axes. There will be three colors available to the frame buffer - a unit green, a unit red, and a combination of a unit red and a unit green. The vectors each of these colors produce is drawn at the top of the "Color Vector Matching" Diagram, shown below. At each step in the process, the target vector is labeled "T" and the solution vector is labeled "S." In addition, the test vectors that are not used are shown, with no labels on the endpoints.



Color Vector Matching

In actuality, the entire set of colors available to the frame buffer is not necessarily used for creating a color. Before the color matching process is started, the colors available to the frame buffer are sorted into two groups; those within the target cube, and those outside the target cube. The target cube is the cube formed by using the origin of the RGB color space and a point representing 16 times the target vector as diagonal corners to form a cube. Going back to our two dimensional model, we will construct a target square for the system. For a vector near one of the axes, the unit vector on the other axes will be excluded from the solution set, since it lies outside the target square.



Two Dimensional Target Square

Once the colors have been selected for the solution vector, the colors are sorted by luminosity and filled into the following precedence matrix (the most luminous color is filled into the lowest numbered pixel):

1	13	4	16
9	5	12	8
3	15	2	14
11	7	10	6

The dither precedence matrix is actually tied to pixel locations on the CRT. The matrix is repeated 128 times across the CRT and 97.5 times from the top to the bottom of the CRT (for a 512 by 390 display - just divide the number of pixels on each axis by 4 to get the number for other display sizes). Areas to be filled are mapped against the fixed dithering pattern. All dither cells completely within an outline to be filled are turned on according to the precedence pattern. Cells which are only partially within the border are only partially enabled. If the area fill pattern calls for a pixel outside the boundary to be set, it will not be.

There are problems with dithering:

- The dithered colors are not necessarily accurate representations of the color specified. Looking at the "Color Vector Matching" Diagram shown above, the solution vector does not actually match the target vector, it just comes near it. This is highly dependent on the colors available to the frame buffer. A 4-by-4 dither cell with one full intensity green pixel does not look the same as the same cell filled with 1/15 green.
- The dithered color selection tends to produce textures. In some cases, the textures overwhelm the shade produced.
- The dithered colors are not stable if the color map is altered on a Model 236 Color Computer. (This is discussed in more detail under "Color Maps," below.)
- The dithering operation produces anomalies when the area to be filled is thin. If it is less than four pixels wide or high, it cannot contain the entire dither cell and the results can be surprising for colors which turn on small portions of the cell.

If You Need More Colors

If you have an application that requires more colors than are available to your frame buffer, the first thing to do is see if you can redefine it to use the colors available to the frame buffer. In many cases this is possible, and the higher quality of the frame buffer palette is worth a little checking to see if you can use it.

If you have to use dithering, here are some hints for getting the best results:

- Check the colors to see if you are going to get objectionable texturing. Sometimes relatively minor shifts in color definition can produce significant differences in the patterns used in dithering.
- Remember - *you can't draw lines with dithered colors.* The DGL will automatically use the closest available color from the frame buffer.
- If you are on a Model 236 Color Computer, make sure your color palette is correctly set up for dithering.

On all frame buffers other than the Model 236 Color Computer, all the color table entries are potential dithered colors. On a Model 236 Color Computer, however, only the upper half of the color table (16 thru 31) are dithered colors. The lower half of the color table maps directly to the hardware *color map.* The color map is one of the most powerful graphic tools yet invented. It is described below, under "The Model 236 Color Computer Color System."

Frame Buffer Contents

Now that you understand frame buffers and dithering, it's possible to describe what is actually found in a frame buffer. At any given time, the values written to the frame buffer fall into four categories:

- *Background Value* - Whenever CLEAR_DISPLAY is executed, all the pixel locations in the frame buffer are set to the current background color. The background color is described by entry 0 in the color table.
- *Line Value* - The SET_COLOR statement is used to determine the value written to the frame buffer for all lines drawn. This includes all lines (including characters created by GTEXT) and outlines (for polygons with the edge parameter true in the polygon style table).
- *Polygon Interiors* - The SET_PGN_COLOR statement is used to specify the value written to the frame buffer for filling areas (for polygons with the fill attribute true in the polygon style table).
- *Dithered Colors* - when an application uses more colors than the frame buffer can support directly (see "Frame Buffer Depth," below), dithering is used to create as close an approximation of the color as can be done by mixing colors available to the frame buffer. Dithered colors can only be used for the background and for polygon interiors, not for lines.

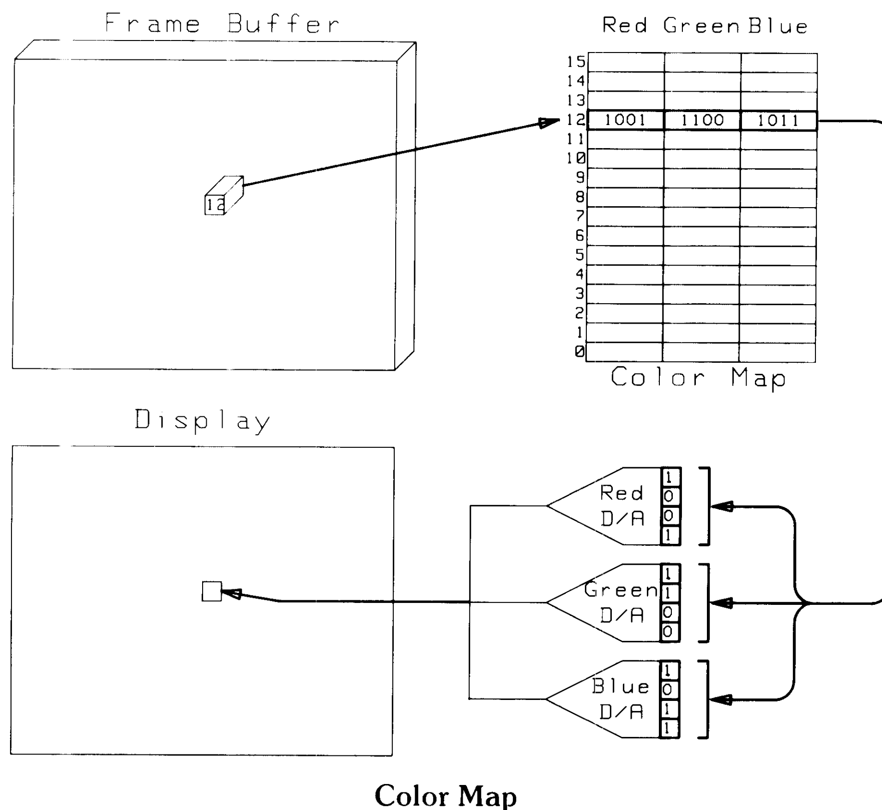
The Model 236 Color Computer Color System

The biggest benefit of the Model 236 Color Computer is that it makes experimenting with color so easy. With a bit-mapped frame buffer and a color map, it is easy to test out ideas before you use them. It is also possible to use the color map for simple animation effects and some just plain impressive images.

It is possible to use the Model 236 Color Computer with the default color map. The color used will depend directly on the value in the frame buffer. This is fine if the work you are doing can be accomplished using the 16 colors supplied as the system defaults. This is often not the case, and this overlooks one of the most powerful features of the Model 236 Color Computer - the color map.

The Color Map

The color-mapped system uses the value in the frame buffer as an index into a color map. The color map contains a much larger description of the color to be used (12 bits in the Model 236 Color Computer) and, just as importantly, the color description used is *indirect*. Thus, the value in the frame buffer does not say "use color 12", but rather "use the color described by register number 12".



Color Map

The CRT refresh circuitry reads the value from the pixel location in the frame buffer, uses it to look up the color value in the color map, and displays that color at that pixel location on the CRT. Thus, it is possible to draw a picture with a given set of colors in the color map (a set of colors is called a *palette*) and then change palettes and produce a new picture by redefining the colors, rather than having to redraw the picture. (The binary numbers in the color map are created by the system. The user deals with normalized values, as described under "Color Specification.")

True User Definable Color

The colors available are true user definable colors. The color can be changed on a pixel-by-pixel basis, so there are no restrictions on how the colors can be used (as there are with dithered shades, which can only be used for filling polygons). There are also no problems with texturing, as the color is not produced by mixing dot patterns.

Retroactive Color Changes

Another advantage of the color map colors comes from the indirect nature of the color map. Since the frame buffer contents only point to locations in the color map, it is possible to change the contents of the color map after an image has been created in the frame buffer, allowing "fine tuning" of the image after it has been created.

If You Need More Than 16 Colors

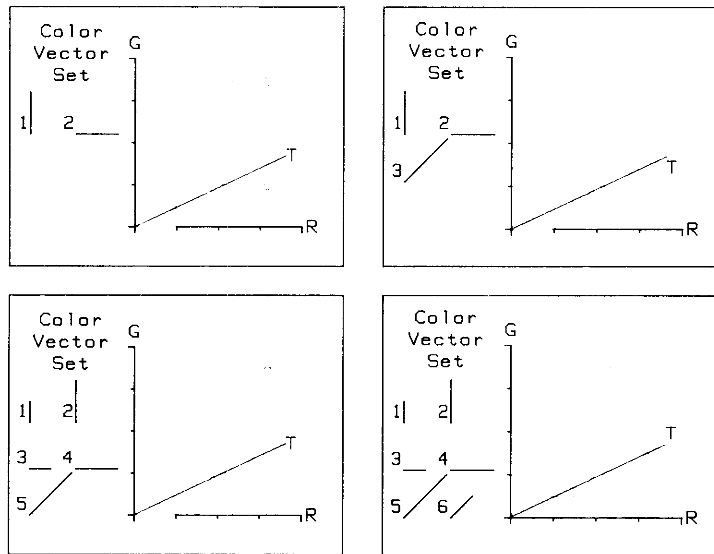
If you have an application that requires more than 16 colors, the first thing to do is see if you can redefine it to use 16 colors. In many cases this is possible, and the higher quality of the color mapped palette is worth a little checking to see if you can use it.

The Model 236 Color Computer provides dithering for applications that require more shades than the 16 colors that are available at any single time with the color map. The upper half of the color table (entries 16 thru 31) provide access to dithered colors, although they will fill with a single pen if the color requested exists in the current color map.

If you absolutely have to get at a larger palette, then load a palette optimized for dithering (optimizing for dithering is described below) and **stick with dithering**. Don't try to mix color map redefinition and dithering - it will probably cause you a lot of grief. Especially, do not try to do interactive redefinition of the color map in a system that is also using dithering.

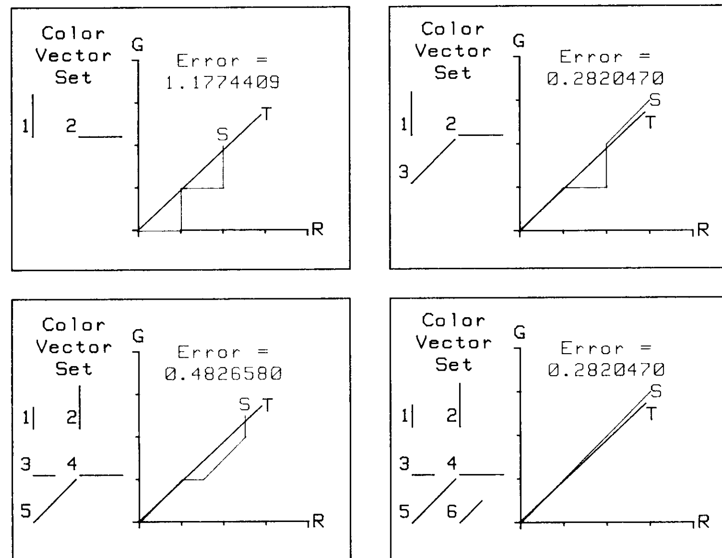
Optimizing For Dithering

The actual color palette you require determines the optimum color map values. Below are some plots of color matching on the simplified color system introduced under the discussion of dithering. Each plot is trying to match the same target vector, but using a different palette. The effect of various color maps on the distance between the target and solution vectors is striking.



Color Map Effect on Color Vector Matching

It's obvious from the drawing above that the larger the color map, the closer the match to the target color, right? Well, it's obvious from that drawing, but let's take a look at a slightly different color to match, and see what happens.



Color Map Effect on Color Vector Matching - Part 2

The point is, that the quality of color matching depends on both the contents of the color map **and** the color to be matched.

Resolution and Color Models

The resolution available with the two color models depends on the hardware being used to generate the color. Resolution on devices that use dithering is complicated by the variation in quality of the colors produced by dithering. Resolution of the color map is easier to deal with, so let's see what's available.

RGB Resolution

The resolution of the RGB model is limited by the 4-bit digital to analog converters in the Model 236 Color Computer graphics hardware. The 4-bit converter allows 16 states to exist for each of the CRT electron guns, so the resolution of each of the RGB parameters is 1/15, from 0 thru 1. In fact, since the SET_COLOR_TABLE statement accepts real arguments, you can express the values as fractions, and let the computer convert to decimals. The following call would set the background to about 50% gray.

```
SET_COLOR_TABLE (0, 7/15, 7/15, 7/15)
```

HSL Resolution

The resolution of the HSL model is not specified anywhere. This is because the resolution for the various parameters is not a fixed value. The resolution for any parameter of the HSL system is dependent on all three of the parameters. The resolution is not only changed by the other two parameters, but also by the magnitude of the parameter you are varying.

Writing Modes and Color

Since HP Series 200 frame buffer devices are bit mapped, it makes sense that various logical combinations of the bits in the frame buffer with the bits being added by a drawing operation should be possible. Since this is a highly device dependent operation, the various drawing modes are specified with calls to OUTPUT_ESC. Four drawing modes are available:

- Dominant
- Non-Dominant
- Erase
- Complement

Three of these drawing modes have already been introduced (all but non-dominant) in Chapter 2. The meaning of the modes is slightly different for a color system than for monochromatic systems. The actual meaning of each of the modes is discussed below, but first, a slightly modified version of the DrawingMode procedure presented in Chapter 2 is listed below. The non-dominant drawing mode has been added to it.

```
$page$ {*********************************************************************}
procedure DrawingMode(Mode: DrawingModeType);
{---------------------------------------------------------------------------}
{    This procedure selects drawing modes for a color-mapped CRT.            }
{---------------------------------------------------------------------------}
const
  SetDrawingMode=       1052;          {mnemonic better than magic number}
var
  DrawMode:     array [1..1] of integer;        { \   This is all stuff that  }
  Rarray:       array [1..1] of real;           {  >  is needed by the        }
  Error:        integer;                        { /   "output_esc" procedure. }
begin                                  {procedure "DrawingMode"}
case Mode of                   { \                                 }
  Dominant:     DrawMode[1]:=0;    {  \   Convert DrawingMode enumerated  }
  NonDominant:  DrawMode[1]:=1;    {   \  type into the appropriate       }
  Erase:        DrawMode[1]:=2;    {   /  value for OUTPUT_ESC procedure. }
  Complement:   DrawMode[1]:=3;    {  /                                }
end; {case}                    { /                                 }
output_esc(SetDrawingMode,1,0,DrawMode,Rarray,Error);          {set it}
if Error<>0 then writeln('Error ',Error:0,' in procedure "DrawingMode".');
end;                                   {procedure "DrawingMode"}
```

The global TYPE declaration for `DrawingModeType` must also be changed:

```
DrawingModeType = (Dominant, Erase, Complement, NonDominant);
```

"Draw" has been changed to "Dominant" to make it consistent with references to the non-dominant mode'

Dominant Writing

Dominant writing is the easiest to understand. When DGL has a new value to write to a location in the frame buffer, whatever is already in the frame buffer is overwritten, and thus lost. The system wakes up in the dominant mode.

Non-Dominant Writing

All the techniques described up until now have dealt with dominant writing to the frame buffer. In the dominant writing mode, the color selector is written directly to the color map, and overwrites whatever is currently in the frame buffer. In non-dominant writing, a bit-by-bit logical-or is performed on the contents of the frame buffer and the table entry selector value being written to the frame buffer. Thus, if color selector 1 is written to a buffer location that has already been written to with color selector 6, the buffer location will contain 7, but writing color selector 2 to a buffer location that has already been written to with color selector 6 will not change the contents.

Erasing

Erasing is a fairly simple concept in frame buffers that are a single bit deep. You just restore the background by setting the portion of the frame buffer you wish to erase to 0. The concept is a little more complex in frame buffers with more depth (such as the Model 236 Color Computer.) At the simplest level, you can simply set the contents of the frame buffer to the background color, using a call to CLEAR_DISPLAY.

It is also valuable to erase a single line. This can be done by setting the drawing mode to erase, and then re-drawing the line you wish to erase. In the erase mode, the erasure is done non-dominantly. This means that the bits which have a 1 value in the current color table entry selector are cleared to 0 in the frame buffer entries that are modified by the line drawn in the erase mode. For example, if a table entry selector of 5 is used to erase the a line written with a table entry of 5, the frame buffer entries are returned to 0. If, however, the same line crosses a frame buffer entry of 7, the result is a value of 2 (only the bits set in 5 are cleared to 0 by the operation.

The only method that *insures* erasing a line is to select the dominant writing mode and draw over the line in the background color. This is done with a table entry selector of 0 (for the frame buffer background) or a table entry selector equal to a "local background," if the line you are trying to erase is drawn across an area filled with a color other than the background color.

Complementary Writing

The complementary drawing mode is provided for operations (such as making your own cursor) that need to put an image on the screen that is always visible, but that can also be taken off the screen without damaging the background. On the Model 236 Color Computer, the concept of a complementary pen is extended to deal with the 4-bit values in the color map. In the non-dominant mode, the bit pattern represented by the table entry selector will be exclusively-ORed with the contents of the frame buffer.

The complement occurs only for the bits which are one in the table entry selector. Thus an entry selector of -6 would complement bits 1 and 2 of the frame buffer. If a 1 exists in a frame buffer location and a line is drawn over it with entry selector 6, a 7 will now be in the location. Writing over the pixel with the same table entry selector will return it to a 1.

Making Sure Echoes Are Visible

It is important to understand that the complementing is of the frame buffer, not the color map. You are responsible for making sure that the complimented frame buffer values are visible against one another. Be careful of placing the same color in two locations on the color map that are complements of one another. If you pick one of them as an echo color and then try to use the echo over an area filled with the other value, you will not be able to see it.

Drawing Modes and the Frame Buffer

Let's try to make things a little more concrete. We will look at a 9 × 9 section of a frame buffer, and draw some lines in the various modes, with different table entry selectors. Starting in the dominant mode, if we draw a cross with a table entry selector of 5, and then put a square with a table entry selector of 7 down on top of it, the following frame buffer results:

0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0
0	0	7	7	7	7	7	0	0
0	0	7	0	5	0	7	0	0
5	5	7	5	5	5	7	5	5
0	0	7	0	5	0	7	0	0
0	0	7	7	7	7	7	0	0
0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0

Dominant Writing to the Frame Buffer

If we then set the erase drawing mode and use a table entry selector of 5 to try to erase the horizontal element of the cross, we end up with two pixels of the horizontal element *not* erased, since the square had changed those locations to a 7, and the erase mode only erases the bits that are set to one in the table entry selector. The frame buffer ends up looking like this:

0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0
0	0	7	7	7	7	7	0	0
0	0	7	0	5	0	7	0	0
0	0	2	0	0	0	2	0	0
0	0	7	0	5	0	7	0	0
0	0	7	7	7	7	7	0	0
0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0

Erase Writing to the Frame Buffer

If you want to set a line to the background color, do it in dominant mode, with a table entry selector (in SET_COLOR) of 0.

Now, clear the frame buffer, and let's take a look at non-dominant writing. Non-dominant writing or's the contents of the frame buffer with the table entry selector. Let's put the cross and the square in the frame buffer, again, but this time we will use non-dominant mode, and a pen selector of 2 for the square. The cross will be written first, and then the square. The following frame buffer results:

0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0
0	0	2	2	7	2	2	0	0
0	0	2	0	5	0	2	0	0
5	5	7	5	5	5	7	5	5
0	0	2	0	5	0	2	0	0
0	0	2	2	7	2	2	0	0
0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0

Non-Dominant Writing to the Frame Buffer

Now let's try some complementary writing to the frame buffer we got from the non-dominant writing example, above. We will draw over the horizontal line, using a color table entry selector of 7. The first time, we get the following:

0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0
0	0	2	2	7	2	2	0	0
0	0	2	0	5	0	2	0	0
2	2	0	2	2	2	0	2	2
0	0	2	0	5	0	2	0	0
0	0	2	2	7	2	2	0	0
0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0

Complimentary Writing to the Frame Buffer

If we do it again, we end up with this:

0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0
0	0	2	2	7	2	2	0	0
0	0	2	0	5	0	2	0	0
5	5	7	5	5	5	7	5	5
0	0	2	0	5	0	2	0	0
0	0	2	2	7	2	2	0	0
0	0	0	0	5	0	0	0	0
0	0	0	0	5	0	0	0	0

More Complimentary Writing to the Frame Buffer

Notice that the first line is highly visible (assuming the color map contents do not produce the same colors for several entries in the frame buffer), but that the frame buffer is restored to it's original values after the second operation. *This will not be true if a line is drawn through the area before the complimentary line is "undrawn."* Always undraw complimentary line before you try to add things to the frame buffer.

Special Considerations

The drawing modes mentioned above are only available on frame buffers. There are some special interactions with various primitives in the graphics system that need to be taken into consideration.

Text

When text is written in the complimentary mode, gaps will be produced in the characters, wherever the character intersects itself. This includes crossovers *and* endpoints of lines that overlap. Readability of the text can be heavily impacted by this. Make sure you want the result before putting GTEXT calls while the drawing mode is complimentary.

Polygons

Device independent polygons (INT_POLYGON and POLYGON) are written to the frame buffer using the current drawing mode. Device dependent polygons (INT_POLYGON_DD and POLYGON_DEV_DEP) **ignore** the drawing mode. Make sure you use the correct one if you want the drawing mode to work.

Effective Use of Color

At the beginning of this chapter, it was pointed out that color is a very powerful tool, and that it is also easy to misuse. While it is beyond the scope of this book to provide an exhaustive guide to color use, a few comments can be made on using color effectively.

This section will deal with seeing color first, to lay the groundwork. This is followed by a discussion on designing effective display images, since effective color use is almost impossible if the image is fundamentally unsound.

After laying the groundwork, effective color use is discussed, from both the objective and subjective standpoints.

Seeing Color

The human eye responds to wavelengths of electromagnetic radiation from about 400 nm to about 700 nm (4000 to 7000 angstrom). We call this visible light. Visible light ranges from violet (400 nm) to red (700 nm). If all the frequencies of visible light are approximately equally mixed, the result is called white light.

The eye's ability to discriminate color is reduced as the light level is reduced. This means that the variety of colors perceivable at low light levels is smaller than the variety at higher light levels.

The eye is most sensitive to colors in the middle of the visible spectrum, a yellow-green color. The eye is least sensitive to the shorter wavelengths, which are at the blue end of the spectrum. Sensitivity to red is between that of yellow-green and blue. Two things seem to be associated with the sensitivity of the eye to various colors:

- The eye can distinguish the widest range of colors in the yellow-green region, and the smallest variety of colors in the blue region.
- The eye is most sensitive to detail in the yellow-green region.

Why and how any of the above works is explained by color theorists. There are a large number of theories of color, and all of them work for explaining the specific phenomena the researchers were studying when they developed the theory, but none of them seem to be able to explain it all. The list of references at the end of this chapter include several on how vision works.

It's All Subjective, Anyway

One of the reasons that there are so many color theories is that no two people seem to perceive color the same way. In fact, the same person will many times perceive color differently at different times. In addition to the physiological and psychological variables in color perception, many environmental factors are important. Ambient lighting and surrounding color affect the perceived color tremendously.

At this point, it will be well worth your time to compile and execute the program "COLOR", from the "DGLPRG:" disk. Try setting the background color to each of the pen colors, and see how different the foreground colors look against the different colors. In some cases, the lines even look slightly different from the filled rectangles of the same color. It turns out that the size of a color sample affects how it is interpreted, too.

The subjectivity of color, and the importance of background color in interpreting colors is the whole reason the program "COLOR" is provided. The color selector program lets you select the background color and provides both filled areas and lines due to the effect of the background color and the size of the color sample on the perception of color. The only way to insure a set of colors works well together is to try it and see.

Mixing Colors

If two distinct audio tones are played simultaneously, you will hear both of them. If the same area is illuminated by two or more different colors of light, you will not perceive the original colors of light, but rather a single color, and it will be not be one of the original colors. What you will perceive is called the *dominant wavelength*.

The CRT uses three different colored phosphors (Red, Green, and Blue) and mixes various intensities of the resulting lights to produce one of 4096 colors at any point on the CRT. What you actually see is the resulting dominant wavelength. This is an additive color system.

Mixing with pigments is a little different. Pigments in inks and paints absorb light. The idea with pigments is to subtract all but the color you want out of a white light source. This is a subtractive color system, and the primary colors are cyan, magenta, and yellow.

The different mechanisms for mixing additive and subtractive colors make it difficult to reproduce images created with additive colors (like a CRT) in a subtractive medium (like a plotted or printed page.) Photographing the CRT is the best method currently available for color hard copy. This problem is discussed in more depth at the end of this chapter under "Color Hard Copy."

Designing Displays

While the design of displays is not really a color topic, a few words about it are in order before we get into the effective use of color. If the design of an image is fundamentally unsound, all the good color usage in the world is not going to help it.

Whenever you put an image on a CRT, you have created a graphic design. The design will either be a good one or a bad one, and if you know this, you have automatically increased your chances of creating a good design. If you are going to be creating a lot of displays, either in a lot of programs or in a single large program, you need a graphic designer. Many people have a natural talent for graphics - an ability to look at an image and tell whether it is graphically sound or not. If you don't have that talent (or feel you could use some help) there are two courses of action that might be productive for you; you can hire a graphic designer or become one. Renting one is expensive and becoming one is time-consuming, but if you are trying to communicate with users, *you have to understand graphic design.* While getting a degree in graphic arts may be impractical for some programmers, a course or two in the field will prove very useful if you do very much programming.

While this book can't turn you into a graphic designer, a few simple hints may help you on your next program.

The most important thing in communicating with people is to keep it simple. Don't try to communicate the total sum of human knowledge in a single image. It is much more effective to have several screens of information that a user can call up as required, than a single screen so complicated that the user can't find what he wants on it.

Try to redundantly encode everything, in case one of the encoding methods fails. For example, if you color code information, use positional coding (the location of the information tells something about the nature of the information) too. Remember, the person reading the screen is probably *not* the person who wrote the program, and even if you are writing the program for yourself, you may forget how it works by the next time you try to use it.

Another important thing to remember is to be consistent. Always try to place the same type of information in the same area of the CRT and use the same encoding methods for similar messages. Don't using flashing to encode important information on one display and then use inverse video for the same thing seven displays into the program.

Objective Color Use

In spite of the subjectivity of color, there are some fairly objective things that you should know about color. Some of the things that can be done with color don't depend heavily on subjective interpretation.

Color Blindness

A fact of life that it is dangerous to ignore is that some people are color-blind. The most common form of color blindness is red-green color blindness (the inability to distinguish red and green). Avoid encoding information using red-green discrimination, or these people will have difficulty using the system.

Subjective Color Use

Choosing appropriate colors for a program to use can be tricky, and constitutes a significant part of the job of a good graphic designer. In the final analysis, it is a largely a matter of trying combinations until you come up with a set of colors that look good together. If your application is complex, it will be well worth your while to consult with a graphic designer about the color scheme and layout of information displays for your program. There are, however, a few fairly fundamental things to remember in designing your programs.

Choosing Colors

First, and probably most important, is to use color sparingly. Color always has a communication value and using it when it carries no specific information adds noise to the communication.

Use some method for selecting the colors - one of the best is a color wheel, similar to the one shown in the section on the HSL color model.

- Try varying the luminosity or saturation of a color and its complement (opposite it on the color wheel).
- Try color triplets (three equally-spaced colors) and other small sets of colors equally-spaced around the color wheel.
- Pastels (less than fully-saturated colors) tend not to clash.

Give careful attention to your background color. Remember that a filled area can become the background color for a portion of the image on the CRT.

- If you are using a small number of colors, use the complement of one of them for the background.
- If you are using a large number of colors, use a gray background.

If two colors are not harmonious, a thin black border between them can help.

Use subtle changes (such as varying the saturation or luminosity of a hue) for differentiating subtly different messages and major changes (such as large changes in the hue of saturated colors) to convey major differences.

Most of all, think and experiment. The final criteria is "Does this display communicate the message?".

Psychological Color Temperature

Temperatures ranging from cool to hot are associated with colors ranging from blue to red (ice blue - fire red). This is actually the opposite of physical reality, where the higher the temperature, the shorter the wavelength (blue is a black body radiation of about 7600° K while red is about 3200° K) but this is what people *perceive* as the relation between temperature and color. This is probably because people very seldom deal with the high temperatures and associate the blues with non-temperature related natural phenomena (oceans and ice). If you are trying to portray temperature, electrical field strength, stress, or some other continuous physical system, using the psychological color temperature can serve as a useful starting point for color coding the values.

Cultural Conventions

When trying to use color for communicating, cultural conventions are useful. Red is widely associated with danger in most western cultures, giving extra emphasis to a flashing red indicator. By the same token, a flashing green indicator would be less effective for communicating an out of range value in a system. In any specific application, it is important to understand the color associations that are common for the group using the application.

Reproducing Color Graphics

Color Gamuts

The range of colors a physical system can represent is called its *color gamut.* Color gamuts are important when you want to convert between different physical systems, because the source system may be able to produce colors the destination system cannot reproduce. An exhaustive treatment of color gamuts is beyond the scope of this book. However, here are some rules of thumb:

- The color gamuts for CRTs and photographic film are not the same, but are fairly close. If you are lucky, you can photograph the CRT and catch it on film. It may take more than one exposure, so be careful and bracket everything with several exposures.
- The color gamut for printing is significantly smaller than that of either photographic film or of a CRT. The fact that you have a picture of a CRT does not mean you can hand it to a printer and get a faithful reproduction of it.
- The color gamut of a plotter is much smaller than that of a CRT. You have to create images with the limitations of a plotter in mind if you intend to reproduce them on a plotter (see "Plotting and the CRT," below.)

The different color gamuts available are not a problem unless you forget the differences and try to act like all physical systems have the same gamut. Think ahead if you have to reproduce images - it will save a lot a trouble.

Color Hard Copy

Color hard copy represents a translation between color systems, and many of the problems in color hard copy arise from the fact that the color gamuts available to the CRT and the hard copy device are different.

There are two basic ways to get a color hard copy of what is displayed on the Model 236 Color Computer:

- Take a picture of the CRT.
- Re-run the program that generated the image with an external plotter selected as the display device.

The first method is the easiest and can capture (usually) whatever is on the CRT, regardless of what colors are used (see "Color Gamuts," above.) The second requires setting up the Model 236 Color Computer color map to match the pens in a plotter, and is not as likely to capture what you see on the screen. Both methods are discussed below.

Photographing the CRT

Photography is an art, not a science. Capturing images off a CRT is relatively straightforward, but sometimes unpredictable due to the different color gamuts available for film and the CRT. The following guidelines will provide a starting point. If your images are not "typical" (whatever that means) you may have to go back and re-photograph some of them. Many of the CRT images in this book were captured using these guidelines.

- Use ISO 64 Color film. (Most of the color photos in this book were taken on Kodak Ektachrome 64[1].)
- Set up your equipment in a room that can be darkened. It will have to be darkened for the one-second exposure.
- Use a telephoto lens (the longer the better). This minimizes the effects of the curvature of the CRT.
- Use a tripod.
- Darken the room and take a one-second exposure.
- Bracket the aperture around f5.6. (One stop above and below.)

Plotting and the CRT

There are two basic reasons the CRT is hard to capture on a plotter.

- The CRT is an additive color device and a plotter is a subtractive color device.
- The color gamut of the CRT is much larger than that of the plotter.

The conversion from additive to subtractive colors is not a huge problem if the plot is a simple line drawing with few intersections and area fills. If the plot is complex, especially with lots of intersections and overlapping filled areas, the plot is much less likely to capture the display image accurately.

A possible technique described below *purposely* limits the color gamut of the CRT to give the plotter some chance of capturing it.

To set up the color map and plotter to match one another:

- Set your background to white.
- Set up pens matching the color map colors in slots 1 through 8 in the same order they are presented in the default color map listed under "Default Colors."
- Use color table entry selectors from 8 through 15 in your drawings.
- Run the program with the color mapped CRT as the display device, modifying it as necessary to produce the image you want on the CRT.
- Re-run the program with the plotter as the display device. You will need to subtract 8 from the color table entry selectors to properly select the pens on the plotter.

While it is possible to get some idea of the plot that will be produced on the plotter, don't be surprised if they don't look exactly the same. Colors on a CRT are different in source and form from colors on a plotter, as described under "Seeing Color," above.

Color References

The following references deal with color and vision. Texts that serve as useful introductions to the topic are starred.

* Cornsweet, T., *Visual Perception.* New York: Academic Press, 1970

Farrell, R. J. and Booth, J. M., *Design Handbook for Imagery Interpretation Equipment* (AD/A-025453) Seattle: Boeing Aerospace Co., 1975

Graham, C. H., (Ed.) *Vision and Visual Perception* New York: J. Wiley & sons, Inc., 1965

* Hurvich, L. M., *Color Vision: An introduction.* Sunderland, MA: Sinauer Assoc., 1980

Judd, D. B., *Contributions to Color Science* (Edited by D. MacAdam; 545) NBS special publication Washington: U. S. Government Printing Office, 1979

* Rose, A., *Vision: human and electronic.* New York: Plenum, 1973

Listings of Example Programs

Appendix A

Directory

AxesGrid:	Shows visual impact of axes and grids.
BAR_KNOB:	Shows interactivity with one degree of freedom.
BAR_KNOB2:	Shows interactivity with two degrees of freedom.
CharCell:	Relationship between characters and characters cells.
COLOR:	Demonstrates the color map.
CsizeProg:	Shows how to select character size.
DataPoint:	Supplies the data for all programs whose names start with "Sin".
DrawMdPrg:	How to specify drawing modes (draw, erase, complement).
FillProg:	Shows how to do hatched and dithered area fills.
FillGraph:	Does a broken-line chart with the area beneath the curve shaded.
GstorProg:	Storing and retrieving graphic images.
IsoProg:	Isotropic scaling.
JustProg:	Label justification.
LdirProg:	How to specify label direction.
LOCATOR:	Demonstrates interactive drawing with many types of graphics cursors.
LogPlot:	Shows how to make logarithmic axes.
MarkrProg:	Uses markers to highlight data points on a curve.
PLineProg:	Demonstrates the POLYLINE procedure.
PolyProg:	Using POLYGON procedure.
SinAspect:	Defining aspect ratio of plotting device.
SinAxes1:	Unclipped axes.
SinAxes2:	Labelled, clipped axes.
SinClip:	Clipped axes.
SinLabel1:	Single-sized, horizontal letters.
SinLabel2:	Labels with sizes and directions specified.
SinLabel3:	Bold main title.
SinLine:	No viewport, no window, not much information.
SinViewpt:	Data displayed inside framed viewport.
SinWindow:	Data mapped into user window.

AxesGrid

```
program AxesGrid(output);
import dgl_lib,dgl_inq;                    {get graphics routines}
const
  CrtAddr=            3;                   {address of internal CRT}
  ControlWord=        0;                   {device control; 0 for CRT}
type
  RoundType=          (Up, Down, Near);        {used by function Round2}
var
  Ratio:              real;
  VirtXmax, VirtYmax: real;
  LeftEdge, RightEdge: real;
  BottomEdge, TopEdge: real;
  ClipXmin, ClipXmax: real;
  ClipYmin, ClipYmax: real;
  ErrorReturn:        integer;             {variable for initialization outcome}
$page$  {********************************************************************}
procedure Frame;
{----------------------------------------------------------------------------}
{    This procedure draws a frame around the current window limits.          }
{----------------------------------------------------------------------------}
const
  WindowLimits= 450;                       {mnemonic better than magic number}
type
  LimitOrder=    (Xmin, Xmax, Ymin,Ymax);
  LimitType=     array [LimitOrder] of real;
var
  Pac:           packed array [1..1] of char;    { \  These are the sundries  }
  Iarray:        array [1..1] of integer;        {  \ needed by the call to   }
  Window:        LimitType;                      {  / the DGL procedure       }
  Error:         integer;                        { /  "inq_ws".               }
begin                                            {body of procedure "Frame"}
inq_ws(WindowLimits,0,0,4,Pac,Iarray,Window,Error);
if Error=0 then begin
  move(Window[Xmin],Window[Ymin]);               {move to lower left corner}
  line(Window[Xmin],Window[Ymax]);               {draw to upper left corner}
  line(Window[Xmax],Window[Ymax]);               {draw to upper right corner}
  line(Window[Xmax],Window[Ymin]);               {draw to lower right corner}
  line(Window[Xmin],Window[Ymin]);               {draw to lower left corner}
end  {Error=0?}
else writeln('Error ',Error:0,' occurred in "Frame"');
end;  {procedure "Frame"}                  {return}
$page$  {********************************************************************}
procedure ClipLimit(Xmin, Xmax, Ymin, Ymax: real);
{----------------------------------------------------------------------------}
{  This procedure defines the four global variables which specify where the  }
{  soft clip limits are.                                                      }
{----------------------------------------------------------------------------}
```

```
begin
if Xmin<Xmax then begin                        { \                           }
  ClipXmin:=Xmin;                              {  \    Force the minimum soft }
  ClipXmax:=Xmax;                              {   \   clip limit in X to be  }
end                                            {    \  the smaller of the two }
else begin                                     {    /  X values passed into   }
  ClipXmin:=Xmax;                              {   /   the procedure.         }
  ClipXmax:=Xmin;                              {  /                           }
end;                                           { /                            }
if Ymin<Ymax then begin                        { \                           }
  ClipYmin:=Ymin;                              {  \    Force the minimum soft }
  ClipYmax:=Ymax;                              {   \   clip limit in Y to be  }
end                                            {    \  the smaller of the two }
else begin                                     {    /  Y values passed into   }
  ClipYmin:=Ymax;                              {   /   the procedure.         }
  ClipYmax:=Ymin;                              {  /                           }
end;                                           { /                            }
end;
$page$ {*********************************************************************}
procedure ClipDraw(X1, Y1, X2, Y2: real);
{-----------------------------------------------------------------------------}
{    This procedure takes the endpoints of a line, and clips it.  The soft    }
{  clip limits are the real global variables ClipXmin, ClipXmax, ClipYmin,   }
{  and ClipYmax.  These may be defined through the procedure ClipLimit.       }
{-----------------------------------------------------------------------------}
label
  1;
type
  Edges=         (Left,Right,Top,Bottom);        {possible edges to cross}
  OutOfBounds=   set of Edges;                   {set of edges crossed}
var
  Out,Out1,Out2:OutOfBounds;
  X, Y:          real;
{-----------------------------------------------------------------------------}
procedure Code(X, Y: real; var Out: OutOfBounds);
begin                                          {nested procedure "Code"}
Out:=[];                                       {null set}
if x<ClipXmin then Out:=[left]                   {off left edge?}
else if x>ClipXmax then Out:=[right];            {off right edge?}
if y<ClipYmin then Out:=Out+[bottom]             {off the bottom?}
else if y>ClipYmax then Out:=Out+[top];          {off the top?}
end;                                           {nested procedure "Code"}
{-----------------------------------------------------------------------------}
begin                                          {body of procedure "ClipDraw"}
Code(X1,Y1,Out1);                              {figure status of point 1}
Code(X2,Y2,Out2);                              {figure status of point 2}
while (Out1<>[]) or (Out2<>[]) do begin {loop while either point out of range}
  if (Out1*Out2)<>[] then goto 1;              {if intersection non-null, no line}
  if Out1<>[] then Out:=Out1
    else Out:=Out2;                            {Out is the non-empty one}
  if left in Out then begin                    {it crosses the left edge}
    y:=Y1+(Y2-Y1)*(ClipXmin-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmin;                               {new x is left edge}
  end  {left in Out?}
  else if right in Out then begin              {it crosses right edge}
    y:=Y1+(Y2-Y1)*(ClipXmax-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmax;                               {new x is right edge}
  end  {right in Out?}
```

```
  else if bottom in Out then begin      {it crosses the bottom edge}
    x:=X1+(X2-X1)*(ClipYmin-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmin;                        {new y is bottom edge}
  end  {bottom in Out?}
  else if top in Out then begin         {it crosses the top edge}
    x:=X1+(X2-X1)*(ClipYmax-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmax;                        {new y is top edge}
  end;  {top in Out?}
  if Out=Out1 then begin
    X1:=x;   Y1:=y;   Code(x,y,Out1);   {redefine first end point}
  end  {Out=Out1?}
  else begin
    X2:=x;   Y2:=y;   Code(x,y,Out2);   {redefine second end point}
  end;  {else begin}
end;  {while}
move(x1,y1);                            {if we get to this point, the line...}
line(x2,y2);                            {...is completely visible, so draw it}
1:   end;  {procedure "ClipDraw"}       {return}
$page$  {****************************************************************}
function Round2(N, M: real; Mode: RoundType): real;
{--------------------------------------------------------------------------}
{  This function rounds "N" to the nearest "M", according to "Mode".  This   }
{  function works only when the argument is in the range of MININT..MAXINT.  }
{--------------------------------------------------------------------------}
const
  epsilon=      1E-10;                  {roundoff error fudge factor}
var
  Rounded:      real;                   {temporary holding area}
  Negative:     boolean;                {flag: "It is negative?"}
begin                                   {body of "Round2"}
Negative:=(N<0.0);                      {is the number negative?}
if Negative then begin
  N:=abs(N);                            {work with a positive number}
  if Mode=Up then Mode:=Down            {if number is negative, ...}
  else if Mode=Down then Mode:=Up;      {...reverse up and down}
end;
case Mode of                            {should we round the number...}
  Down: Rounded:=trunc(N/M)*M;          {...left on the number line?}
  Up:   begin
          Rounded:=N/M;                 {...right on the number line?}
          if abs(Rounded-round(Rounded))>epsilon then
            Rounded:=(trunc(Rounded)+1.0)*M
          else
            Rounded:=trunc(Rounded)*M;
        end;
  Near: Rounded:=trunc(N/M+M*0.5)*M;    {...to the nearest multiple?}
end;  {case}
if Negative then Rounded:=-Rounded;     {reinstate the sign}
Round2:=Rounded;                        {assign to function name}
end;                                    {function "Round2"}
```

```
$page$ {********************************************************************}
procedure XaxisClip(Spacing, Location: real; Major: integer;
  Majsize,Minsize: real);
{---------------------------------------------------------------------------}
{  This procedure draws an X-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                      }
{    Spacing:   The distance between tick marks on the axis.                 }
{    Location:  The Y-value of the X-axis.                                   }
{    Major:     The number of tick marks to ge before drawing a major tick   }
{               mark.  If Major=5, every fifth tick mark will be major.      }
{    Majsize:   The length, in world units, of the major tick marks.         }
{    Minsize:   The length, in world units, of the minor tick marks.         }
{---------------------------------------------------------------------------}
var
  X:            real;
  SemiMajsize:  real;
  SemiMinsize:  real;
  Counter:      integer;              {keeps track of when to do major ticks}
begin                                 {body of procedure "XaxisClip"}
SemiMajsize:=MajSize*0.5;
SemiMinsize:=MinSize*0.5;
Counter:=0;                           {start with a major tick}
ClipDraw(ClipXmin,Location,ClipXmax,Location);
X:=Round2(ClipXmin,Spacing*Major,Down); {round to next lower major}
while X<=ClipXmax do begin
  if Counter=0 then
    ClipDraw(X,Location-SemiMajsize,X,Location+SemiMajsize)
  else
    ClipDraw(X,Location-SemiMinsize,X,Location+SemiMinsize);
  Counter:=(Counter+1) mod Major;
  X:=X+Spacing;
end;  {while}
end;                                  {procedure "XaxisClip"}
$page$ {********************************************************************}
procedure YaxisClip(Spacing, Location: real; Major: integer;
  Majsize, Minsize: real);
{---------------------------------------------------------------------------}
{  This procedure draws an Y-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                      }
{    Spacing:   The distance between tick marks on the axis.                 }
{    Location:  The X-value of the Y-axis.                                   }
{    Major:     The number of tick marks to ge before drawing a major tick   }
{               mark.  If Major=5, every fifth tick mark will be major.      }
{    Majsize:   The length, in world units, of the major tick marks.         }
{    Minsize:   The length, in world units, of the minor tick marks.         }
{---------------------------------------------------------------------------}
var
  Y:            real;
  SemiMinsize:  real;
  SemiMajsize:  real;
  Counter:      integer;              {keeps track of when to do major ticks}
begin                                 {body of procedure "YaxisClip"}
SemiMajsize:=Majsize*0.5;
SemiMinsize:=Minsize*0.5;
Counter:=0;                           {start with a major tick}
ClipDraw(Location,ClipYmin,Location,ClipYmax);
Y:=Round2(ClipYmin,Spacing*Major,Down); {round to next lower major}
```

```
while Y<=ClipYmax do begin
  if Counter=0 then
    ClipDraw(Location-SemiMajsize,Y,Location+SemiMajsize,Y)
  else
    ClipDraw(Location-SemiMinsize,Y,Location+SemiMinsize,Y);
  Counter:=(Counter+1) mod Major;
  Y:=Y+Spacing;
end;  {while}
end;                                    {procedure "YaxisClip"}
$page$  {*****************************************************************}
procedure Grid(Xspacing,Yspacing,XlocY,YlocX: real; Xmajor, Ymajor: integer;
  Xminsize, Yminsize: real);
{---------------------------------------------------------------------------}
{  This procedure draws a grid on the plotting surface, with user-definable  }
{  minor tick size.  Parameters are as follows:                              }
{    Xspacing:  The distance between tick marks on the X axis.               }
{    Yspacing:  The distance between tick marks on the Y axis.               }
{    XlocY:     The X-value of the Y-axis.                                   }
{    YlocX:     The Y-value of the X-axis.                                   }
{    Xmin,Xmax: The left and right ends of the X-axis, respectively.         }
{    Xmajor,    The number of tick marks to ge before drawing a major tick   }
{    Ymajor:    mark.  If Major=5, every fifth tick mark will be major.      }
{    Xminsize:  The length, in world units, of the X minor tick marks.       }
{    Yminsize:  The length, in world units, of the Y minor tick marks.       }
{---------------------------------------------------------------------------}
var
  X, Y:          real;
  Xstart,Ystart:real;
  XsemiMinsize: real;
  YsemiMinsize: real;
  Counter:      integer;
begin                                   {body of procedure "Grid"}
XsemiMinsize:=Xminsize*0.5;
YsemiMinsize:=Yminsize*0.5;
Xstart:=Round2(ClipXmin,Xspacing*Xmajor,Down);  {round to next lower major}
Ystart:=Round2(ClipYmin,Yspacing*Ymajor,Down);  {round to next lower major}
{===== Draw vertical major ticks ==========================================}
X:=Xstart;
while X<=ClipXmax do begin
  ClipDraw(X,ClipYmin,X,ClipYmax);
  X:=X+Xspacing*Xmajor;
end;
{===== Draw horizontal major ticks ========================================}
Y:=Ystart;
while Y<=ClipYmax do begin
  ClipDraw(ClipXmin,Y,ClipXmax,Y);
  Y:=Y+Yspacing*Ymajor;
end;
```

```
{===== Draw vertical minor ticks ==========================================}
X:=Xstart;
Counter:=0;
while X<=ClipXmax do begin
  if Counter<>0 then begin
    Y:=Ystart;
    while Y<=ClipYmax do begin
      ClipDraw(X,Y-YSemiMinsize,X,Y+YSemiMinsize);
      Y:=Y+Yspacing;
    end;  {while Y<=ClipYmax}
  end;  {counter<>0?}
  Counter:=(Counter+1) mod Xmajor;
  X:=X+Xspacing;
end;  {while}
{===== Draw horizontal minor ticks ========================================}
Y:=Ystart;
Counter:=0;
while Y<=ClipYmax do begin
  if Counter<>0 then begin
    X:=Xstart;
    while X<=ClipXmax do begin
      ClipDraw(X-XSemiMinsize,Y,X+XSemiMinsize,Y);
      X:=X+Xspacing;
    end;  {while X<=ClipXmax}
  end;  {counter<>0?}
  Counter:=(Counter+1) mod Ymajor;
  Y:=Y+Yspacing;
end;  {while}
end;                                        {procedure "Grid"}
$page$  {*****************************************************************}
begin                                       {program "AxesGrid"}
graphics_init;
display_init(CrtAddr,ControlWord,ErrorReturn);
if ErrorReturn=0 then begin
  {=== Do program setup ===================================================}
  Ratio:=511/389;
  set_aspect(Ratio,1);
  if Ratio>1 then begin
    VirtXmax:=1;
    VirtYmax:=1/Ratio;
  end
  else begin
    VirtXmax:=Ratio;
    VirtYmax:=1;
  end;
  {=== Upper left viewport ================================================}
  LeftEdge:=0;
  RightEdge:=0.48*VirtXmax;
  BottomEdge:=0.52*VirtYmax;
  TopEdge:=VirtYmax;
  set_viewport(LeftEdge,RightEdge,BottomEdge,TopEdge);
  set_window(0,80,0,40);
  Frame;
  ClipLimit(0,80,0,40);
  XaxisClip(1,0,5,2,1);
  YaxisClip(1,0,5,2,1);
```

```
  {=== Upper right viewport ===================================================}
  LeftEdge:=0.52*VirtXmax;
  RightEdge:=VirtXmax;
  BottomEdge:=0.52*VirtYmax;
  TopEdge:=VirtYmax;
  set_viewport(LeftEdge,RightEdge,BottomEdge,TopEdge);
  set_window(0,80,0,40);
  Frame;
  ClipLimit(0,80,0,40);
  Grid(5,5,0,0,4,4,1,0.8);
  {=== Lower left viewport ====================================================}
  LeftEdge:=0;
  RightEdge:=0.48*VirtXmax;
  BottomEdge:=0;
  TopEdge:=0.48*VirtYmax;
  set_viewport(LeftEdge,RightEdge,BottomEdge,TopEdge);
  set_window(0,80,0,40);
  Frame;
  ClipLimit(0,80,0,40);
  Grid(2,1,0,0,10,10,0.001,0.001);
  {=== Lower right viewport ===================================================}
  LeftEdge:=0.52*VirtXmax;;
  RightEdge:=VirtXmax;
  BottomEdge:=0;
  TopEdge:=0.48*VirtYmax;;
  set_viewport(LeftEdge,RightEdge,BottomEdge,TopEdge);
  set_window(0,80,0,40);
  Frame;
  ClipLimit(0,80,0,40);
  XaxisClip(1,0,5,2,1);
  YaxisClip(1,0,5,2,1);
  XaxisClip(1,40,5,2,1);
  YaxisClip(1,80,5,2,1);
  Grid(10,10,0,0,1,1,2,2);
end;  {ErrorReturn=0?}
graphics_term;
end.                                        {program "AxesGrid"}
```

BAR_KNOB

```
$ucsd,debug$
program Test (Keyboard,output);
import dgl_vars,dgl_types,dgl_lib,dgl_inq;
type
  States=               (On,Off);
  DrawMode=             (Draw,Erase,Comp,NonDom);
const
  FS=                   chr(28);
  BS=                   chr(8);
  US=                   chr(31);
  LF=                   chr(10);
  CR=                   chr(13);
  Q=                    'Q';
  Q1=                   'q';
  Underline=            chr(132);
  Ind_off=              chr(128);
  Inv_On=               chr(129);
  MinBarY=              0;
  MaxBarY=              100;
  MinBarX=              180;
  MaxBarX=              220;
  IncDelta=             0.1;
var
  Error_num:            integer;
  I,TempInt:            integer;
  Level,LastLevel:      real;
  Delta:                real;
  CharWidth,CharHeight: real;
  Character:            char;
  Done:                 boolean;
  Keyboard:             text;
  TempString:          Gstring255;
$page$  {*********************************************************************}
procedure GraphicsDisplay(State:States {On/Off});
const
  GraphicsDisp=         1050;
var
  Error:integer;
  SwitchArray:integer;
  Dummy:real;
begin                                   {procedure GraphicsDisplay}
case State of
  On:SwitchArray:=1;
  Off:SwitchArray:=0;
end; {case State of}
output_esc(GraphicsDisp,1,0,SwitchArray,Dummy,Error) ;
if Error <> 0 then
  writeln ('Error ',Error:1,' encountered in GraphicsDisplay');
end;                                    {procedure GraphicsDisplay}
```

```
$page$ {*****************************************************************}
procedure AlphaDisplay(State:States {On/Off});
const
  AlphaDisp=1051;
var
  Error:integer;
  SwitchArray:integer;
  Dummy:real;
begin                                  {procedure AlphaDisplay}
case State of
  On:SwitchArray:=1;
  Off:SwitchArray:=0;
end; {case State of}
output_esc(AlphaDisp,1,0,SwitchArray,Dummy,Error) ;
if Error <> 0 then
  writeln ('Error ',Error:1,' encountered in AlphaDisplay');
end;                                   {procedure AlphaDisplay}
$page$ {*****************************************************************}
begin                                  {Main Program}
Level:=0;                              {current height of bar}
LastLevel:=Level;                      {previous height of bar}
graphics_init;                         {initialize the graphics system}
display_init(3,0,Error_Num);           {which output device?}
if Error_Num=0 then begin              {output device initialization OK?}
  AlphaDisplay(Off);                   {turn off alpha display}
  GraphicsDisplay(On);                 {turn on graphics display}
  set_aspect(511,389);                 {use whole screen}
  set_window(0,400,-30,120);           {scale the window for the data}
  set_color(1);                        {color number 1: white}
  CharWidth:=(0.035*400);              {char width: 3.5% of screen width}
  CharHeight:=(0.05*150);              {char height: 5% of screen height}
  set_char_size(CharWidth, CharHeight); {install character size}
  {---- Outline the Bar -----------------------------------------------}
  move(MinBarX-0.5,MinBarY-0.5);       {move to lower left corner...}
  line(MinBarX-0.5,MaxBarY+0.5);       {...draw to upper left corner...}
  line(MaxBarX+0.5,MaxBarY+0.5);       {...draw to upper right corner...}
  line(MaxBarX+0.5,MinBarY-0.5);       {...draw to lower left corner...}
  line(MinBarX-0.5,MinBarY-0.5);       {...and draw to lower left corner.}
  {---- Label the bar (numeric labels) -------------------------------}
  for I:=0 to 10 do begin
    strwrite(TempString,1,TempInt,I*10:3,'-');
    move (179-strlen(TempString)*CharWidth,I*10-0.24*CharHeight);
    gtext (TempString);
  end; {for I:=1 to 10 }
  {---- Label the bar (textual labels) -------------------------------}
  move (221, 80-CharHeight/2);
  gtext ('-High Normal');
  move (221, 60-CharHeight/2);
  gtext ('-Low Normal');
  {---- How about some instructions ----------------------------------}
  CharWidth:=(0.02*400);               {char width: 2% of screen width}
  CharHeight:=(0.035*150);             {char height: 3.5% of screen height}
  set_char_size(CharWidth, CharHeight); {install character size}
  move (0, 5);
  TempString:='Use the Knob to'+CR+LF;
  gtext (TempString);
  TempString:='Adjust the value.'+CR+LF;
  gtext (TempString);
```

```
  TempString:=' '+CR+LF;
  gtext (TempString);
  TempString:='SHIFT with the Knob '+CR+LF;
  gtext (TempString);
  TempString:='speeds it up.'+CR+LF;
  gtext (TempString);
  TempString:='';
  {---- Set a good character size ------------------------------------------}
  CharWidth:=(0.035*400);             {char width: 3.5% of screen width}
  CharHeight:=(0.05*150);             {char height: 5% of screen height}
  set_char_size(CharWidth, CharHeight); {install character size}
  repeat
    read(Keyboard,Character);         {get character without echo to screen}
    Delta:=0;                         {start by assuming no motion}
    case Character of                 {what's the character?}
      FS:   Delta:=IncDelta;          {right arrow?}
      BS:   Delta:=-IncDelta;         {left arrow (backspace)?}
      LF:   Delta:=10*IncDelta;       {down arrow?}
      US:   Delta:=-10*IncDelta;      {up arrow?}
      Q,Q1: Done:=TRUE;               {or Quit?}
    otherwise                         {if none of the above, ignore it}
    end; {case ord(Character)}
    if Delta>0 then begin             {Going Up}
      set_color(1);                   {we want to draw lines}
      while (Level<LastLevel+Delta) and (Level<MaxBarY-IncDelta) do begin
        Level:=Level+IncDelta;        {new top of bar}
        move(MinBarX,Level);          {move to left edge...}
        line(MaxBarX,Level);          {...and draw to right edge}
      end  {while (Level<LastLevel) and (Level<MaxBarY)}
    end  {if (Delta>0) and (Level<100) }
    else begin                        {Going Down}
      if (Delta<0) and (Level>=0.5*IncDelta) then begin
        set_color(0);                 {we want to erase lines}
        repeat
          move(MinBarX, Level);       {move to the left edge...}
          line(MaxBarX, Level);       {...and draw to the right edge}
          Level:=Level-IncDelta;      {new top of bar}
        until (Level<=LastLevel+Delta) or (Level<=MinBarY)
      end;  {if (Delta<0) and (Level>0)}
    end;
    {---- How about some numbers? ------------------------------------------}
    set_color(0);                     {we want to erase lines}
    strwrite(TempString,1,TempInt,LastLevel:5:1);   {convert level to chars}
    move(MinBarX+(MaxBarX-MinBarX)/2-strlen(TempString)*CharWidth/2,
      MinBarY-2*CharHeight);
    gtext(TempString);                {erase the old number}
    set_color(1);                     {we want to erase lines}
    strwrite(TempString,1,TempInt,Level:5:1);
    move (MinBarX+(MaxBarX-MinBarX)/2-strlen(TempString)*CharWidth/2,
      MinBarY-2*CharHeight);
    gtext(TempString);                {write the new}
    LastLevel:=Level;                 {remember the old number}
  until Done;                         {repeat until user hits [Q]}
  GraphicsDisplay (Off);              {turn off graphics display}
  AlphaDisplay (On);                  {turn on alpha display}
  display_term;                       {clean up loose ends}
end;
graphics_term;                        {terminate the graphics package}
end.                                  {main program}
```

BAR_KNOB2

```
$ucsd,debug$
program Test(Keyboard,output);
import dgl_vars,dgl_types,dgl_lib,dgl_inq;
type
  DrawMode=               (Draw,Erase,Comp,NonDom);
  BarX=                   array[1..5] of integer;
  States=                 (On,Off);
const
  FS=                     chr(28);
  BS=                     chr(8);
  US=                     chr(31);
  LF=                     chr(10);
  CR=                     chr(13);
  Q=                      'Q';
  Q1=                     'q';
  Underline=              chr(132);
  Ind_off=                chr(128);
  Inv_On=                 chr(129);
  MinBarY=                0;
  MaxBarY=                100;
  MinBarX=                BarX[40,130,220,310,400];
  MaxBarX=                BarX[80,170,260,350,440];
  IncDelta=               0.1;
var
  Error_num:              integer;
  I,TempInt:              integer;
  Level,LastLevel:        array [1..5] of real;
  Bar:                    integer;
  Delta:                  real;
  CharWidth,CharHeight:   real;
  Character:              char;
  Done:                   boolean;
  Keyboard:               text;
  TempString:             Gstring255;
$page$  {*****************************************************************}
procedure SetDrawMode(Mode: DrawMode);
const
  OpSelector=             1052;           {mnemonic better than magic number}
var
  IntArray:               integer;        { \   All this stuff is needed  }
  RealArray:              integer;        {  >  by the DGL procedure      }
  Error:                  integer;        { /   OUTPUT_ESC.               }
begin                                     {procedure SetDrawMode}
 case Mode of
   Draw:     IntArray:=0;                 { \                             }
   NonDom:   IntArray:=1;                 {  \   Magic numbers for the    }
   Erase:    IntArray:=2;                 {  /   four drawing modes.      }
   Comp:     IntArray:=3                  { /                             }
 end;  {case Mode of}
 output_esc(OpSelector,1,0,IntArray,RealArray,Error);
end;                                      {procedure SetDrawMode}
```

```
$page$ {*********************************************************************}
procedure GraphicsDisplay(State: States {On/Off});
const
  GraphicsDisp=         1050;          {mnemonic better than magic number}
var
  Error:                integer;       { \   All this stuff is needed  }
  SwitchArray:          integer;       {  >  by the DGL procedure      }
  Dummy:                real;          { /   OUTPUT_ESC.               }
begin                                  {procedure GraphicsDisplay}
  case State of
    On:   SwitchArray:=1;              {1=on, and...}
    Off:  SwitchArray:=0;              {0=off.}
  end; {case State of}
  output_esc(GraphicsDisp,1,0,SwitchArray,Dummy,Error);
  if Error<>0 then
    writeln('Error ',Error:1,' encountered in GraphicsDisplay');
end;                                   {procedure GraphicsDisplay}
$page$ {*********************************************************************}
procedure AlphaDisplay(State: States {On/Off});
const
  AlphaDisp=            1051;          {mnemonic better than magic number}
var
  Error:                integer;       { \   All this stuff is needed  }
  SwitchArray:          integer;       {  >  by the DGL procedure      }
  Dummy:                real;          { /   OUTPUT_ESC.               }
begin                                  {procedure AlphaDisplay}
   case State of
      On:   SwitchArray:=1;            {1=on, and...}
      Off:  SwitchArray:=0;            {0=off.}
   end; {case State of}
   output_esc(AlphaDisp,1,0,SwitchArray,Dummy,Error);
   if Error<>0 then
      writeln('Error ',Error:1,' encountered in AlphaDisplay');
end;                                   {procedure AlphaDisplay}
$page$ {*********************************************************************}
procedure ClearInd(Bar: integer);
begin                                  {procedure ClearInd}
  SetDrawMode(Erase);
  move(MinBarX[Bar],MinBarY-1.3*CharHeight);
  line(MaxBarX[Bar],MinBarY-1.3*CharHeight);
end;                                   {procedure ClearInd}
$page$ {*********************************************************************}
procedure SetInd(Bar: integer);
begin                                  {procedure SetInd }
  SetDrawMode(Draw);
  move(MinBarX[Bar],MinBarY-1.3*CharHeight);
  line(MaxBarX[Bar],MinBarY-1.3*CharHeight);
end;                                   {procedure SetInd }
```

```
$page$  {*******************************************************************}
procedure UpdateValue(Bar:integer);
var
  LastCharWidth,LastCharHeight: real;
begin                                   {procedure UpdateValue }
LastCharWidth:=CharWidth;               {store old character width}
LastCharHeight:=CharHeight;             {store old character height}
CharWidth:=(0.025*512);                 {new char width: 2.5% of screen width}
CharHeight:=(0.045*150);                {new char width: 2.5% of screen height}
set_char_size(CharWidth,CharHeight);    {install the character size}
{---- Erase the old -------------------------------------------------------}
SetDrawMode(Erase);                     {draw with black lines}
TempString:='';                         {null out any old value}
strwrite(TempString,1,TempInt,LastLevel[Bar]:5:1);      {convert to string}
move(MinBarX[Bar]+(MaxBarX[Bar]-MinBarX[Bar])/2-        {move to right place}
  strlen(TempString)*CharWidth/2,MinBarY-2.5*CharHeight);
gtext(TempString);                      {label the string}
{---- Write the new -------------------------------------------------------}
SetDrawMode(Draw);                      {draw with white lines}
TempString:='';                         {null out any old value}
strwrite(TempString,1,TempInt,Level[Bar]:5:1);          {convert to string}
move(MinBarX[Bar]+(MaxBarX[Bar]-MinBarX[Bar])/2-        {move to right place}
  strlen(TempString)*CharWidth/2,MinBarY-2.5*CharHeight);
gtext(TempString);                      {label the string}
{---- Reinstate the old character size ------------------------------------}
CharWidth:=LastCharWidth;               {restore old character width}
CharHeight:=LastCharHeight;             {restore old character height}
set_char_size(CharWidth,CharHeight);    {install old character size}
end;  {procedure UpdateValue }
$page$  {*******************************************************************}
begin                                   {Main Program}
graphics_init;                          {initialize the graphics system}
display_init(3,0,Error_Num);            {which output device?}
if Error_Num=0 then begin               {output device initialization OK?}
  AlphaDisplay(Off);                    {turn the alpha display off}
  GraphicsDisplay(On);                  {turn the graphics display on}
  set_aspect(511,389);                  {use the whole screen}
  set_window(0,511,-50,110);            {scale the window for the data}
  set_color(1);                         {draw with white}
  SetDrawMode(Draw);                    {dominant drawing mode}
  CharWidth:=(0.020*400);               {char width: 2% of screen width}
  CharHeight:=(0.035*150);              {char height: 3.5% of screen height}
  set_char_size(CharWidth,CharHeight);  {install the character size}
  {---- Make the Bars -------------------------------------------------------}
```

```
for Bar:=1 to 5 do begin
  {---- Initialize the levels ------------------------}
  Level[Bar]:=0;                       {all bars at level zero}
  LastLevel[Bar]:=Level[Bar];          {old values, too}
  {---- Outline the Bar ------------------------------}
  move(MinBarX[Bar]-1,MinBarY-(160/389));     {move to lower left corner...}
  line(MinBarX[Bar]-1,MaxBarY+(160/389));     {...draw to upper left...}
  line(MaxBarX[Bar]+1,MaxBarY+(160/389));     {...draw to upper right...}
  line(MaxBarX[Bar]+1,MinBarY-(160/389));     {...draw to lower right...}
  line(MinBarX[Bar]-1,MinBarY-(160/389));     {...and draw to lower left.}
  {---- Label the bar -------------------------------}
  TempString:='';                      {null out any old value}
  strwrite(TempString,1,TempInt,'Bar ',Bar:1);        {convert to string}
  move(MinBarX[Bar]+(MaxBarX[Bar]-MinBarX[Bar])/2-    {move to right place}
    strlen(TempString)*CharWidth/2,MinBarY-1.25*CharHeight);
  gtext(TempString);                   {label the text}
  {---- Put numbers alongside the bars ---------------}
  for I:=0 to 10 do begin
    TempString:='';                    {null out any old value}
    strwrite(TempString,1,TempInt,I*10:3,'-');        {convert to string}
    move(MinBarX[Bar]-strlen(TempString)*CharWidth,   {move to right place}
      I*10-0.24*CharHeight);
    gtext(TempString);                 {label the text}
  end; {for I:=1 to 10 }
  UpdateValue(Bar);                    {modify the bar}
end;  {for}
{---- How about some instructions --------------------}
CharWidth:=(0.02*511);                 {char width: 2% of screen width}
CharHeight:=(0.035*160);               {char height: 3.5% of screen height}
set_char_size(CharWidth,CharHeight);   {install character size}
move(0,-30);
TempString:='Use Number Keys to select a bar.'+CR+LF;
gtext(TempString);
TempString:=''+CR+LF;
gtext(TempString);
TempString:='Use the Knob to adjust the value.'+CR+LF;
gtext(TempString);
TempString:='SHIFT speeds up the Knob.'+CR+LF;
gtext(TempString);
{---- Start the interactivity ------------------------}
Bar:=3;                                {which bar active at first?}
SetInd(Bar);                           {tell the program so}
repeat
  read(Keyboard,Character);            {read character with no echo to screen}
  Delta:=0;                            {assume no motion until told otherwise}
  case Character of
    FS:   Delta:=IncDelta;             {right arrow?}
    BS:   Delta:=-IncDelta;            {left arrow (or backspace)?}
    LF:   Delta:=10*IncDelta;          {down arrow?}
    US:   Delta:=-10*IncDelta;         {up arrow?}
    Q,Ql: Done:=true;                  {or Quit?}
    '1'..'5': begin
        ClearInd(Bar);                 {deactivate old bar}
        Bar:=ord(Character)-ord('0'); {determine new bar's number}
        SetInd(Bar);                   {activate new bar}
      end;
    otherwise                          {if none of the above, do nothing}
  end; {case}
```

```
    if(Delta>0)then begin                     {Going Up}
      SetDrawMode(Draw);                      {draw with white lines}
      while (Level[Bar]<LastLevel[Bar]+Delta)
          and (Level[Bar]<MaxBarY-IncDelta)do begin
        Level[Bar]:=Level[Bar]+IncDelta;          {calculate new level}
        move(MinBarX[Bar],Level[Bar]);  {move to the left end...}
        line(MaxBarX[Bar],Level[Bar]);  {...and draw to the right edge}
      end  {while}
    end  {if}
    else begin                                {delta<0}
      if (Delta<0) and (Level[Bar]>=0.5*IncDelta) then begin    {Going Down}
        SetDrawMode(Erase);                   {draw with black lines}
        repeat
          move(MinBarX[Bar],Level[Bar]);          {move to left edge...}
          line(MaxBarX[Bar],Level[Bar]);          {...and draw to right edge}
          Level[Bar]:=Level[Bar]-IncDelta;        {decrement level}
        until (Level[Bar]<=LastLevel[Bar]+Delta) or (Level[Bar]<=MinBarY)
      end;  {if}
    end;  {else}
    {---- How about some numbers? ----------------------}
    UpdateValue(Bar);                         {change the bar's numeric label}
    LastLevel[Bar]:=Level[Bar];               {remember the current value}
  until Done;                                 {pressed [Q] yet?}
  GraphicsDisplay(Off);                       {turn off graphics display}
  AlphaDisplay(On);                           {turn on alpha display}
  display_term;                               {clean up loose ends}
end;
graphics_term;                                {terminate the graphics system}
end.   {Main Program}
```

CharCell

```
program CharCell(output);                    {program name same as file name}
import dgl_lib, dgl_inq;                     {access the necessary procedures}
const
  Crt=                  3;                   {device address of graphics raster}
  Control=              0;                   {device control word; ignored for CRT}
type
  LorgType=             1..9;                {the valid values to pass the "Lorg"}
  Str255=               string[255];         {for the procedure "Glabel"}
var
  Error:                integer;             {display_init return variable; 0 = ok}
  I, X, Y:              integer;             {loop control variables}
$page$  {********************************************************************}
begin                                        {body of program "CharCell"}
graphics_init;                               {initialize graphics library}
display_init(Crt,Control,Error);             {initialize CRT}
if Error=0 then begin                        {if no error occurred...}
  set_aspect(511,389);                       {use the whole screen}
  move(-1,-1);  line(-1,1);  line(1,1);  line(1,-1);  line(-1,-1);
  set_window(-2,38,-7.5,22.5);               {define appropriate window}
  set_char_size(1,2);                                { \                       }
  move(1,21);                                        {  >  Do main label.      }
  gtext('Size of Character in Character Cell'); { /                       }
  for X:=0 to 36 do begin                    { \                         }
    for Y:=0 to 15 do begin                  {  \                        }
      move(X-0.1,y+0.1);                     {   \   Draw the four 9x15    }
      line(X+0.1,Y-0.1);                     {    \  character cells. Make }
      move(X+0.1,Y+0.1);                     {    /  a frame around each,  }
      line(X-0.1,Y-0.1);                     {   /   and an X at every     }
    end;  {for y}                            {  /    point.                }
  end;  {for x}                              { /                         }
  for I:=0 to 3 do begin                     {draw a frame around each char cell}
    move(I*9,0); line(I*9,15); line(I*9+9,15); line(I*9+9,0); line(I*9,0);
  end;
  set_char_size(9,15);                       {big characters}
  move(1,4);                                 {go to starting position}
  gtext('Gby;');                             {label some characters}
end;  {Error=0?}                             {end of conditional code}
graphics_term;                               {terminate graphics library}
end.  {program "CharCell"}                   {end of program}
```

COLOR

```
$ucsd,debug$
program Test(keyboard,output);
import dgl_vars,dgl_types,dgl_lib,dgl_poly,dgl_inq;
type
  Colors=           (Red,Yellow,Green,Cyan,Blue,Magenta,White,Black);
  Modes=            (Hue,Sat,Lum,Table,Copy1,Copy2);
  EntryRange=       -1..16;
  FunnyArray=       array [Colors] of char; {array for alpha color}
const
  FS=               chr(28);      {right arrow}
  BS=               chr(8);       {left arrow or backspace}
  US=               chr(31);      {up arrow}
  LF=               chr(10);      {down arrow}
  CR=               chr(13);      {carriage return}
  C=                'C';          { \                         }
  C1=               'c';          {  \                        }
  E=                'E';          {   \                       }
  E1=               'e';          {    \                      }
  H=                'H';          {     \                     }
  H1=               'h';          {      \ These are the valid  }
  L=                'L';          {      / user responses.    }
  L1=               'l';          {     /                     }
  Q=                'Q';          {    /                      }
  Q1=               'q';          {   /                       }
  S=                'S';          {  /                        }
  S1=               's';          { /                         }
  Display_Cont=     1050;         {mnemonic better than magic number}
  Underline=        chr(132);     {alpha enhancement: underlining}
  Ind_off=          chr(128);     {turn enhancements off}
  Inv_On=           chr(129);     {alpha enhancement: inverse video}
  FunnyChar=        FunnyArray[chr(139),chr(137),   {\  Array for   }
                           chr(136),chr(140),   { \ holding the }
                           chr(142),chr(143),   { / alpha-color }
                           chr(141),chr(138)];  {/  controllers }
```

```
$page$ {*************************************************************}
var
  Error_num:            integer;        {return variable}
  I,TempInt:            integer;        {temporary variables}
  OpArray:              array[1..5] of gshortint;
  Xarray,Yarray,                                {locations of points}
  Yfill_array:          array[1..5] of real;    {same points, but filled}
  Delta:                real;
  HueVal:               array[0..15] of real;   { \   For each of the   }
  SatVal:               array[0..15] of real;   {  \  sixteen pens, we  }
  LumVal:               array[0..15] of real;   {   \ need to know the  }
  GreenVal:             array[0..15] of real;   {   / HSL values as     }
  BlueVal:              array[0..15] of real;   {  /  well as the RGB   }
  RedVal:               array[0..15] of real;   { /   values.           }
  Hue_ind:              char;           { \                             }
  Sat_ind:              char;           {  >  Various indicators.       }
  Lum_ind:              char;           { /                             }
  Tab_ind:              char;           {and another}
  Character:            char;           {utility variable}
  Done:                 boolean;        {are we through yet?}
  Keyboard:             text;           {non-echoing input}
  TempString:           Gstring255;     {temporary holding place for text}
  Mode,LastMode:        Modes;
  CursorColor:          EntryRange;
  Copy_Source:          EntryRange;
  LastTableEntry:       EntryRange;
  TableEntry:           EntryRange;
  X_Loc:                array[0..15] of integer;
  Y_Loc:                array[0..15] of integer;
  Int_a:                integer;
  Real_A:               real;
  RedBack,GreenBack,BlueBack:   real;
  LabelColor:           char;
  BackSum,OldBackSum:   0..7;
$page$ {*************************************************************}
procedure MenuLine;
begin                                   {procedure MenuLine}
writeln(LabelColor);                    {write in appropriate color}
gotoxy(0,0);                            {go to upper left corner of the screen}
write('Color Selector:',Underline,'H',Ind_off,'ue,');
write(Underline,'S',Ind_off,'aturation,');
write(Underline,'L',Ind_off,'uminosity,');
write('table ',Underline,'E',Ind_off,'ntry,');
write(Underline,'C',Ind_off,'opy color,');
writeln(Underline,'Q',Ind_off,'uit');
end;  {procedure MenuLine}
```

```
$page$ {********************************************************************}
procedure DisplayStuff;
begin                                {procedure DisplayStuff}
writeln(LabelColor);                 {write in appropriate color}
case Mode of                         {which value are we tweaking?}
  Hue:          Hue_Ind:=Inv_On;     { \                           }
  Sat:          Sat_Ind:=Inv_On;     {  \  Turn on the display     }
  Lum:          Lum_Ind:=Inv_On;     {   > enhancement for the     }
  Table:        Tab_Ind:=Inv_On;     {  /  appropriate indicator.  }
  Copy1,Copy2:  {No Indicators on};  { /                           }
end;  {case}
gotoxy(0,3);                         {fourth row, first column}
writeln(Hue_ind,' Hue ',Ind_off);
writeln(HueVal[TableEntry]:5:2);
writeln(Sat_ind,' Sat ',Ind_off);
writeln(SatVal[TableEntry]:5:2);
writeln(Lum_ind,' Lum ',Ind_off);
writeln(LumVal[TableEntry]:5:2);
writeln(Tab_ind,' Entry ',Ind_off);
writeln(TableEntry:3);
gotoxy(0,20);                        {twenty-first row, first column}
Hue_ind:=chr(128);                   { \                           }
Sat_ind:=chr(128);                   {  \     Turn all display     }
Lum_ind:=chr(128);                   {  /     enhancements off     }
Tab_ind:=chr(128);                   { /                           }
end;  {procedure DisplayStuff}
$page$ {********************************************************************}
procedure UpdateCursor(TableEntry:integer);
{----------------------------------------------------------------------------}
procedure DrawCursor(TableEntry:integer);
begin                                {procedure DrawCursor}
Xarray[1]:=X_Loc[TableEntry]+0.1;    { \                           }
Xarray[2]:=X_Loc[TableEntry]+0.5;    {  \                          }
Xarray[3]:=X_Loc[TableEntry]+0.9;    {   \                         }
Xarray[4]:=X_Loc[TableEntry]+0.1;    {    \  Define the triangular }
Yarray[1]:=Y_Loc[TableEntry]-0.09;   {    /  cursor.               }
Yarray[2]:=Y_Loc[TableEntry]-0.01;   {   /                         }
Yarray[3]:=Y_Loc[TableEntry]-0.09;   {  /                          }
Yarray[4]:=Y_Loc[TableEntry]-0.09;   { /                           }
set_pgn_style(15);                   {entry #15 from the polygon table}
polygon_dev_dep(4,Xarray,Yarray,OpArray);    {draw the cursor}
end;                                 {procedure DrawCursor}
{----------------------------------------------------------------------------}
begin                                {procedure UpdateCursor}
if LastTableEntry<>TableEntry then begin     {any change?}
  set_pgn_color(0);                  {choose fill color of background}
  DrawCursor(LastTableEntry)         {draw in background color (erase)}
end; {if}
set_pgn_color(CursorColor);          {select polygon color}
DrawCursor(TableEntry);              {draw the new cursor}
LastTableEntry:=TableEntry           {remember the new cursor position}
end;  {procedure UpdateCursor}
```

```
$page$  {*****************************************************************}
begin                                   {Main Program}
Hue_ind:=chr(128);                      { \                          }
Sat_ind:=chr(128);                      {  \  All highlights initially  }
Lum_ind:=chr(128);                      {  /  off.                   }
Tab_ind:=chr(128);                      { /                          }
TableEntry:=0;                          {currently indicated entry}
LastTableEntry:=0;                      {previously indicated entry}
Mode:=Table;                            {selection mode first}
CursorColor:=1;                         {make sure the cursor is visible}
LabelColor:=FunnyChar[Black];           {labels contrast with background}

graphics_init;                          {initialize the graphics system}
display_init(3,0,Error_Num);            {which output device?}
if Error_Num=0 then begin               {successfully initialized}
  set_char_size(0.175,0.15);            {define the character size}
  set_pgn_style(15);                    {select the polygon style}
  set_aspect(511,389);                  {use the whole screen}
  set_window(-1.1,8,-0.7,2.2);          {scale the window for the data}
  {---- Set up color system  and  set background color ----------------------}
  set_color_model(2);                   {HSL}
  HueVal[TableEntry]:=0;                { \  Current TableEntry: 0.     }
  SatVal[TableEntry]:=0;                {  > Current entry's color:     }
  LumVal[TableEntry]:=0.6;              { /  60% gray.                  }
  set_color_table(TableEntry,           { \                             }
                HueVal[TableEntry],     {  \ Install the currently-     }
                SatVal[TableEntry],     {  / defined color.             }
                LumVal[TableEntry]);    { /                             }
  {---- Read the colors from the color map ---------------------------------}
  for I:=0 to 15 do inq_color_table(I,HueVal[I],SatVal[I],LumVal[I]);
  {---- Initialize arrays for polygon --------------------------------------}
  OpArray[1]:=2;                        {2: First vertex of a polygon}
  for I:=2 to 5 do OpArray[I]:=1;       {1: Draw from the last vertex to this}
  {---- Set up arrays for the lower row -------------------------------------}
  Yarray[1]:=0.1;                       { \                             }
  Yarray[2]:=0.1;                       {  \  Define the outline of     }
  Yarray[3]:=0.9;                       {   > the tall, unfilled        }
  Yarray[4]:=0.9;                       {  /  rectangle.                }
  Yarray[5]:=0.1;                       { /                             }
  Yfill_array:=Yarray;                  { \   Define the outline of     }
  Yfill_array[3]:=0.5;                  {  >  the short, filled         }
  Yfill_array[4]:=0.5;                  { /   rectangle.                }
  {---- Draw the lower row -------------------------------------------------}
  for I:=0 to 7 do begin
    Xarray[1]:=I;                       { \                             }
    Xarray[2]:=I+0.9;                   {  \  Define the X positions    }
    Xarray[3]:=I+0.9;                   {   > for this particular       }
    Xarray[4]:=I;                       {  /  rectangle.                }
    Xarray[5]:=I;                       { /                             }
    strwrite(TempString,1,TempInt,I:2); {convert to a string}
    set_color(1);                       {set the color for text}
    move(I+0.5-0.075,0);                {move to just right of bottom center}
    gtext(TempString);                  {label the table entry number}
    set_pgn_color(I);                   {set the color for polygon fills}
    set_color(I);                       {set the color for lines}
    polyline(5,Xarray,Yarray);          {draw the tall unfilled rectangle}
    polygon_dev_dep(5,Xarray,Yfill_array,OpArray);      {draw and fill shortie}
    X_Loc[I]:=round(Xarray[1]);         {store X locations}
    Y_Loc[I]:=round(Yarray[1]);         {store Y locations}
  end; {for I:=0 to 15}
```

```
{---- Set up the arrays for the upper row ----------------------------------}
Yarray[1]:=1.1;                 { \                             }
Yarray[2]:=1.1;                 {   \                           }
Yarray[3]:=1.9;                 {    >  Redefine Y values only.  }
Yarray[4]:=1.9;                 {   /                           }
Yarray[5]:=1.1;                 { /                             }
Yfill_array:=Yarray;            {   \                           }
Yfill_array[3]:=1.5;            {    >  Redefine Y values only.  }
Yfill_array[4]:=1.5;            {   /                           }
{---- Draw the upper row ---------------------------------------------------}
for I:=0 to 7 do begin
  Xarray[1]:=I;                 { \                             }
  Xarray[2]:=I+0.9;             {   \  Define the X positions   }
  Xarray[3]:=I+0.9;             {    > for this particular      }
  Xarray[4]:=I;                 {   /  rectangle.               }
  Xarray[5]:=I;                 { /                             }
  strwrite(TempString,1,TempInt,I+8:2);      {convert to a string}
  set_color(1);                 {set the color for text}
  move(I+0.5-0.075,1);          {move to just right of bottom center}
  gtext(TempString);            {label the table entry number}
  set_pgn_color(I+8);           {set the color for polygon fills}
  set_color(I+8);               {set the color for lines}
  polyline(5,Xarray,Yarray);    {draw the tall unfilled rectangle}
  polygon_dev_dep(5,Xarray,Yfill_array,OpArray);      {draw and fill shortie}
  X_Loc[I+8]:=round(Xarray[1]);       {store X locations}
  Y_Loc[I+8]:=round(Yarray[1]);       {store Y locations}
end; {for I:=0 to 15}
{---- Start interactivity --------------------------------------------------}
MenuLine;                       {write the menu}
UpdateCursor(TableEntry);       {initial cursor}
DisplayStuff;                   {initial readouts}
Done:=false;                    {not done yet}
repeat                          {this starts the actual color selector}
  read(Keyboard,Character);     {get a character,no echo}
  Delta:=0;                     {start by assuming zero}
  case Character of             {analyze the character}
    FS:   Delta:=0.01;          { \                             }
    BS:   Delta:=-0.01;         {   \  Cursor-control           }
    LF:   Delta:=0.1;           {   /  characters               }
    US:   Delta:=-0.1;          { /                             }
    H,Hl: Mode:=Hue;            {Hue-changing mode}
    L,Ll: Mode:=Lum;            {Luminosity-changing mode}
    Q,Ql: Done:=true;           {Quit the program}
    S,Sl: Mode:=Sat;            {Saturation-changing mode}
    E,El: Mode:=Table;          {Entry-changing mode}
    C,Cl: begin
        if Mode=Copy1 then begin    {Have source, will copy.}
          Copy_Source:=TableEntry;  {Put it where?}
          CursorColor:=Copy_Source;
          UpdateCursor(TableEntry); {note current entry}
          gotoxy(0,21);             {twenty-second row, first column}
          write('Use Knob to select location to ');
          write('copy color to,then press');
          writeln(' C');
          Mode:=Copy2               {do second section next time}
        end
```

```
      else begin
        if Mode=Copy2 then begin  {Copy color to}
          gotoxy(0,21);           {this location}
          writeln(strrpt(' ',79));        {"erase" old text}
          inq_color_table(Copy_Source,            { \  Get the    }
                          HueVal[TableEntry],     {  \ HSL values }
                          SatVal[TableEntry],     {  / from the   }
                          LumVal[TableEntry]);    { /  table.     }
          CursorColor:=1;                 {reinitialize cursor color}
          UpdateCursor(TableEntry);       {indicate new cursor position}
          Mode:=LastMode                  {third section next time}
        end
        else begin                        {Initiate copy mode}
          LastMode:=Mode;
          Copy_Source:=TableEntry;
          gotoxy(0,21);
          write('Use Knob to select color ');
          write('to be copied,then press');
          writeln(' C');
          Mode:=Copy1
        end;
      end;
    end;
  otherwise
  end; {case}
  {---- use delta created above to modify the proper value ----------------}
  case Mode of                      {what am I doing?}
    Hue: begin
        HueVal[TableEntry]:=HueVal[TableEntry]+Delta;         {adjust it}
        if HueVal[TableEntry]>1 then HueVal[TableEntry]:=0;   {Keep it...}
        if HueVal[TableEntry]<0 then HueVal[TableEntry]:=1;   {...in limits}
      end;
    Sat: begin
        SatVal[TableEntry]:=SatVal[TableEntry]+Delta;         {adjust it}
        if SatVal[TableEntry]>1 then SatVal[TableEntry]:=1;   {Keep it...}
        if SatVal[TableEntry]<0 then SatVal[TableEntry]:=0;   {...in limits}
      end;
    Lum: begin
        LumVal[TableEntry]:=LumVal[TableEntry]+Delta;         {adjust it}
        if LumVal[TableEntry]>1 then LumVal[TableEntry]:=1;   {Keep it...}
        if LumVal[TableEntry]<0 then LumVal[TableEntry]:=0;   {...in limits}
      end;
    Table,Copy1,Copy2: begin
      if Delta<>0 then begin
        if Delta>0 then TableEntry:=TableEntry+1      { \  Adjust     }
        else TableEntry:=TableEntry-1;                { /  the value  }
        if TableEntry>15 then TableEntry:=15;         { \  Keep it    }
        if TableEntry<0 then TableEntry:=0;           { /  in limits  }
        UpdateCursor(TableEntry);            {indicate new entry}
      end;
    end;
 end;  {case}
```

```
    set_color_table(TableEntry,                        { \                    }
                    HueVal[TableEntry],                {  \  Modify the       }
                    SatVal[TableEntry],                {  /  color map.       }
                    LumVal[TableEntry]);               { /                    }
    if TableEntry=0 then begin {Background color}
      set_color_model(1);                              {RGB}
      inq_color_table(0,RedBack,GreenBack,BlueBack);        {get RGB values}
      BackSum:=0;                                      { \  Calculate the     }
      if RedBack<0.5 then BackSum:=4;                  {  \ background color  }
      if GreenBack<0.5 then BackSum:=BackSum+2;        {  / in order to make  }
      if BlueBack<0.5 then BackSum:=BackSum+1;         { /  contrasting text. }
      if OldBackSum<>BackSum then begin {Color change}
        case BackSum of
          0:  LabelColor:=FunnyChar[Black];            { \                    }
          1:  LabelColor:=FunnyChar[Blue];             {  \                   }
          2:  LabelColor:=FunnyChar[Green];            {   \  Translate the   }
          3:  LabelColor:=FunnyChar[Cyan];             {    \ RGB background  }
          4:  LabelColor:=FunnyChar[Red];              {    / sum to a        }
          5:  LabelColor:=FunnyChar[Magenta];          {   /  complementary   }
          6:  LabelColor:=FunnyChar[Yellow];           {  /   text color.     }
          7:  LabelColor:=FunnyChar[White];            { /                    }
        end; {case BacksSum of}
        MenuLine;                                      {print the menu line}
        OldBackSum:=BackSum;                           {store for future comparisons}
        set_color_table(1,1-RedBack,                   { \  Make pen one      }
                          1-GreenBack,                 {  > complementary,    }
                          1-BlueBack);                 { /  too.              }
      end; {if}
      set_color_model(2);                              {HSL}
    end; {if TableEntry=0}
    DisplayStuff;                                      {update alpha information}
  until Done;                                          {until user pushes [Q]}
  writeln(FunnyChar[Green],chr(128));                  {restore text screen to normal}
  {---- Report all this good stuff ----------------------------------------}
  Int_A:=0;
  output_esc(Display_Cont,1,0,Int_A,Real_A,Error_Num);
  set_color_model(1);                                  {RGB}
  for I:=0 to 15 do inq_color_table(I,RedVal[I],           { \  Get the RGB  }
                                      GreenVal[I],         {  > definition   }
                                      BlueVal[I]);         { /  of the color }
  writeln('Table');
  write('Index    Hue    Sat    Lum');
  writeln('             Red   Green   Blue');
  for I:=0 to 15 do begin                              { \                    }
    write(I:3,'      ');                               {  \                   }
    write(HueVal[I]:3:2,'   ');                        {   \   Write the color }
    write(SatVal[I]:3:2,'   ');                        {    \  map entries as  }
    write(LumVal[I]:3:2,'          ');                 {     > both HSL and    }
    write(RedVal[I]:3:2,'   ');                        {    /  RGB numbers.    }
    write(GreenVal[I]:3:2,'   ');                      {   /                   }
    writeln(BlueVal[I]:3:2);                           {  /                    }
  end;                                                 { /                     }
  display_term;                                        {deactivate the display}
end;
graphics_term;                                         {terminate the graphics system}
end.                                                   {Main Program}
```

CsizeProg

```
program CsizeProg(output);
import dgl_lib, dgl_inq;                    {get graphics routines}
const
  Crt=                  3;                  {address of internal CRT}
  Control=              0;                  {device control; 0 for CRT}
var
  Error:                integer;            {variable for initialization outcome}
  I, J:                 integer;            {utility variables}
  Strng:                string[10];         {temporary holding place for strings}
$include 'DGLPRG:ConvVtoW'$                 {virtual-to-world conversion}
$page$  {**********************************************************************}
procedure CharSize(Height, AspectRatio: real);
{---------------------------------------------------------------------------}
{    This procedure defines character cell size and the puts the Width and   }
{  Height values into global variables for later use.  The arguments passed  }
{  in are the height of the character cell in VIRTUAL coordinates, and the   }
{  aspect ratio of the character cell.  The values for the window limits     }
{  may be anything; they are taken into account and do not affect the size   }
{  of the characters, since they are defined in virtual coordinates.  This   }
{  procedure, along with Lorg and Ldir, define global variables for use by   }
{  Glabel.                                                                   }
{---------------------------------------------------------------------------}
var
  Width:        real;                       {temporary spot for width}
  X0, Y0:       real;                       {0,0 (virtual) in world}
  X1, Y1:       real;                       {1,1 (virtual) in world}
begin                                       {body of procedure "CharSize"}
ConvertVirtualToWorld(0,0,X0,Y0);           {convert 0,0 in virtual to world}
ConvertVirtualToWorld(1,1,X1,Y1);           {convert 1,1 in virtual to world}
Height:=Height*(Y1-Y0);                     {convert height in virtual to world}
Width:=Height*AspectRatio*(X1-X0)/(Y1-Y0);  {convert width in virtual to world}
set_char_size(Width,Height);                {invoke the parameters}
end;                                        {procedure "CharSize"}
$page$  {*********************************************************************}
begin                                       {body of program "CsizeProg"}
graphics_init;                              {initialize the graphics system}
display_init(Crt,Control,Error);            {which output device?}
if Error=0 then begin                       {output device initialization OK?}
  set_aspect(511,389);                      {use the whole screen}
  set_window(1,2,100,0);                    {scale the window for the data}
  for I:=1 to 6 do begin                    {six different character sizes}
    CharSize(I*I*0.01,0.6);                 {install character size}
    move(1,I*I*I*0.4+I);                    {move to a appropriate place}
    strwrite(Strng,1,J,I*I:0);              {convert number to string}
    gtext(Strng+'%');                       {label the string}
  end;  {for i}
end;  {Error=0?}
graphics_term;                              {terminate the graphics package}
end.                                        {program "CsizeProg"}
```

DataPoint

```
$page$  {*********************************************************************}
function DataPoint(I: integer): real;   {function that returns the y-values}
{---------------------------------------------------------------------}
{    This function returns one of the one hundred values in the structured   }
{  constant "Voltages" every time it is called.  This function is called by  }
{  the "Progressive Example" programs in the graphics techniques chapters.   }
{---------------------------------------------------------------------}
type
  VoltsType=    array [1..100] of real;
const
  Voltages=     VoltsType[0.1610, 0.1625, 0.1625, 0.1628, 0.1636,
                          0.1631, 0.1627, 0.1608, 0.1610, 0.1606,
                          0.1607, 0.1617, 0.1614, 0.1626, 0.1634,
                          0.1640, 0.1656, 0.1660, 0.1644, 0.1651,
                          0.1635, 0.1641, 0.1628, 0.1619, 0.1630,
                          0.1624, 0.1627, 0.1644, 0.1644, 0.1657,
                          0.1660, 0.1670, 0.1672, 0.1666, 0.1658,
                          0.1662, 0.1646, 0.1633, 0.1634, 0.1636,
                          0.1645, 0.1652, 0.1656, 0.1677, 0.1689,
                          0.1680, 0.1696, 0.1680, 0.1674, 0.1677,
                          0.1669, 0.1655, 0.1665, 0.1662, 0.1667,
                          0.1668, 0.1681, 0.1688, 0.1687, 0.1707,
                          0.1716, 0.1716, 0.1694, 0.1698, 0.1683,
                          0.1683, 0.1671, 0.1681, 0.1683, 0.1684,
                          0.1681, 0.1698, 0.1705, 0.1723, 0.1730,
                          0.1734, 0.1714, 0.1722, 0.1716, 0.1696,
                          0.1702, 0.1699, 0.1684, 0.1706, 0.1696,
                          0.1715, 0.1730, 0.1737, 0.1739, 0.1751,
                          0.1732, 0.1747, 0.1729, 0.1717, 0.1710,
                          0.1707, 0.1706, 0.1709, 0.1713, 0.1720];
begin                                   {body of function "DataPoint"}
DataPoint:=Voltages[I];                 {assign it to the function name}
end;    {function "DataPoint"}          {return}
```

DrawMdPrg

```
program DrawMdPrg(output);              {program name same as file name}
import dgl_lib;                         {access the necessary procedures}
const
  Polygons=     100;                    {how many polygons?}
  Sides=        3;                      {how many sides apiece?}
  crt=          3;                      {device address of graphics raster}
  control=      0;                      {device control word; ignored for CRT}
type
  short_int=    -32768..32767;          {16-bit integer}
  DrawingModeType=      (Dominant,Erase,Complement);
  DisplayStates=(Off,On);
var
  X:            array [0..Polygons-1,1..Sides] of short_int;
  Y:            array [0..Polygons-1,1..Sides] of short_int;
  Dx, Dy:       array [1..Sides] of short_int;
  Poly, Side:   short_int;              {loop control variables}
  DrawMode:     DrawingModeType;
  Temp:         short_int;              {temporary holding area}
  New,Previous: short_int;              {for efficient use of arrays}
  seed:         integer;                {random number seed}
  error:        integer;                {display_init return variable; 0 = ok}
```

```
$page$  {*******************************************************************}
procedure Alpha(State: DisplayStates);
{-------------------------------------------------------------------------}
{  This procedure turns the alpha raster on or off.                       }
{-------------------------------------------------------------------------}
const
  AlphaRaster=  1051;                 {mnemonic better than magic number}
var
  AlphaOn:      array [1..1] of integer;       { \   This is all stuff that  }
  Rarray:       array [1..1] of real;          {  >  is needed by the        }
  Error:        integer;                       { /   "output_esc" procedure. }
begin                                 {procedure "Alpha"}
if State=On then AlphaOn[1]:=1
else AlphaOn[1]:=0;
output_esc(AlphaRaster,1,0,AlphaOn,Rarray,Error);
if Error<>0 then writeln('Error ',Error:0,' in procedure "Alpha".');
end;                                  {procedure "Alpha"}
$page$  {*******************************************************************}
procedure DrawingMode(Mode: DrawingModeType);
{-------------------------------------------------------------------------}
{    This procedure selects drawing modes for a monochromatic CRT.        }
{-------------------------------------------------------------------------}
const
  SetDrawingMode=       1052;         {mnemonic better than magic number}
var
  DrawMode:     array [1..1] of integer;       { \   This is all stuff that  }
  Rarray:       array [1..1] of real;          {  >  is needed by the        }
  Error:        integer;                       { /   "output_esc" procedure. }
begin                                 {procedure "DrawingMode"}
case Mode of                    { \                                 }
  Erase:        DrawMode[1]:=2;     {  \   Convert DrawingMode enumerated  }
  Dominant:     DrawMode[1]:=0;     {   >  type into the appropriate       }
  Complement:   DrawMode[1]:=3;     {  /   value for OUTPUT_ESC procedure. }
end;  {case}                    { /                                 }
output_esc(SetDrawingMode,1,0,DrawMode,Rarray,Error);         {set it}
if Error<>0 then writeln('Error ',Error:0,' in procedure "DrawingMode".');
end;                                  {procedure "DrawingMode"}
$page$  {*******************************************************************}
function Rand: short_int;
begin                                 {function "Rand"}
Seed:=((Seed+13579)*39777) mod 10000;   {make new seed}
Rand:=Seed;                           {return current value of seed}
end;                                  {function "Rand"}
$page$  {*******************************************************************}
procedure DefineDeltas;
var
  Side:         short_int;
begin                                 {body of procedure "DefineDeltas"}
for Side:=1 to Sides do begin         {for each vertex}
  Dx[Side]:=Rand mod 5+2;             {magnitude of this dx}
  if Rand>=5000 then Dx[Side]:=-Dx[Side];       {sign of this dx}
  Dy[Side]:=Rand mod 5+2;             {magnitude of this dy}
  if Rand>=5000 then Dy[Side]:=-Dy[Side];       {sign of this dy}
end;  {for side}
end;                                  {body of procedure "DefineDeltas"}
```

```
$page$  {******************************************************************}
begin                                  {body of program "DrawMdPrg"}
DrawMode:=Dominant;                    {specify drawing mode}
Seed:=1173;                            {initialize random number seed}
graphics_init;                         {initialize graphics library}
display_init(crt,control,error);       {initialize CRT}
if error=0 then begin                  {if no error occurred...}
  set_aspect(511,389);                 {use the whole screen}
  set_window(0,511,0,389);             {one user unit=one pixel}
  Alpha(Off);                          {turn off the alpha screen}
  DrawingMode(DrawMode);               {select specified drawing mode}
  for Side:=1 to Sides do begin        {define the first polygon}
    X[0,Side]:=Rand mod 511;           {define X component}
    Y[0,Side]:=Rand mod 389;           {define Y component}
    if Side=1 then                     { \                       }
      int_move(X[0,Side],Y[0,Side])    {  \   Move to the first  }
    else                               {   >  point, and draw to }
      int_line(X[0,Side],Y[0,Side]);   {  /   all the rest.      }
  end;  {for side}
  if Sides>2 then
    int_line(X[0,1],Y[0,1]);           {if simple line, don't close}
  DefineDeltas;                        {define dx and dy for each vertex}
  for Poly:=1 to Polygons-1 do begin   {draw all the polygons}
    for Side:=1 to Sides do begin      {each vertex of each polygon}
      Temp:=X[Poly-1,Side]+Dx[Side];   {avoid recalculation}
      if Temp>511 then                 { \                   }
        Dx[Side]:=-Dx[Side]            {  \  Is X off the    }
      else if Temp<0 then              {  /  screen?         }
        Dx[Side]:=-Dx[Side];           { /                   }
      X[Poly,Side]:=X[Poly-1,Side]+Dx[Side];    {calculate next x}
      Temp:=Y[Poly-1,Side]+Dy[Side];   {avoid recalculation}
      if Temp>389 then                 { \                   }
        Dy[Side]:=-Dy[Side]            {  \  Is Y off the    }
      else if Temp<0 then              {  /  screen?         }
        Dy[Side]:=-Dy[Side];           { /                   }
      Y[Poly,Side]:=Y[Poly-1,Side]+Dy[Side];    {calculate next y}
      if Side=1 then int_move(X[Poly,Side],Y[Poly,Side]) {move to first point,}
      else int_line(X[Poly,Side],Y[Poly,Side]);          {draw to all the rest}
    end;  {for side}
    if Sides>2 then                    {if simple line, don't close polygon}
      int_line(X[Poly,1],Y[Poly,1]);
  end;  {for poly}
  New:=0;                              {start re-use at entry 0}
  while true do begin                  {ad infinitum...}
    if New=0 then Previous:=Polygons-1  {start re-using over}
    else Previous:=(Previous+1) mod Polygons;   {re-use next entry}
    if DrawMode=Dominant then DrawMode:=Erase;  { \ If Dominant, toggle state}
    DrawingMode(DrawMode);             {select specified drawing mode}
    for Side:=1 to Sides do begin      {erase the oldest line}
      if Side=1 then                           {  \   Move to the      }
        int_move(X[New,Side],Y[New,Side])      {   \  first point,     }
      else                                     {   /  draw to all the  }
        int_line(X[New,Side],Y[New,Side]);     {  /   rest.            }
    end;  {for Side}
    if Sides>2 then                    {if simple line, don't close polygon}
      int_line(X[New,1],Y[New,1]);
    if DrawMode=Erase then DrawMode:=Dominant;  { \ If Erase, toggle state}
    DrawingMode(DrawMode);             {select specified drawing mode}
```

```
    for Side:=1 to Sides do begin                              {\                   }
      Temp:=X[Previous,Side]+Dx[Side];                         { \                  }
      if Temp>511 then                                         {  \                 }
        Dx[Side]:=-Dx[Side]                                    {   \                }
      else if Temp<0 then                                      {    \               }
        Dx[Side]:=-Dx[Side];                                   {     \              }
      X[New,Side]:=X[Previous,Side]+Dx[Side];                  {      \   Draw the  }
      Temp:=Y[Previous,Side]+Dy[Side];                         {       \  new line  }
      if Temp>389 then                                         {        \ the same  }
        Dy[Side]:=-Dy[Side]                                    {        / way as    }
      else if Temp<0 then                                      {       /  before.   }
        Dy[Side]:=-Dy[Side];                                   {      /             }
      Y[New,Side]:=Y[Previous,Side]+Dy[Side];                  {     /              }
      if Side=1 then int_move(X[New,Side],Y[New,Side])         {    /               }
      else int_line(X[New,Side],Y[New,Side]);                  {   /                }
    end;  {for side}                                           {  /                 }
    if Sides>2 then                                            { /                  }
      int_line(X[New,1],Y[New,1]);                             {/                   }
    New:=(New+1) mod Polygons;          {next one to re-use}
  end;  {while}
end;  {error=0?}                        {end of conditional code}
graphics_term;                          {terminate graphics library}
end.  {program "DrawMdPrg"}             {end of program}
```

FillProg

```
program FillProg(output);               {program name same as file name}
import
  dgl_lib,dgl_types,dgl_poly,dgl_inq;   {access the necessary procedures}
const
  MaxPoints=          27;               {number of points in arrays}
  Crt=                3;                {device address of graphics raster}
  Control=            0;                {device control word; ignored for CRT}
type
  Reals=       array [1..MaxPoints] of real;   {to contain X and Y values}
  Word=        -32768..32767;                  {16-bit word}
  Integers=    array [1..Maxpoints] of Word;      {to contain op. selectors}
const
  Xvalues=     Reals[ 1.5, 2.5, 2.5, 1.5,-1.5,-2.5,-2.5,-1.5,  {Octagon}
                     -2.5, 2.5, 2.5,-2.5,-2.5,                 {Box}
                     -2.5,-4.5,-2.5,-5.0,-4.0,                 {Left leg}
                      2.5, 4.5, 2.5, 5.0, 4.0,                 {Right leg}
                     -0.5,-1.0, 1.0, 0.5];                     {Nozzle}
  Yvalues=     Reals[ 1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0,  {Octagon}
                      1.0, 1.0,-2.0,-2.0, 1.0,                 {Box}
                     -2.0,-4.0, 0.0,-4.0,-4.0,                 {Left leg}
                     -2.0,-4.0, 0.0,-4.0,-4.0,                 {Right leg}
                     -2.0,-3.0,-3.0,-2.0];                     {Nozzle}
  OpCodes=     Integers[2,1,1,1,1,1,1,1,                       {Octagon}
                        2,1,1,1,1,                             {Box}
                        2,1,1,2,1,                             {Right leg}
                        2,1,1,2,1,                             {Left leg}
                        2,1,1,1];                              {Nozzle}
var
  Error:              integer;          {display_init return variable; 0 = ok}
  I:                  integer;          {loop control variable}
  LemX, LemY:         Reals;            {so we can pass it to "polygon"}
  OpSelectors:        Integers;         {ditto}
  Points:             integer;          {ditto}
```

```
$page$  {*******************************************************************}
begin                                     {body of program "FillProg"}
LemX:=Xvalues;                            { \  Put into variable array so  }
LemY:=Yvalues;                            {  > it can be passed by         }
OpSelectors:=OpCodes;                     { /  reference into the DGL proc.}
Points:=MaxPoints;                        {put constant into an array variable}
graphics_init;                            {initialize graphics library}
display_init(Crt,Control,Error);          {initialize CRT}
if Error=0 then begin                     {if no error occurred...}
  set_aspect(511,389);                    {use the whole screen}
  set_window(-7.5,18.5,-10,10);           {invoke isotropic units}
  set_pgn_style(14);                      {crosshatched fill}
  polygon(Points,LemX,LemY,OpSelectors);        {draw the lines}
  set_window(-18.5,7,-10,10);            {invoke isotropic units}
  set_pgn_table(14,0.51,0,1);             {set the "do a fill" flag}
  set_color_table(1,0.125,0.125,0.125); {specify 12.5% gray scale}
  set_pgn_color(1);                       {use specified "color"}
  polygon_dev_dep(Points,LemX,LemY,OpSelectors);        {draw the lines}
end;  {Error=0?}                          {end of conditional code}
graphics_term;                            {terminate graphics library}
end.  {program "FillProg"}                {end of program}
```

FillGraph

```
program FillGraph(output);
import dgl_lib, dgl_types, dgl_poly;
const
  CrtAddr=              3;
  ControlWord=          0;
type
  RDataType=            array [0..12] of real;
  WDataType=            array [0..12] of -32768..32767;
const
  Xvalues=              RDataType[0,1,2,3,4,5,6,7,8,9,10,10,0];
  Yvalues=              RDataType[2,4,3,6,5,5,3,7,5,6,8,0,0];
  OperationSelectors=   WDataType[2,1,1,1,1,1,1,1,1,1,1,1,1];
var
  ErrorReturn:          integer;
  X, Y:                 RDataType;
  OpSel:                WDataType;
$page$  {*******************************************************************}
begin                                     {program "FillGraph"}
graphics_init;
display_init(CrtAddr,ControlWord,ErrorReturn);
if ErrorReturn=0 then begin
  set_aspect(511,389);
  set_window(0,10,0,10);
  move(0,0);  line(0,10);  line(10,10);  line(10,0);  line(0,0);
  X:=Xvalues;     Y:=Yvalues;     OpSel:=OperationSelectors;
  set_pgn_table(1,0.333,17.34,1);
  set_pgn_style(1);
  polygon(13,X,Y,OpSel);
end;  {ErrorReturn=0?}
graphics_term;
end.                                      {program "FillGraph"}
```

GstorProg

```
$sysprog on$                                {so we can define array addresses}
program GstorProg(Keyboard,output);         {program name same as file name}
label 1;
import dgl_lib, dgl_inq;                    {access the necessary procedures}
const
  Crt=          3;                          {device address of graphics raster}
  Control=      0;                          {device control word; ignored for CRT}
  GRasterAddr=  hex('530000');              {address of graphics memory}
  GRasterSize=  6240;                       {32-bit integers in graphics raster}
  Ratio=        1.31362467886;              {aspect ratio of the Model 36 CRT}
type
  GRasterType=  array [1..GRasterSize] of integer;
  HJustifyType= (Left,HCentered,Right);
  VJustifyType= (Bottom,VCentered,Top);
  DisplayStates=(Off,On);
  AngType=      (Deg,Rad,Grad);             {used by procedure Ldir}
  RoundType=    (Up, Down, Near);           {used by function Round2}
  Str255=       string[255];                {used by procedure Glabel}
var
  Error:                 integer;           {display_init return variable; 0 = ok}
  Decade, Units:         integer;           {for logarithmic X-axis}
  X,Dx:                  real;              {x-axis variables}
  Xmin,Xmax,Xrange:      integer;           {more x-axis variables}
  Y,Dy:                  real;              {y-axis variables}
  Ymin,Ymax,Yrange:      integer;           {more y-axis variables}
  I:                     integer;           {utility variable}
  Strng:                 Str255;            {another utility variable}
  Character:             char;              {and yet another}
  Temperature:           real;              {need a larger range than an integer}
  OldX, OldY:            real;              {last point drawn to}
  GRaster[GRasterAddr]:  GRasterType;       {actual graphics raster}
  Screen:                GRasterType;       {user's screen image}
  Keyboard:              text;              {allow GETs from the keyboard}
  CharWidth,CharHeight:  real;              { \  These are global variables      }
  HJustification:        HJustifyType;      {  \ used by the CharSize/           }
  VJustification:        VJustifyType;      {  / LabelDirection/LabelJustify/    }
  CharTheta:             real;              { /  Glabel series of procedures.    }
  ClipXmin, ClipXmax:    real;              {soft clip limits in x}
  ClipYmin, ClipYmax:    real;              {soft clip limits in y}
$include 'DGLPRG:ConvVtoW'$
$page$  {******************************************************************}
procedure CharSize(Height, AspectRatio: real);
{--------------------------------------------------------------------------}
{    This procedure defines character cell size and the puts the Width and  }
{  Height values into global variables for later use.  The arguments passed }
{  in are the height of the character cell in VIRTUAL coordinates, and the  }
{  aspect ratio of the character cell.  The values for the window limits    }
{  may be anything; they are taken into account and do not affect the size  }
{  of the characters, since they are defined in virtual coordinates.  This   }
{  procedure, along with Lorg and Ldir, define global variables for use by   }
{  Glabel.                                                                   }
{--------------------------------------------------------------------------}
```

```
var
  Width:        real;                   {temporary spot for width}
  X0, Y0:       real;                   {0,0 (virtual) in world}
  X1, Y1:       real;                   {1,1 (virtual) in world}
begin                                   {body of procedure "CharSize"}
ConvertVirtualToWorld(0,0,X0,Y0);       {convert 0,0 in virtual to world}
ConvertVirtualToWorld(1,1,X1,Y1);       {convert 1,1 in virtual to world}
Height:=Height*(Y1-Y0);                 {convert height in virtual to world}
Width:=Height*AspectRatio*(X1-X0)/(Y1-Y0);  {convert width in virtual to world}
set_char_size(Width,Height);            {invoke the parameters}
end;                                    {procedure "CharSize"}
$page$  {*******************************************************************}
procedure LabelDirection(Direction: real; Units: AngType);
{--------------------------------------------------------------------------}
{    This procedure is used in conjunction with LabelOrigin, CharSize and   }
{  Glabel.  It sets the labelling direction to be used, and places the      }
{  direction into a global variable so Glabel can use it.                   }
{--------------------------------------------------------------------------}
const
  Deg_per_rad=  57.2957795131;  {180/pi: for converting degrees to radians}
  Grad_per_rad= 63.6619772368;  {200/pi: for converting grads to radians}
begin                                   {procedure "LabelDirection"}
case Units of
  Deg:  Direction:=Direction/Deg_per_rad;   {degrees to radians}
  Rad:  ;                                   {correct units already}
  Grad: Direction:=Direction/Grad_per_rad;  {grads to radians}
end;  {case}
CharTheta:=Direction;                   {put into a global variable}
set_text_rot(cos(CharTheta),sin(CharTheta));    {invoke the new text direction}
end;                                    {procedure "LabelDirection"}
$page$  {*******************************************************************}
procedure LabelJustify(HJust: HJustifyType;  VJust: VJustifyType);
{--------------------------------------------------------------------------}
{    This procedure is used in conjunction with procedures CharSize,        }
{  LabelDirection, and Glabel.   This just puts a value into global         }
{  variables which will be subsequently used by Glabel.                     }
{--------------------------------------------------------------------------}
begin                                   {procedure "LabelJustify"}
HJustification:=HJust;
VJustification:=VJust;
end;                                    {procedure "LabelJustify"}
$page$  {*******************************************************************}
function Atan(Y, X: real): real;
{--------------------------------------------------------------------------}
{    This function returns the value of the arctangent of Y/X, placing it  }
{  in the correct quadrant.  If Y and X  are both zero, the result is zero. }
{--------------------------------------------------------------------------}
const
  Pi=           3.14159265359;          {pi}
begin                                   {function "Atan"}
if X=0.0 then Atan:=(Pi/2+Pi*ord(Y<0.0))*ord(Y<>0.0)
else Atan:=arctan(Y/X)+Pi*ord(X<0.0)+2*Pi*ord((X>0.0) and (Y<0.0));
end;                                    {function "Atan"}
```

```
$page$ {*******************************************************************}
procedure Glabel(Text: Str255);
{-----------------------------------------------------------------------}
{    This procedure labels a string of text at the current pen position.     }
{  It takes into account the current label direction (set by procedure       }
{  "LabelDirection", the current character size (set by procedure            }
{  "CharSize"), and the current label justification (set by procedure        }
{  "LabelJustify").                                                           }
{-----------------------------------------------------------------------}
const
  CharSizeCode=          250;            {mnemonic better than magic number}
  CurrentPosition=       259;            {ditto}
type
  Positions=             (X,Y);
  PositionType=          array [Positions] of real;
  CharAttributes=        (Width,Heighth);
  CharAttrType=          array [CharAttributes] of real;
var
  Chars:                 integer;
  Charsize:              CharAttrType;
  Len,Height:            real;           {length and height of character string}
  Dx,Dy:                 real;
  R,Theta:               real;           {for rectangular-to-polar conversion}
  Pac:                   packed array [1..1] of char;    {  \  These are the   }
  Iarray:                array [1..1] of integer;        {   \ sundry items    }
  Position:              PositionType;                   {   / needed for the  }
  Error:                 integer;                        {  /  call to "inq_ws"}
begin                                    {procedure "Glabel"}
inq_ws(CharSizeCode,0,0,2,Pac,Iarray,Charsize,Error);     {get pen position}
if Error<>0 then writeln('Error',Error:0,' in "Glabel".');
Chars:=strlen(text);
Len:=Charsize[Width]*(7*Chars+2*(Chars-1))/9;  {length minus inter-char gap}
Height:=Charsize[Heighth]*8/15;                {height minus inter-line gap}
Dx:=Len*(-ord(HJustification)/2);
Dy:=Height*(-ord(VJustification)/2);
R:=sqrt(Dx*Dx+Dy*Dy);                    { \  Convert to polar coordinates so  }
Theta:=Atan(Dy,Dx);                      { /  rotation is easy.                }
Theta:=Theta+CharTheta;                  {add the LabelDirection angle}
Dx:=R*cos(Theta);                        { \  Convert R and the new Theta back }
Dy:=R*sin(Theta);                        { /  to rectangular coodinates.       }
inq_ws(CurrentPosition,0,0,2,Pac,Iarray,Position,Error);     {get pen position}
if Error=0 then begin
  move(Position[X]+Dx,Position[Y]+Dy);   {move to the new starting point}
  gtext(text);
end  {Error=0?}
else writeln('Error',Error:0,' in "Glabel".');
end;                                     {procedure "Glabel"}
$page$ {*******************************************************************}
procedure MainTitle(X, Y: real; Title: Str255);
{-----------------------------------------------------------------------}
{    This procedure writes a large label for the main title of a plot.       }
{-----------------------------------------------------------------------}
begin                                    {procedure "MainTitle"}
CharSize(0.06,0.6);
LabelDirection(0,Deg);
LabelJustify(HCentered,Top);
move(X,Y);
Glabel(Title);
end;                                     {procedure "MainTitle"}
```

```
$page$  {*******************************************************************}
procedure XAxisTitle(X, Y: real; Title: Str255);
{-------------------------------------------------------------------------}
{    This procedure writes a small label for the X-axis title of a plot.     }
{-------------------------------------------------------------------------}
begin                                   {procedure "XaxisTitle"}
CharSize(0.04,0.6);
LabelDirection(0,Deg);
LabelJustify(HCentered,Bottom);
move(X,Y);
Glabel(Title);
end;                                    {procedure "XaxisTitle"}
$page$  {*******************************************************************}
procedure YaxisTitle(X,Y: real; Title: Str255);
{-------------------------------------------------------------------------}
{    This procedure writes a large label for the Y-axis title of a plot.     }
{-------------------------------------------------------------------------}
begin                                   {procedure "YaxisTitle"}
CharSize(0.04,0.6);
LabelDirection(90,Deg);
LabelJustify(HCentered,Top);
move(X,Y);
Glabel(Title);
end;                                    {procedure "YaxisTitle"}
$page$  {*******************************************************************}
procedure ClipLimit(Xmin, Xmax, Ymin, Ymax: real);
{-------------------------------------------------------------------------}
{  This procedure defines the four global variables which specify where the  }
{  soft clip limits are.                                                     }
{-------------------------------------------------------------------------}
begin
if Xmin<Xmax then begin                 { \                           }
  ClipXmin:=Xmin;                       {  \    Force the minimum soft  }
  ClipXmax:=Xmax;                       {   \   clip limit in X to be   }
end                                     {    \  the smaller of the two  }
else begin                              {    /  X values passed into    }
  ClipXmin:=Xmax;                       {   /   the procedure.          }
  ClipXmax:=Xmin;                       {  /                            }
end;                                    { /                             }
if Ymin<Ymax then begin                 { \                           }
  ClipYmin:=Ymin;                       {  \    Force the minimum soft  }
  ClipYmax:=Ymax;                       {   \   clip limit in Y to be   }
end                                     {    \  the smaller of the two  }
else begin                              {    /  Y values passed into    }
  ClipYmin:=Ymax;                       {   /   the procedure.          }
  ClipYmax:=Ymin;                       {  /                            }
end;                                    { /                             }
end;
```

```
$page$  {*******************************************************************}
procedure ClipDraw(X1, Y1, X2, Y2: real);
{-----------------------------------------------------------------------------}
{    This procedure takes the endpoints of a line, and clips it.  The soft    }
{  clip limits are the real global variables ClipXmin, ClipXmax, ClipYmin,    }
{  and ClipYmax.  These may be defined through the procedure ClipLimit.       }
{-----------------------------------------------------------------------------}
label
  1;
type
  Edges=         (Left,Right,Top,Bottom);       {possible edges to cross}
  OutOfBounds=  set of Edges;                   {set of edges crossed}
var
  Out,Out1,Out2:OutOfBounds;
  X, Y:          real;
{-----------------------------------------------------------------------------}
procedure Code(X, Y: real; var Out: OutOfBounds);
begin                                   {nested procedure "Code"}
Out:=[];                                {null set}
if x<ClipXmin then Out:=[left]            {off left edge?}
else if x>ClipXmax then Out:=[right];     {off right edge?}
if y<ClipYmin then Out:=Out+[bottom]      {off the bottom?}
else if y>ClipYmax then Out:=Out+[top];   {off the top?}
end;                                    {nested procedure "Code"}
{-----------------------------------------------------------------------------}
begin                                   {body of procedure "ClipDraw"}
Code(X1,Y1,Out1);                       {figure status of point 1}
Code(X2,Y2,Out2);                       {figure status of point 2}
while (Out1<>[]) or (Out2<>[]) do begin {loop while either point out of range}
  if (Out1*Out2)<>[] then goto 1;       {if intersection non-null, no line}
  if Out1<>[] then Out:=Out1
    else Out:=Out2;                     {Out is the non-empty one}
  if left in Out then begin             {it crosses the left edge}
    y:=Y1+(Y2-Y1)*(ClipXmin-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmin;                        {new x is left edge}
  end  {left in Out?}
  else if right in Out then begin       {it crosses right edge}
    y:=Y1+(Y2-Y1)*(ClipXmax-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmax;                        {new x is right edge}
  end  {right in Out?}
  else if bottom in Out then begin      {it crosses the bottom edge}
    x:=X1+(X2-X1)*(ClipYmin-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmin;                        {new y is bottom edge}
  end  {bottom in Out?}
  else if top in Out then begin         {it crosses the top edge}
    x:=X1+(X2-X1)*(ClipYmax-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmax;                        {new y is top edge}
  end;  {top in Out?}
  if Out=Out1 then begin
    X1:=x;   Y1:=y;   Code(x,y,Out1);   {redefine first end point}
  end  {Out=Out1?}
  else begin
    X2:=x;   Y2:=y;   Code(x,y,Out2);   {redefine second end point}
  end;  {else begin}
end;  {while}
move(x1,y1);                            {if we get to this point, the line...}
line(x2,y2);                            {...is completely visible, so draw it}
1:   end;  {procedure "ClipDraw"}       {return}
```

```
$page$ {*******************************************************************}
function Round2(N, M: real; Mode: RoundType): real;
{-------------------------------------------------------------------------}
{  This function rounds "N" to the nearest "M", according to "Mode".  This  }
{  function works only when the argument is in the range of MININT..MAXINT.  }
{-------------------------------------------------------------------------}
const
  epsilon=      1E-10;                  {roundoff error fudge factor}
var
  Rounded:      real;                   {temporary holding area}
  Negative:     boolean;                {flag: "It is negative?"}
begin                                   {body of "Round2"}
Negative:=(N<0.0);                      {is the number negative?}
if Negative then begin
  N:=abs(N);                            {work with a positive number}
  if Mode=Up then Mode:=Down            {if number is negative, ...}
  else if Mode=Down then Mode:=Up;      {...reverse up and down}
end;
case Mode of                            {should we round the number...}
  Down: Rounded:=trunc(N/M)*M;          {...left on the number line?}
  Up:   begin
          Rounded:=N/M;                 {...right on the number line?}
          if abs(Rounded-round(Rounded))>epsilon then
            Rounded:=(trunc(Rounded)+1.0)*M
          else
            Rounded:=trunc(Rounded)*M;
        end;
  Near: Rounded:=trunc(N/M+M*0.5)*M;    {...to the nearest multiple?}
end;  {case}
if Negative then Rounded:=-Rounded;     {reinstate the sign}
Round2:=Rounded;                        {assign to function name}
end;                                    {function "Round2"}
$page$ {*******************************************************************}
procedure YaxisClip(Spacing, Location: real; Major: integer;
  Majsize, Minsize: real);
{-------------------------------------------------------------------------}
{  This procedure draws an Y-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                      }
{    Spacing:   The distance between tick marks on the axis.                }
{    Location:  The X-value of the Y-axis.                                   }
{    Major:     The number of tick marks to ge before drawing a major tick  }
{               mark.  If Major=5, every fifth tick mark will be major.     }
{    Majsize:   The length, in world units, of the major tick marks.        }
{    Minsize:   The length, in world units, of the minor tick marks.        }
{-------------------------------------------------------------------------}
var
  Y:            real;
  SemiMinsize:  real;
  SemiMajsize:  real;
  Counter:      integer;                {Keeps track of when to do major ticks}
begin                                   {body of procedure "YaxisClip"}
SemiMajsize:=Majsize*0.5;
SemiMinsize:=Minsize*0.5;
Counter:=0;                             {start with a major tick}
ClipDraw(Location,ClipYmin,Location,ClipYmax);
Y:=Round2(ClipYmin,Spacing*Major,Down); {round to next lower major}
```

```
while Y<=ClipYmax do begin
  if Counter=0 then
    ClipDraw(Location-SemiMajsize,Y,Location+SemiMajsize,Y)
  else
    ClipDraw(Location-SemiMinsize,Y,Location+SemiMinsize,Y);
  Counter:=(Counter+1) mod Major;
  Y:=Y+Spacing;
end;  {while}
end;                                {procedure "YaxisClip"}
$page$  {*****************************************************************}
procedure Alpha(State: DisplayStates);
{-------------------------------------------------------------------------}
{  This procedure turns the alpha raster on or off.                       }
{-------------------------------------------------------------------------}
const
  AlphaRaster=  1051;               {mnemonic better than magic number}
var
  AlphaOn:      array [1..1] of integer;        { \   This is all stuff that  }
  Rarray:       array [1..1] of real;           {  >  is needed by the        }
  Error:        integer;                        { /   "output_esc" procedure. }
begin                               {procedure "Alpha"}
if State=On then AlphaOn[1]:=1
else AlphaOn[1]:=0;
output_esc(AlphaRaster,1,0,AlphaOn,Rarray,Error);
if Error<>0 then writeln('Error ',Error:0,' in procedure "Alpha".');
end;                                {procedure "Alpha"}
$page$  {*****************************************************************}
function Log10(X: real): real;
{-------------------------------------------------------------------------}
{  This function returns the logarithm to the base ten of a number.       }
{-------------------------------------------------------------------------}
const
  Log_10=       2.30258509299;      {log to the base e of 10}
begin                               {function "Log10"}
Log10:=ln(X)/Log_10;
end;                                {function "Log10"}
$page$  {*****************************************************************}
function XtoY(X, Y: real): real;
{-------------------------------------------------------------------------}
{  This function evaluates X to the Yth power.                            }
{-------------------------------------------------------------------------}
begin                               {function "XtoY"}
XtoY:=exp(Y*ln(X));                 {an logarithmic identity}
end;                                {function "XtoY"}
$page$  {*****************************************************************}
procedure Gload(var Screen: GRasterType);
{-------------------------------------------------------------------------}
{    This procedure loads a user's array into graphics memory.            }
{-------------------------------------------------------------------------}
begin                               {procedure "Gload"}
GRaster:=Screen;                    {copy user array into graphics memory}
end;                                {procedure "Gload"}
```

```
$page$  {*******************************************************************}
procedure Gstore(var Screen: GRasterType);
{-------------------------------------------------------------------------}
{    This procedure stores graphics memory into a user's array.          }
{-------------------------------------------------------------------------}
begin                                   {procedure "Gstore"}
Screen:=GRaster;                        {copy graphics memory into user array}
end;                                    {procedure "Gstore"}
$page$  {*******************************************************************}
function Sign(X: real): integer;
{-------------------------------------------------------------------------}
{  This function returns the sign of a number, i.e., -1 if the number is  }
{  negative, 0 if the number is zero, and +1 if the number is positive.   }
{-------------------------------------------------------------------------}
begin                                   {function "Sign"}
Sign:=ord(X>0.0)-ord(X<0.0);            {<0 -> -1; =0 -> 0; >0 -> 1}
end;                                    {function "Sign"}
$page$  {*******************************************************************}
function Intensity(Wavelength, Temperature: real): real;
begin                                   {function "Intensity"}
Intensity:=37410/XtoY(Wavelength,5)/(exp(14.39/(Wavelength*Temperature))-1);
end;                                    {function "Intensity"}
$page$  {*******************************************************************}
begin                                   {body of program "GstorProg"}
graphics_init;                          {initialize graphics library}
display_init(Crt,Control,Error);        {initialize CRT}
if Error=0 then begin                   {if no error occurred...}
  Alpha(Off);
  set_aspect(Ratio,1);                  {enable entire plotting surface}
  {===== Label the graph =================================================}
  for I:=-3 to 3 do                     {seven iterations}       {\  Write    }
    MainTitle(I*0.002,1,'Blackbody Radiation');                  { \ the four }
  XaxisTitle(0,0.83,'Temperature (K):       ');                  {   >main    }
  YaxisTitle(-1,0,'Intensity of Radiation');                     { / labels.  }
  XaxisTitle(0,-0.92,'Wavelength (microns)');                    {/           }
  set_viewport(0.1,0.98,0.15/Ratio,0.9/Ratio);
                                        {define subset of plotting surface}
  Xmin:=-4;                             {smallest power for wavelength}
  Xmax:=3;                              {largest power for wavelength}
  Xrange:=Xmax-Xmin;                    {distance between X extremes}
  Dx:=0.1;                              {increment of X}
  Ymin:=-5;                             {smallest power for intensity}
  Ymax:=25;                             {largest power for intensity}
  Yrange:=Ymax-Ymin;                    {distance between Y extremes}
  Dy:=1;                                {increment of Y}
  set_window(Xmin,Xmax,Ymin,Ymax);      {scale the window for the data}
  {===== Draw and label logarithmic X-axis grid ==========================}
  for Decade:=Xmin to Xmax do begin     {one decade equals one mantissa cycle}
    for Units:=1 to 1+8*ord(Decade<Xmax) do begin   {do 2-9 if not last cycle}
      X:=Decade+Log10(Units);           {calculate X for screen}
      move(X,Ymin);                     { \   Draw a vertical line for     }
      line(X,Ymax);                     { /   Y-axis at appropriate place  }
    end;  {for units}
  end;  {for decade}
  X:=Xmin;                              {starting place for X-axis labels}
  Strng:='';                            {null out the string}
```

```
  while X<=Xmax do begin                          { \                     }
    LabelJustify(HCentered,Top);                  {  \                    }
    CharSize(0.025,0.6);                          {   \                   }
    move(X,Ymin-Yrange*0.015);                    {    \                  }
    Glabel('10 ');                                {     \  Write the      }
    CharSize(0.015,0.6);                          {      > labels for the }
    LabelJustify(Left,Bottom);                    {     /  X-axis.        }
    move(X+Xrange*0.01,Ymin-Yrange*0.025);        {    /                  }
    strwrite(Strng,1,I,X:2:0);                    {   /                   }
    Glabel(strltrim(Strng));                      {  /                    }
    X:=X+Dx*10;                                   { /                     }
  end;  {while}
  {===== Draw and label logarithmic Y-axis grid ===========================}
  ClipLimit(Xmin,Xmax,Ymin,Ymax);         {define soft clip limits for axes}
  YaxisClip(1,Xmin,1,0.01,0.01);          {plot left Y-axis}
  YaxisClip(1,Xmax,1,0.01,0.01);          {plot right Y-axis}
  Y:=Ymin;                                {starting value for y}
  while Y<=Ymax do begin
    move(Xmin,Y);                         { \  Draw horizontal line for   }
    line(Xmax,Y);                         { /  an X-axis.                 }
    LabelJustify(Right,VCentered);        {label origin: center of right edge}
    CharSize(0.025,0.6);                          { \                     }
    move(Xmin-Xrange*0.03,Y);                     {  \                    }
    Glabel('10');                                 {   \                   }
    CharSize(0.015,0.6);                          {    \  Write Y-axis    }
    LabelJustify(Left,Bottom);                    {     > labels in       }
    move(Xmin-Xrange*0.025,Y+Yrange*0.01);        {    /  exponential     }
    strwrite(Strng,1,I,Y:2:0);                    {   /   form.           }
    Glabel(strltrim(Strng));                      {  /                    }
    Y:=Y+5*Dy;                                    { /                     }
  end;  {while}
  {===== Here is where the action starts ==================================}
  Gstore(Screen);                         {put the image into array}
  CharSize(0.03,0.6);                     {set up charsize for temperature}
  LabelJustify(Left,Bottom);              {set up label origin for temperature}
  while true do begin
    read(Keyboard,Character);             {get character from keyboard}
    if Character='q' then Character:='Q';
    if (Character>='1') and (Character<='9') then begin
      Gload(Screen);                            {load the image}
      I:=ord(Character)-ord('0');               {translate char to number}
      Temperature:=I*XtoY(10,3);                {build temperature}
      move(0,25.6);                             {move to where temp is to be}
      strwrite(Strng,1,I,Temperature:15:0);     {translate to string}
      Glabel(strltrim(Strng));                  {label the temperature}
      OldX:=Xmin;                               {X starting point}
      OldY:=Intensity(XtoY(10,OldX),Temperature);     {Y starting point}
      X:=OldX+Dx;                               {second point for X}
      while X<=Xmax do begin                    {loop through all Xs}
        Y:=Intensity(XtoY(10,X),Temperature);   {calculate current X}
        ClipDraw(OldX,Log10(OldY),X,Log10(Y));{draw line after clipping}
        OldX:=X;    OldY:=Y;                    {save new old values}
        X:=X+2*Dx;                              {speed up the curve}
      end;  {while}
    end
    else if Character='Q' then goto 1
    else write(#G);                             {beep}
  end;  {while true}
end;  {Error=0?}                          {end of conditional code}
1: graphics_term;                         {terminate graphics library}
end.  {program "GstorProg"}               {end of program}
```

IsoProg

```
program IsoProg(input,output,Keyboard);
import dgl_lib, dgl_inq;                    {access the necessary procedures}
const
  Crt=                  3;                  {device address of graphics raster}
  Control=              0;                  {device control word; ignored for CRT}
  Ratio=                1.31362467886;      {aspect ratio of Model 236 screen}
type
  RoundType=            (Up,Down,Near);     {used by function Round2}
var
  Error:                integer;            {display_init return variable; 0 = ok}
  Xmin,Xmax,Ymin,Ymax:  real;               {isotropic units for window}
  Character:            string[1];          {for continue message}
  ClipXmin, ClipXmax:   real;               {soft clip limits in X}
  ClipYmin, ClipYmax:   real;               {soft clip limits in Y}
  Keyboard:             text;               {non-echoing input}
$page$  {*****************************************************************}
procedure Frame;
{-------------------------------------------------------------------------}
{    This procedure draws a frame around the current window limits.      }
{-------------------------------------------------------------------------}
const
  WindowLimits= 450;                        {mnemonic better than magic number}
type
  LimitOrder=   (Xmin, Xmax, Ymin,Ymax);
  LimitType=    array [LimitOrder] of real;
var
  Pac:          packed array [1..1] of char;     { \  These are the sundries  }
  Iarray:       array [1..1] of integer;         {   \ needed by the call to  }
  Window:       LimitType;                       {   / the DGL procedure      }
  Error:        integer;                         { /   "inq_ws".              }
begin                                            {body of procedure "Frame"}
inq_ws(WindowLimits,0,0,4,Pac,Iarray,Window,Error);
if Error=0 then begin
  move(Window[Xmin],Window[Ymin]);               {move to lower left corner}
  line(Window[Xmin],Window[Ymax]);               {draw to upper left corner}
  line(Window[Xmax],Window[Ymax]);               {draw to upper right corner}
  line(Window[Xmax],Window[Ymin]);               {draw to lower right corner}
  line(Window[Xmin],Window[Ymin]);               {draw to lower left corner}
end  {Error=0?}
else writeln('Error ',Error:0,' occurred in "Frame"');
end;  {procedure "Frame"}                   {return}
```

```
$page$ {*****************************************************************}
procedure ClipLimit(Xmin, Xmax, Ymin, Ymax: real);
{-----------------------------------------------------------------------------}
{  This procedure defines the four global variables which specify where the  }
{  soft clip limits are.                                                     }
{-----------------------------------------------------------------------------}
begin                                {procedure "ClipLimit"}
if Xmin<Xmax then begin              { \                         }
  ClipXmin:=Xmin;                    {  \    Force the minimum soft  }
  ClipXmax:=Xmax;                    {   \   clip limit in X to be   }
end                                  {    \  the smaller of the two  }
else begin                           {    /  X values passed into    }
  ClipXmin:=Xmax;                    {   /   the procedure.          }
  ClipXmax:=Xmin;                    {  /                        }
end;                                 { /                         }
if Ymin<Ymax then begin              { \                         }
  ClipYmin:=Ymin;                    {  \    Force the minimum soft  }
  ClipYmax:=Ymax;                    {   \   clip limit in Y to be   }
end                                  {    \  the smaller of the two  }
else begin                           {    /  Y values passed into    }
  ClipYmin:=Ymax;                    {   /   the procedure.          }
  ClipYmax:=Ymin;                    {  /                        }
end;                                 { /                         }
end;                                 {procedure "ClipLimit"}
$page$ {*****************************************************************}
procedure ClipDraw(X1, Y1, X2, Y2: real);
{-----------------------------------------------------------------------------}
{    This procedure takes the endpoints of a line, and clips it.  The soft   }
{  clip limits are the real global variables ClipXmin, ClipXmax, ClipYmin,   }
{  and ClipYmax.  These may be defined through the procedure ClipLimit.      }
{-----------------------------------------------------------------------------}
label
  1;
type
  Edges=        (Left,Right,Top,Bottom);        {possible edges to cross}
  OutOfBounds=  set of Edges;                   {set of edges crossed}
var
  Out,Out1,Out2:OutOfBounds;
  X, Y:         real;
{-----------------------------------------------------------------------------}
procedure Code(X, Y: real; var Out: OutOfBounds);
begin                                {nested procedure "Code"}
Out:=[];                             {null set}
if x<ClipXmin then Out:=[left]         {off left edge?}
else if x>ClipXmax then Out:=[right];  {off right edge?}
if y<ClipYmin then Out:=Out+[bottom]   {off the bottom?}
else if y>ClipYmax then Out:=Out+[top]; {off the top?}
end;                                 {nested procedure "Code"}
{-----------------------------------------------------------------------------}
```

```
begin                                     {body of procedure "ClipDraw"}
Code(X1,Y1,Out1);                         {figure status of point 1}
Code(X2,Y2,Out2);                         {figure status of point 2}
while (Out1<>[]) or (Out2<>[]) do begin {loop while either point out of range}
  if (Out1*Out2)<>[] then goto 1;         {if intersection non-null, no line}
  if Out1<>[] then Out:=Out1
    else Out:=Out2;                       {Out is the non-empty one}
  if left in Out then begin               {it crosses the left edge}
    y:=Y1+(Y2-Y1)*(ClipXmin-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmin;                          {new x is left edge}
  end  {left in Out?}
  else if right in Out then begin         {it crosses right edge}
    y:=Y1+(Y2-Y1)*(ClipXmax-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmax;                          {new x is right edge}
  end  {right in Out?}
  else if bottom in Out then begin        {it crosses the bottom edge}
    x:=X1+(X2-X1)*(ClipYmin-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmin;                          {new y is bottom edge}
  end  {bottom in Out?}
  else if top in Out then begin           {it crosses the top edge}
    x:=X1+(X2-X1)*(ClipYmax-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmax;                          {new y is top edge}
  end;  {top in Out?}
  if Out=Out1 then begin
    X1:=x;   Y1:=y;   Code(x,y,Out1);     {redefine first end point}
  end  {Out=Out1?}
  else begin
    X2:=x;   Y2:=y;   Code(x,y,Out2);     {redefine second end point}
  end;  {else begin}
end;  {while}
move(x1,y1);                              {if we get to this point, the line...}
line(x2,y2);                              {...is completely visible, so draw it}
1:   end;                                 {procedure "ClipDraw"}
$page$  {*******************************************************************}
function Round2(N, M: real; Mode: RoundType): real;
{-----------------------------------------------------------------------------}
{  This function rounds "N" to the nearest "M", according to "Mode".  This   }
{  function works only when the argument is in the range of MININT..MAXINT.  }
{-----------------------------------------------------------------------------}
const
  epsilon=      1E-10;                    {roundoff error fudge factor}
var
  Rounded:      real;                     {temporary holding area}
  Negative:     boolean;                  {flag: "It is negative?"}
begin                                     {body of "Round2"}
Negative:=(N<0.0);                        {is the number negative?}
if Negative then begin
  N:=abs(N);                              {work with a positive number}
  if Mode=Up then Mode:=Down              {if number is negative, ...}
  else if Mode=Down then Mode:=Up;        {...reverse up and down}
end;
```

```
case Mode of                            {should we round the number...}
  Down: Rounded:=trunc(N/M)*M;          {...left on the number line?}
  Up:   begin
          Rounded:=N/M;                 {...right on the number line?}
          if abs(Rounded-round(Rounded))>epsilon then
            Rounded:=(trunc(Rounded)+1.0)*M
          else
            Rounded:=trunc(Rounded)*M;
        end;
  Near: Rounded:=trunc(N/M+M*0.5)*M;    {...to the nearest multiple?}
end;  {case}
if Negative then Rounded:=-Rounded;     {reinstate the sign}
Round2:=Rounded;                        {assign to function name}
end;                                    {function "Round2"}
$page$  {*************************************************************}
procedure Grid(Xspacing,Yspacing,XlocY,YlocX: real; Xmajor, Ymajor: integer;
  Xminsize, Yminsize: real);
{------------------------------------------------------------------------}
{  This procedure draws a grid on the plotting surface, with user-definable  }
{  minor tick size.  Parameters are as follows:                          }
{    Xspacing:  The distance between tick marks on the X axis.           }
{    Yspacing:  The distance between tick marks on the Y axis.           }
{    XlocY:     The X-value of the Y-axis.                               }
{    YlocX:     The Y-value of the X-axis.                               }
{    Xmajor,    The number of tick marks to ge before drawing a major tick  }
{    Ymajor:    mark.  If Major=5, every fifth tick mark will be major.  }
{    Xminsize:  The length, in world units, of the X minor tick marks.   }
{    Yminsize:  The length, in world units, of the Y minor tick marks.   }
{------------------------------------------------------------------------}
var
  X, Y:          real;
  Xstart,Ystart:real;
  XsemiMinsize: real;
  YsemiMinsize: real;
  Counter:      integer;
begin                                   {body of procedure "Grid"}
XsemiMinsize:=Xminsize*0.5;
YsemiMinsize:=Yminsize*0.5;
Xstart:=Round2(ClipXmin,Xspacing*Xmajor,Down);  {round to next lower major}
Ystart:=Round2(ClipYmin,Yspacing*Ymajor,Down);  {round to next lower major}
{===== Draw vertical major ticks ==========================================}
X:=Xstart;
while X<=ClipXmax do begin
  ClipDraw(X,ClipYmin,X,ClipYmax);
  X:=X+Xspacing*Xmajor;
end;
{===== Draw horizontal major ticks ========================================}
Y:=Ystart;
while Y<=ClipYmax do begin
  ClipDraw(ClipXmin,Y,ClipXmax,Y);
  Y:=Y+Yspacing*Ymajor;
end;
```

```
{===== Draw vertical minor ticks ========================================}
X:=Xstart;
Counter:=0;
while X<=ClipXmax do begin
  if Counter<>0 then begin
    Y:=Ystart;
    while Y<=ClipYmax do begin
      ClipDraw(X,Y-YSemiMinsize,X,Y+YSemiMinsize);
      Y:=Y+Yspacing;
    end;  {while Y<=ClipYmax}
  end;  {counter<>0?}
  Counter:=(Counter+1) mod XmaJor;
  X:=X+Xspacing;
end;  {while}
{===== Draw horizontal minor ticks ======================================}
Y:=Ystart;
Counter:=0;
while Y<=ClipYmax do begin
  if Counter<>0 then begin
    X:=Xstart;
    while X<=ClipXmax do begin
      ClipDraw(X-XSemiMinsize,Y,X+XSemiMinsize,Y);
      X:=X+Xspacing;
    end;  {while X<=ClipXmax}
  end;  {counter<>0?}
  Counter:=(Counter+1) mod YmaJor;
  Y:=Y+Yspacing;
end;  {while}
end;                                   {procedure "Grid"}
$page$  {***************************************************************}
procedure IsotropicWindow(Wxmin,Wxmax,Wymin,Wymax: real);
{-----------------------------------------------------------------------}
{    This procedure allows the user to specify a window which forces the     }
{  units to be isotropic, i.e., X units are exactly as long as Y units are.  }
{-----------------------------------------------------------------------}
const
  ViewportLimits=       451;           {mnemonic better than magic number}
type
  LimitOrder=   (Vxmin,Vxmax,Vymin,Vymax);
  LimitType=    array [LimitOrder] of real;
var
  Pac:          packed array [1..1] of char;    { \   ...sundry variables      }
  Iarray:       array [1..1] of integer;        {  \ needed by the "inq_ws"    }
  Viewport:     LimitType;                      {  / procedure, called to get }
  Error:        integer;                        { /  window limits.            }
  Wxrange, Wyrange:     real;   {X/Y range in window (world) coordinates}
  Vxrange, Vyrange:     real;   {X/Y range in viewport (virtual) coordinates}
  Wratio, Vratio:       real;   {aspect ratios of window and viewport}
  Wxmid, Wymid:         real;   {X/Y midpoints of window}
  WVratio, VWratio:     real;   {ratios of the ratios}
  Multiplier:           real;   {the amount to multiply the semirange by}
```

```
begin                                    {procedure "IsotropicWindow"}
inq_ws(ViewportLimits,0,0,4,Pac,Iarray,Viewport,Error); {get viewport limits}
if Error<>0 then
  writeln('Error ',Error:0,' in procedure "Show".');
Wxrange:=Wxmax-Wxmin;                    {range of X in desired window}
Wyrange:=Wymax-Wymin;                    {range of Y in desired window}
Wratio:=Wxrange/Wyrange;                 {aspect ratio of desired window}
Vxrange:=Viewport[Vxmax]-Viewport[Vxmin];      {range of X in current viewport}
Vyrange:=Viewport[Vymax]-Viewport[Vymin];      {range of Y in current viewport}
Vratio:=Vxrange/Vyrange;                 {aspect ratio of viewport}
if abs(Vratio)<abs(Wratio) then begin    {need more room on top and bottom}
  Wymid:=Wymin+Wyrange*0.5;              {Y midpoint in desired window}
  WVratio:=abs(Wratio/Vratio);           {ratio of aspect ratios}
  Multiplier:=Wyrange*0.5*WVratio;       {what the Y range must be extended by}
  Wymin:=Wymid-Multiplier;               {new minimum Y for window}
  Wymax:=Wymid+Multiplier;               {new maximum Y for window}
end
else begin                               {need more room on right and left}
  Wxmid:=Wxmin+Wxrange*0.5;              {X midpoint in desired window}
  VWratio:=abs(Vratio/Wratio);           {ratio of aspect ratios}
  Multiplier:=Wxrange*0.5*VWratio;       {what the X range must be extended by}
  Wxmin:=Wxmid-Multiplier;               {new minimum X for window}
  Wxmax:=Wxmid+Multiplier;               {new maximum X for window}
end;  {vratio<wratio?}
set_window(Wxmin,Wxmax,Wymin,Wymax);     {set window with twiddled parameters}
end;                                     {procedure "IsotropicWindow"}
$page$  {*******************************************************************}
begin                                    {body of program "IsoProg"}
graphics_init;                           {initialize graphics library}
display_init(Crt,Control,Error);         {initialize CRT}
if Error=0 then begin                    {if no error occurred...}
  set_aspect(Ratio,1);                   {use the whole screen}
  while true do begin                    {until the cows come home...}
    write(#12);                          {clear the alpha screen}
    prompt('Xmin, Xmax, Ymin, Ymax: '); {give the user the prompt}
    readln(Xmin,Xmax,Ymin,Ymax);         {read his/her answers}
    set_line_style(3);                   {invoke dashed line style}
    Frame;                               {draw dashed frame}
    IsotropicWindow(Xmin,Xmax,Ymin,Ymax);      {invoke isotropic units}
    ClipLimit(Xmin,Xmax,Ymin,Ymax);      {set soft clip limits to user's values}
    set_line_style(1);                   {invoke solid lines}
    Grid(1,1,0,0,1,1,1,1);               {show isotropic grid of requested area}
    prompt('Press the space bar to go on.');   {user's continuation prompt}
    read(Keyboard,Character);            {wait until user says to go on}
    clear_display;                       {clear graphics screen}
  end;  {while}
end;  {Error=0?}                         {end of conditional code}
graphics_term;                           {terminate graphics library}
end.                                     {program "IsoProg"}
```

JustProg

```
program JustProg(output);
import dgl_lib,dgl_inq;                 {get graphics routines}
const
  CrtAddr=            3;                {address of internal CRT}
  ControlWord=        0;                {device control; 0 for CRT}
type
  HJustifyType=       (Left,HCentered,Right); {horizontal justification}
  VJustifyType=       (Bottom,VCentered,Top); {vertical justification}
  AngType=            (Deg,Rad,Grad); {used by procedure "LabelDirection"}
  Str255=             string[255];      {for the procedure "Glabel"}
var
  ErrorReturn:        integer;          {variable for initialization outcome}
  HJust:              HJustifyType;     {horizontal justification variable}
  VJust:              VJustifyType;     {vertical justification variable}
  I:                  integer;          {for the strwrite statement}
  Strng:              str255;           {labelled text holder}
  CharWidth,CharHeight: real;           { \  These are global variables    }
  HJustification:     HJustifyType;     {  \ needed by the LabelJustify/   }
  VJustification:     VJustifyType;     {  / LabelDirection/CharSize       }
  CharTheta:          real;             { /  series of procedures.         }
$include 'DGLPRG:ConvVtoW'$             {needed by procedure "CharSize"}
$page$ {*****************************************************************}
procedure Frame;
{-----------------------------------------------------------------------}
{    This procedure draws a frame around the current window limits.     }
{-----------------------------------------------------------------------}
const
  WindowLimits= 450;                    {mnemonic better than magic number}
type
  LimitOrder=    (Xmin, Xmax, Ymin,Ymax);
  LimitType=     array [LimitOrder] of real;
var
  Pac:           packed array [1..1] of char;    { \  These are the sundries  }
  Iarray:        array [1..1] of integer;        {  \ needed by the call to   }
  Window:        LimitType;                      {  / the DGL procedure       }
  Error:         integer;                        { /  "inq_ws".               }
begin                                            {body of procedure "Frame"}
inq_ws(WindowLimits,0,0,4,Pac,Iarray,Window,Error);
if Error=0 then begin
  move(Window[Xmin],Window[Ymin]);               {move to lower left corner}
  line(Window[Xmin],Window[Ymax]);               {draw to upper left corner}
  line(Window[Xmax],Window[Ymax]);               {draw to upper right corner}
  line(Window[Xmax],Window[Ymin]);               {draw to lower right corner}
  line(Window[Xmin],Window[Ymin]);               {draw to lower left corner}
end  {Error=0?}
else writeln('Error ',Error:0,' occurred in "Frame"');
end;  {procedure "Frame"}              {return}
```

```
$page$ {*********************************************************************}
procedure CharSize(Height, AspectRatio: real);
{---------------------------------------------------------------------------}
{    This procedure defines character cell size and the puts the Width and  }
{  Height values into global variables for later use.  The arguments passed }
{  in are the height of the character cell in VIRTUAL coordinates, and the  }
{  aspect ratio of the character cell.  The values for the window limits    }
{  may be anything; they are taken into account and do not affect the size  }
{  of the characters, since they are defined in virtual coordinates.  This  }
{  procedure, along with Lorg and Ldir, define global variables for use by  }
{  Glabel.                                                                  }
{---------------------------------------------------------------------------}
var
  Width:        real;                {temporary spot for width}
  X0, Y0:       real;                {0,0 (virtual) in world}
  X1, Y1:       real;                {1,1 (virtual) in world}
begin                                {body of procedure "CharSize"}
ConvertVirtualToWorld(0,0,X0,Y0);    {convert 0,0 in virtual to world}
ConvertVirtualToWorld(1,1,X1,Y1);    {convert 1,1 in virtual to world}
Height:=Height*(Y1-Y0);              {convert height in virtual to world}
Width:=Height*AspectRatio*(X1-X0)/(Y1-Y0); {convert width in virtual to world}
set_char_size(Width,Height);         {invoke the parameters}
end;                                 {procedure "CharSize"}
$page$ {*********************************************************************}
procedure LabelDirection(Direction: real; Units: AngType);
{---------------------------------------------------------------------------}
{    This procedure is used in conjunction with LabelOrigin, CharSize and   }
{  Glabel.  It sets the labelling direction to be used, and places the      }
{  direction into a global variable so Glabel can use it.                   }
{---------------------------------------------------------------------------}
const
  Deg_per_rad=  57.2957795131;  {180/pi: for converting degrees to radians}
  Grad_per_rad= 63.6619772368;  {200/pi: for converting grads to radians}
begin                                {procedure "LabelDirection"}
case Units of
  Deg:  Direction:=Direction/Deg_per_rad;   {degrees to radians}
  Rad:  ;                                   {correct units already}
  Grad: Direction:=Direction/Grad_per_rad;  {grads to radians}
end;  {case}
CharTheta:=Direction;                {put into a global variable}
set_text_rot(cos(CharTheta),sin(CharTheta));    {invoke the new text direction}
end;                                 {procedure "LabelDirection"}
$page$ {*********************************************************************}
procedure LabelJustify(HJust: HJustifyType;  VJust: VJustifyType);
{---------------------------------------------------------------------------}
{    This procedure is used in conjunction with procedures CharSize,        }
{  LabelDirection, and Glabel.   This just puts a value into global         }
{  variables which will be subsequently used by Glabel.                     }
{---------------------------------------------------------------------------}
begin                                {procedure "LabelJustify"}
HJustification:=HJust;
VJustification:=VJust;
end;                                 {procedure "LabelJustify"}
```

```
$page$  {*****************************************************************}
function Atan(Y, X: real): real;
{-----------------------------------------------------------------------}
{    This function returns the value of the arctangent of Y/X, placing it    }
{  in the correct quadrant.  If Y and X  are both zero, the result is zero.  }
{-----------------------------------------------------------------------}
const
  Pi=          3.14159265359;          {pi}
begin                                  {function "Atan"}
if X=0.0 then Atan:=(Pi/2+Pi*ord(Y<0.0))*ord(Y<>0.0)
else Atan:=arctan(Y/X)+Pi*ord(X<0.0)+2*Pi*ord((X>0.0) and (Y<0.0));
end;                                   {function "Atan"}
$page$  {*****************************************************************}
procedure Glabel(Text: Str255);
{-----------------------------------------------------------------------}
{    This procedure labels a string of text at the current pen position.     }
{  It takes into account the current label direction (set by procedure      }
{  "LabelDirection", the current character size (set by procedure           }
{  "CharSize"), and the current label justification (set by procedure       }
{  "LabelJustify").                                                          }
{-----------------------------------------------------------------------}
const
  CharSizeCode=         250;           {mnemonic better than magic number}
  CurrentPosition=      259;           {ditto}
type
  Positions=            (X,Y);
  PositionType=         array [Positions] of real;
  CharAttributes=       (Width,Heighth);
  CharAttrType=         array [CharAttributes] of real;
var
  Chars:                integer;
  Charsize:             CharAttrType;
  Len,Height:           real;          {length and height of character string}
  Dx,Dy:                real;
  R,Theta:              real;          {for rectangular-to-polar conversion}
  Pac:                  packed array [1..1] of char;    {  \  These are the   }
  Iarray:               array [1..1] of integer;        {   \ sundry items    }
  Position:             PositionType;                   {   / needed for the  }
  Error:                integer;                        {  /  call to "inq_ws"}
begin                                  {procedure "Glabel"}
inq_ws(CharSizeCode,0,0,2,Pac,Iarray,Charsize,Error);     {get pen position}
if Error<>0 then writeln('Error',Error:0,' in "Glabel".');
Chars:=strlen(text);
Len:=Charsize[Width]*(7*Chars+2*(Chars-1))/9;   {length minus inter-char gap}
Height:=Charsize[Heighth]*8/15;                 {height minus inter-line gap}
Dx:=Len*(-ord(HJustification)/2);
Dy:=Height*(-ord(VJustification)/2);
R:=sqrt(Dx*Dx+Dy*Dy);                  { \  Convert to polar coordinates so  }
Theta:=Atan(Dy,Dx);                    { /  rotation is easy.                }
Theta:=Theta+CharTheta;                {add the LabelDirection angle}
Dx:=R*cos(Theta);                      { \  Convert R and the new Theta back }
Dy:=R*sin(Theta);                      { /  to rectangular coodinates.       }
inq_ws(CurrentPosition,0,0,2,Pac,Iarray,Position,Error);     {get pen position}
if Error=0 then begin
  move(Position[X]+Dx,Position[Y]+Dy);  {move to the new starting point}
  gtext(text);
end  {Error=0?}
else writeln('Error',Error:0,' in "Glabel".');
end;                                   {procedure "Glabel"}
```

```
$page$ {*****************************************************************}
begin                                   {body of program "JustProg"}
graphics_init;                          {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin             {output device initialization OK?}
  set_aspect(511,389);                  {use the whole screen}
  set_window(-1,2.5,-0.5,2.5);          {scale the window for the data}
  Frame;                                {draw a frame around the screen}
  CharSize(0.03,0.6);                   {width=3% screen width; asp. ratio=.6}
  LabelDirection(0,Deg);                {horizontal labels}
  {===== Labels at the top ===============================================}
  LabelJustify(HCentered,Top);          {label's reference point: top middle}
  for HJust:=Left to Right do begin     {horizontal loop}
    Strng:='';                          {null the string so nothing left over}
    strwrite(Strng,1,I,HJust);          {convert enumerated type to string}
    move(ord(HJust),2.4);               {move to the appropriate place}
    Glabel(Strng);                      {label the string}
  end; {for HJust}
  {===== Labels on the left edge =========================================}
  LabelJustify(Left,VCentered);         {label's reference point: left middle}
  for VJust:=Top downto Bottom do begin {vertical loop}
    Strng:='';                          {null the string so nothing left over}
    strwrite(Strng,1,I,VJust);          {convert enumerated type to string}
    move(-0.9,ord(VJust));              {move to the appropriate place}
    Glabel(Strng);                      {label the string}
  end; {for VJust}
  {===== Labels ("TEST") with different justifications =====================}
  CharSize(0.06,0.6);                   {characters a bit bigger}
  for HJust:=Left to Right do begin     {horizontal loop}
    for VJust:=Top downto Bottom do begin       {vertical loop}
      LabelJustify(HJust,VJust);                {set label justification}
      move(ord(HJust)+0.03,ord(VJust)+0.03);    { \                     }
      line(ord(HJust)-0.03,ord(VJust)-0.03);    {  \  Make the "x" at   }
      move(ord(HJust)-0.03,ord(VJust)+0.03);    {  /  the appropriate   }
      line(ord(HJust)+0.03,ord(VJust)-0.03);    { /   place.            }
      move(ord(HJust),ord(VJust));      {move to label's starting position}
      Glabel('TEST');                   {label the text}
    end; {for VJust}
  end; {for HJust}
end; {ErrorReturn=0?}
graphics_term;                          {terminate the graphics package}
end.                                    {program "JustProg"}
```

LdirProg

```
program LdirProg;                              {program name same as file name}
import dgl_lib;                                {access the necessary procedures}
const
  Crt=                 3;                      {device address of graphics raster}
  Control=             0;                      {device control word; ignored for CRT}
type
  AngType=             (Deg,Rad,Grad); {used by procedure LabelDirection}
var
  Error:               integer;                {display_init return variable; 0 = ok}
  I,J:                 integer;                {loop control variable and spare}
  Strng:               string[50];             {string to label}
  CharTheta:           real;                   {global variable for label direction}
$page$  {*****************************************************************}
procedure LabelDirection(Direction: real; Units: AngType);
{------------------------------------------------------------------------}
{    This procedure is used in conjunction with LabelOrigin, CharSize and    }
{  Glabel.  It sets the labelling direction to be used, and places the       }
{  direction into a global variable so Glabel can use it.                    }
{------------------------------------------------------------------------}
const
  Deg_per_rad=  57.2957795131;  {180/pi: for converting degrees to radians}
  Grad_per_rad= 63.6619772368;  {200/pi: for converting grads to radians}
begin                                          {procedure "LabelDirection"}
case Units of
  Deg:  Direction:=Direction/Deg_per_rad;    {degrees to radians}
  Rad:  ;                                    {correct units already}
  Grad: Direction:=Direction/Grad_per_rad;  {grads to radians}
end;  {case}
CharTheta:=Direction;                          {put into a global variable}
set_text_rot(cos(CharTheta),sin(CharTheta));    {invoke the new text direction}
end;                                           {procedure "LabelDirection"}
$page$  {*****************************************************************}
begin                                          {body of program "LdirProg"}
graphics_init;                                 {initialize graphics library}
display_init(Crt,Control,Error);               {initialize CRT}
if Error=0 then begin                          {if no error occurred...}
  set_aspect(511,389);                         {use the whole screen}
  set_window(-1,1,-1,1);                       {define appropriate window}
  set_char_size(0.05,0.08);                    {set the size for the characters}
  for I:=0 to 35 do begin                      {every ten degrees}
    Strng:='';                                 {empty the string}
    strwrite(strng,1,J,I*10:0);                {convert the loop variable to degrees}
    Strng:='-------'+Strng+' deg';             {attach prefix and suffix}
    LabelDirection(I*10,Deg);                  {specify label direction}
    move(0,0);                                 {move to the center of the screen}
    gtext(Strng);                              {label the text}
  end;  {for I}
end;  {Error=0?}
graphics_term;                                 {terminate graphics library}
end.                                           {program "LdirProg"}
```

LOCATOR

```
$debug$
program Test(output);
import dgl_vars,dgl_types,dgl_lib,dgl_poly,dgl_inq;
type
  Commands=              0..8;           {nine commands total}
  RealArray=             array [1..5] of real;
const
  FS=                    chr(28);        {right arrow}
  BS=                    chr(8);         {left arrow or backspace}
  US=                    chr(31);        {up arrow}
  LF=                    chr(10);        {down arrow}
  CR=                    chr(13);        {carriage return}
  MinX=                  0;              {minimum X value for screen}
  MinY=                  0;              {minimum Y value for screen}
  MaxX=                  511;            {maximum X value for screen}
  MaxY=                  389;            {maximum Y value for screen}
  Xrange=                MaxX-MinX;      {total range of X}
  Yrange=                MaxY-MinY;      {total range of Y}
  LocatorAddress=        2;              {2 for knob,706 for 9111}
var
  Error_num:             integer;        {error return variable}
  I,TempInt:             integer;        {utility variables}
  ButtonValue:           integer;        {which button selected?}
  Xin,Yin:               real;           {location of digitized point}
  Xlast,Ylast:           real;           {last digitized point}
  CharWidth,CharHeight:  real;           {char size in world coords}
  Done:                  boolean;        {are we supposed to quit?}
  NewLine:               boolean;        {start new line?}
  TempString:            Gstring255;     {utility variable}
  EchoSelect,EchoSelector:       0..9;   {menu selection}
  MenuTop:               real;
  CellWidth:             real;           {width of menu spaces}
  Command:               Commands;       {which command selected?}
$page$  {************************************************************************}
procedure DrawMenu;
var
  I:                     integer;        {loop-control variable}
  Ylabel:                real;           {Y position of entree label}
  Yarray:                RealArray;
```

```
{-----------------------------------------------------------------------}
procedure MenuCell(I:integer);
var
  TempPitch:            real;           {temporary variable}
  Xlabel:               real;           {X position of entree label}
  Xarray:               RealArray;      {X positions of entree cell}
begin                                   {procedure MenuCell}
case I of
  0: begin
      TempString:='STOP';               {label text}
      Xarray[1]:=0;                     { \                          }
      Xarray[2]:=2*CellWidth;           {  \                         }
      Xarray[3]:=2*CellWidth;           {   > X positions for box    }
      Xarray[4]:=0;                     {  /                         }
      Xarray[5]:=0;                     { /                          }
      Xlabel:=MinX+CellWidth-strlen(TempString)*CharWidth/2;
    end;
  1..10: begin
      TempPitch:=CellWidth*I;           {temporary shorthand variable}
      Xarray[1]:=CellWidth+TempPitch;   { \                          }
      Xarray[2]:=2*CellWidth+TempPitch; {  \                         }
      Xarray[3]:=2*CellWidth+TempPitch; {   > X positions for box    }
      Xarray[4]:=CellWidth+TempPitch;   {  /                         }
      Xarray[5]:=CellWidth+TempPitch;   { /                          }
      TempString:=' ';                  {label text}
      if I<=8 then strwrite(TempString,1,TempInt,I:1);
      Xlabel:=Xarray[1]+CellWidth/2+strlen(TempString)*CharWidth/2;
    end
end; {case I of}
polyline(5,Xarray,Yarray);              {draw perimeter of cell}
move(Xlabel,Ylabel);                    {move to the right place}
gtext(TempString);                      {label the text}
end;                                    {procedure MenuCell}
{-----------------------------------------------------------------------}
begin                                   {procedure DrawMenu}
Yarray[1]:=MinY;                        { \                          }
Yarray[2]:=MinY;                        {  \                         }
Yarray[3]:=MenuTop;                     {   > Y values for box       }
Yarray[4]:=MenuTop;                     {  /                         }
Yarray[5]:=MinY;                        { /                          }
Ylabel:=MinY+(MenuTop-MinY)/2-CharHeight/2;     {Y position of label}
for I:=0 to 10 do MenuCell(I);          {do all the entree cells}
end;  {procedure DrawMenu}
$page$  {***************************************************************}
function CheckMenu(Xin:real):Commands;
begin                                   {function CheckMenu}
if Xin<2*CellWidth then CheckMenu:=0    {X outside of menu?}
else begin
  TempInt:=trunc((Xin-CellWidth)/CellWidth);    {which sell chosen?}
  if TempInt>8 then CheckMenu:=Command
  else CheckMenu:=TempInt
end;
end;   {function  CheckMenu }
```

```
$page$  {*****************************************************************}
begin                                     {Main program}
graphics_init;                            {initialize the graphics system}
display_init(3,0,Error_Num);              {which output device?}
if Error_Num<>0 then begin                {output devic initialization OK?}
  writeln('I failed to initialize the display.');
  writeln('Error number ',Error_Num:2,' was returned.');
end {if Error_Num<>0}
else begin
  LOCATOR_init(LocatorAddress,Error_Num);
  if Error_Num<>0 then begin
    writeln('I failed to initialize the locator.');
    writeln('Error number ',Error_Num:2,' was returned.');
  end  {if Error_Num<>0}
  else begin                              {No errors so far}
    set_aspect(511,389);                  {use whole screen}
    set_window(0,511,0,389);              {scale window for data}
    CharWidth:=0.035*511;                 {char width: 3.5% of screen width}
    CharHeight:=0.05*389;                 {char height: 5% of screen height}
    set_char_size(CharWidth,CharHeight);{install character size}
    MenuTop:=Yrange/13;                   {menu is 1/13 the total screen height}
    CellWidth:=Xrange/12;                 {each entree cell 1/12 screen width}
    DrawMenu;                             {draw the menu}
    NewLine:=true;                        {yes, we are starting a new line}
    EchoSelect:=4;                        {start program with default command}
    Command:=4;                           {ditto}
    Done:=false;                          {no, we're not done yet}
    repeat
      if NewLine then                     {starting a new line?}
        EchoSelector:=2
      else
        EchoSelector:=EchoSelect;
      await_locator(EchoSelector,ButtonValue,Xin,Yin);
      if Yin<MenuTop then begin           {user choose menu option?}
        NewLine:=true;                    {start a new line next time}
        Command:=CheckMenu(Xin);          {determine menu selection}
        case Command of                   {which command}
          0: Done:=true;                  {yes, we're done with the program}
          1: EchoSelect:=1;               { \                              }
          2: EchoSelect:=2;               {  \                             }
          3: EchoSelect:=3;               {   \                            }
          4: EchoSelect:=4;               {    \  Select the appropriate   }
          5: EchoSelect:=5;               {    /  EchoSelector.            }
          6: EchoSelect:=6;               {   /                            }
          7: EchoSelect:=7;               {  /                             }
          8: EchoSelect:=8;               { /                              }
        end  {case}
      end  {if}
      else begin                          {not  a menu selection}
        if NewLine then begin             {start a new line}
          NewLine:=false;                 {now we're in the middle of a line}
          set_echo_pos(Xin,Yin);          {move the graphics cursor}
          move(Xin,Yin);                  {cause line-drawing to start there}
          Ylast:=Yin;                     {remember the last X...}
          Xlast:=Xin;                     {...and the last Y}
        end
```

```
          else begin
            set_echo_pos(Xin,Yin);        {move the graphics cursor}
            if (Xin=Xlast) and (Yin=Ylast) then NewLine:=true
            else begin
              case EchoSelect of
                1..7: line(Xin,Yin);      {draw a line}
                8: begin
                    line(Xlast,Yin);
                    line(Xin,Yin);
                    line(Xin,Ylast);
                    line(Xlast,Ylast);
                    NewLine:=true;
                  end
                otherwise
              end; {case EchoSelect of}
              Xlast:=Xin;                 {remember the last X...}
              Ylast:=Yin;                 {...and the last Y}
            end
          end;
        end;
      until Done;                         {are we done yet?}
      locator_term;                       {terminate the locator}
      display_term;                       {terminate the display}
    end; {Error trap}
  end;
  graphics_term;                          {terminate the graphics system}
  end.                                    {Main program}
```

LogPlot

```
program LogPlot(Keyboard,output);
import dgl_lib;
const
  Xmin=         -4;                     { \                    }
  Xmax=         2;                      {  \  Decade minima    }
  Ymin=         0;                      {  /  and maxima.      }
  Ymax=         3;                      { /                    }
  Crt=          3;                      {device address of graphics raster}
  Control=      0;                      {device control word; ignored for CRT}
type
  RDataType=    array [1..15] of real;
const
  Xvalues=      RDataType[0.0003, 0.0009, 0.004, 0.008, 0.01, 0.07, 0.22, 0.5,
                          1.2, 2.6, 8.9, 18.6, 34, 56, 97];
  Yvalues=      RDataType[1.1, 4.5, 13.38, 45.9, 60.33, 130.7, 346, 690.4,
                          899, 933, 903, 841, 720, 505, 390];
var
  Error:                integer;        {display_init return variable; 0 = ok}
  Decade:               integer;
  Units, UpperLimit:    integer;
  X, Y:                 real;
  I:                    integer;
```

```
$page$  {*******************************************************************}
function Log10(X: real): real;
{-----------------------------------------------------------------------------}
{  This function returns the logarithm to the base ten of a number.          }
{-----------------------------------------------------------------------------}
const
  Log_10=       2.30258509299;          {log to the base e of 10}
begin                                   {function "Log10"}
Log10:=ln(X)/Log_10;
end;                                    {function "Log10"}
$page$  {*******************************************************************}
begin                                   {body of program "LogPlot"}
graphics_init;                          {initialize the graphics system}
display_init(Crt,Control,Error);
if Error=0 then begin
  set_aspect(511,389);
  set_window(Xmin,Xmax,Ymin,Ymax);
  {===== Draw and label logarithmic X-axis grid ============================}
  for Decade:=Xmin to Xmax do begin     {one decade equals one mantissa cycle}
    if Decade=Xmax then UpperLimit:=1
    else UpperLimit:=9;
    for Units:=1 to UpperLimit do begin         {do 2-9 if not last cycle}
      X:=Decade+Log10(Units);
      move(X,Ymin);
      line(X,Ymax);
    end;  {for units}
  end;  {for decade}
  {===== Draw and label logarithmic Y-axis grid ============================}
  for Decade:=Ymin to Ymax do begin     {one decade equals one mantissa cycle}
    if Decade=Ymax then UpperLimit:=1
    else UpperLimit:=9;
    for Units:=1 to UpperLimit do begin         {do 2-9 if not last cycle}
      Y:=Decade+Log10(Units);
      move(Xmin,Y);
      line(Xmax,Y);
    end;  {for units}
  end;  {for decade}
  {===== Draw the logarithmic data curve ===================================}
  for I:=1 to 15 do begin
    if I=1 then move(Log10(XValues[I]),Log10(Yvalues[I]))
    else line(Log10(XValues[I]),Log10(Yvalues[I]));
  end;  {for i}
end;  {Error=0?}                        {end of conditional code}
graphics_term;                          {terminate graphics library}
end.                                    {program "LogPlot"}
```

MarkrProg

```
program MarkrProg(output);
import dgl_lib,dgl_inq;
const
  CrtAddr=              3;
  ControlWord=          0;
type
  MarkerNumType=        array [0..4] of integer;
  DataType=             array [0..10] of integer;
const
  MarkerNumber=         MarkerNumType[2,5,6,8,13];
  Data=                 DataType[0,2,1,4,3,3,1,5,3,4,6];
var
  ErrorReturn:          integer;
  I, J:                 integer;
$page$  {*********************************************************************}
begin                                     {program "MarkrProg"}
graphics_init;
display_init(CrtAddr,ControlWord,ErrorReturn);
if ErrorReturn=0 then begin
  set_aspect(511,389);
  set_window(0,10,0,10);
  move(0,0);  line(0,10);  line(10,10);  line(10,0);  line(0,0);
  for I:=0 to 4 do begin
    for J:=0 to 10 do begin
      if J<>0 then marker(MarkerNumber[I]);
      if J=0 then move(J,Data[J]+I)
      else line(J,Data[J]+I);
    end;  {for J}
  end;  {for i}
end;  {ErrorReturn=0?}
graphics_term;
end.                                      {program "MarkrProg"}
```

PLineProg

```
program PLineProg(output);
import dgl_lib,dgl_inq;
const
  CrtAddr=              3;
  ControlWord=          0;
type
  RDataType=            array [0..10] of real;
const
  Xvalues=              RDataType[0,1,2,3,4,5,6,7,8,9,10];
  Yvalues=              RDataType[0,2,1,4,3,3,1,5,3,4,6];
var
  ErrorReturn:          integer;
  X, Y:                 RDataType;
$page$ {*******************************************************************}
begin                                   {program "PLineProg"}
graphics_init;
display_init(CrtAddr,ControlWord,ErrorReturn);
if ErrorReturn=0 then begin
  set_aspect(511,389);
  set_window(0,10,0,10);
  move(0,0);  line(0,10);  line(10,10);  line(10,0);  line(0,0);
  X:=Xvalues;  Y:=Yvalues;
  polyline(11,X,Y);
end;  {ErrorReturn=0?}
graphics_term;
end.                                    {program "PLineProg"}
```

PolyProg

```
program PolyProg(output);                {program name same as file name}
import
  dgl_lib,dgl_types,dgl_poly,dgl_inq;   {access the necessary procedures}
const
  MaxPoints=          27;               {number of points in arrays}
  Crt=                3;                {device address of graphics raster}
  Control=            0;                {device control word; ignored for CRT}
type
  Reals=       array [1..MaxPoints] of real;   {to contain X and Y values}
  Word=        -32768..32767;                  {16-bit word}
  Integers=    array [1..MaxPoints] of Word;      {to contain op. selectors}
const
  Xvalues=     Reals[ 1.5, 2.5, 2.5, 1.5,-1.5,-2.5,-2.5,-1.5,  {Octagon}
                     -2.5, 2.5, 2.5,-2.5,-2.5,                 {Box}
                     -2.5,-4.5,-2.5,-5.0,-4.0,                 {Left leg}
                      2.5, 4.5, 2.5, 5.0, 4.0,                 {Right leg}
                     -0.5,-1.0, 1.0, 0.5];                     {Nozzle}
  Yvalues=     Reals[ 1.0, 2.0, 3.0, 4.0, 4.0, 3.0, 2.0, 1.0,  {Octagon}
                      1.0, 1.0,-2.0,-2.0, 1.0,                 {Box}
                     -2.0,-4.0, 0.0,-4.0,-4.0,                 {Left leg}
                     -2.0,-4.0, 0.0,-4.0,-4.0,                 {Right leg}
                     -2.0,-3.0,-3.0,-2.0];                     {Nozzle}
  OpCodes=     Integers[2,1,1,1,1,1,1,1,                       {Octagon}
                        2,1,1,1,1,                             {Box}
                        2,1,1,2,1,                             {Right leg}
                        2,1,1,2,1,                             {Left leg}
                        2,1,1,1];                              {Nozzle}
var
  Error:              integer;          {display_init return variable; 0 = ok}
  I:                  integer;          {loop control variable}
  LemX, LemY:         Reals;            {so we can pass it to "polygon"}
  OpSelectors:        Integers;         {ditto}
  Points:             integer;          {ditto}
$page$  {*******************************************************************}
begin                                   {body of program "PolyProg"}
LemX:=Xvalues;                          { \  Put into variable array so   }
LemY:=Yvalues;                          {  > it can be passed by          }
OpSelectors:=OpCodes;                   { /  reference into the DGL proc.}
Points:=MaxPoints;                      {put constant into an array variable}
graphics_init;                          {initialize graphics library}
display_init(Crt,Control,Error);        {initialize CRT}
if Error=0 then begin                   {if no error occurred...}
  set_aspect(511,389);                  {use the whole screen}
  set_window(-13,13,-10,10);            {invoke isotropic units}
  polygon(Points,LemX,LemY,OpSelectors);     {draw the lines}
end;  {Error=0?}                        {end of conditional code}
graphics_term;                          {terminate graphics library}
end.  {program "PolyProg"}              {end of program}
```

SinAspect

```
program SinAspect(output);
import dgl_lib;                          {get graphics routines}
const
  CrtAddr=          3;                   {address of internal CRT}
  ControlWord=      0;                   {device control; 0 for CRT}
var
  ErrorReturn:  integer;                 {variable for initialization outcome}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$            {function: y:=f(x) }
$page$  {*************************************************************}
begin                                    {body of program "SinAspect"}
graphics_init;                           {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin              {output device initialization OK?}
  set_aspect(511,389);                   {use the whole screen}
  set_window(0,100,0.16,0.18);           {scale the window for the data}
  for X:=1 to 100 do begin               {100 points total}
    Y:=DataPoint(X);                     {get a point from the function}
    if X=1 then move(X,Y)                {move to the first point...}
    else line(X,Y)                       {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                           {terminate the graphics package}
end.                                     {program "SinAspect"}
```

SinAxes1

```
program SinAxes1(output);
import dgl_lib, dgl_inq;                {get graphics routines}
const
  CrtAddr=      3;                      {address of internal CRT}
  ControlWord=  0;                      {device control; 0 for CRT}
type
  RoundType=    (Up, Down, Near);       {used by procedure Round2}
var
  CharWidth:    real;                   {width of char in world coords}
  CharHeight:   real;                   {height of char in world coords}
  Text:         string[20];             {temporary holding place for text}
  ErrorReturn:  integer;                {variable for initialization outcome}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$            {function: y:=f(x) }
$page$  {*****************************************************************}
function Round2(N, M: real; Mode: RoundType): real;
{-----------------------------------------------------------------------}
{  This function rounds "N" to the nearest "M", according to "Mode".  This   }
{  function works only when the argument is in the range of MININT..MAXINT.  }
{-----------------------------------------------------------------------}
const
  epsilon=      1E-10;                  {roundoff error fudge factor}
var
  Rounded:      real;                   {temporary holding area}
  Negative:     boolean;                {flag: "It is negative?"}
begin                                   {body of "Round2"}
Negative:=(N<0.0);                      {is the number negative?}
if Negative then begin
  N:=abs(N);                            {work with a positive number}
  if Mode=Up then Mode:=Down            {if number is negative, ...}
  else if Mode=Down then Mode:=Up;      {...reverse up and down}
end;
case Mode of                            {should we round the number...}
  Down: Rounded:=trunc(N/M)*M;          {...left on the number line?}
  Up:   begin
          Rounded:=N/M;                 {...right on the number line?}
          if abs(Rounded-round(Rounded))>epsilon then
            Rounded:=(trunc(Rounded)+1.0)*M
          else
            Rounded:=trunc(Rounded)*M;
        end;
  Near: Rounded:=trunc(N/M+M*0.5)*M;    {...to the nearest multiple?}
end;  {case}
if Negative then Rounded:=-Rounded;     {reinstate the sign}
Round2:=Rounded;                        {assign to function name}
end;                                    {function "Round2"}
```

```
$page$ {*******************************************************************}
procedure Xaxis(Spacing,Location,Xmin,Xmax: real;
                                     Major: integer;
                              Majsize,Minsize: real);
{-----------------------------------------------------------------------}
{  This procedure draws an X-axis at any intersection point on the plotting }
{  surface.  Parameters are as follows:                                   }
{    Spacing:   The distance between tick marks on the axis.              }
{    Location:  The Y-value of the X-axis.                                }
{    Xmin,Xmax: The left and right ends of the X-axis, respectively.      }
{    Major:     The number of tick marks to ge before drawing a major tick }
{               mark.  If Major=5, every fifth tick mark will be major.   }
{    Majsize:   The length, in current units, of the major tick marks.    }
{    Minsize:   The length, in current units, of the minor tick marks.    }
{-----------------------------------------------------------------------}
var
  X:             real;                 {current X of tick marks}
  SemiMinsize:   real;                 {half of minor tick size}
  SemiMajsize:   real;                 {half of major tick size}
  Counter:       integer;              {keeps track of when to do major ticks}
begin                                  {body of procedure "Xaxis"}
move(Xmin,Location);                   {left end of the x-axis}
line(Xmax,Location);                   {draw x-axis}
SemiMinsize:=Minsize*0.5;              {half of every tick mark needs to...}
SemiMajsize:=Majsize*0.5;              {...be on each side of the axis}
X:=Round2(Xmin,Spacing*Major,Down);    {round start point to next lower major}
Counter:=0;                            {start with a major tick}
while X<=Xmax do begin                 {loop until greater than Xmax}
  if Counter=0 then begin              {should we do a major tick?}
    move(X,Location-SemiMajsize);      {move to bottom of major tick, and...}
    line(X,Location+SemiMajsize);      {...draw to the top of major tick}
  end  {Counter=0?}
  else begin
    move(X,Location-SemiMinsize);      {move to bottom of minor tick, and...}
    line(X,Location+SemiMinsize);      {...draw to the top of minor tick}
  end;  {else begin}
  Counter:=(Counter+1) mod Major;      {keep track of which length tick to do}
  X:=X+Spacing;                        {go to next tick position}
end;  {while}                          {loop if not done}
end;                                   {procedure "Xaxis"}
```

```
$page$  {*********************************************************************}
procedure Yaxis(Spacing,Location,Ymin,Ymax: real;
                                     Major: integer;
                          Majsize,Minsize: real);
{---------------------------------------------------------------------------}
{  This procedure draws an Y-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                      }
{    Spacing:   The distance between tick marks on the axis.                 }
{    Location:  The X-value of the Y-axis.                                   }
{    Ymin,Ymax: The left and right ends of the Y-axis, respectively.         }
{    Major:     The number of tick marks to ge before drawing a major tick   }
{               mark.  If Major=5, every fifth tick mark will be major.      }
{    Majsize:   The length, in current units, of the major tick marks.       }
{    Minsize:   The length, in current units, of the minor tick marks.       }
{---------------------------------------------------------------------------}
var
  Y:             real;                   {current Y of tick marks}
  SemiMinsize:   real;                   {half of minor tick size}
  SemiMajsize:   real;                   {half of major tick size}
  Counter:       integer;                {Keeps track of when to do major ticks}
begin                                    {body of procedure "Axes"}
move(Location,Ymin);                     {lower end of the y-axis}
line(Location,Ymax);                     {draw y-axis}
SemiMinsize:=Minsize*0.5;                {half of every tick mark needs to...}
SemiMajsize:=Majsize*0.5;                {...be on each side of the axis}
Y:=Round2(Ymin,Spacing*Major,Down);      {round start point to next lower major}
Counter:=0;                              {start with a major tick}
while Y<=Ymax do begin                   {loop until greater than Xmax}
  if Counter=0 then begin                {should we do a major tick?}
    move(Location-SemiMajsize,Y);        {move to left of major tick, and...}
    line(Location+SemiMajsize,Y);        {...draw to the right of major tick}
  end  {Counter=0?}
  else begin
    move(Location-SemiMinsize,Y);        {move to left of major tick, and...}
    line(Location+SemiMinsize,Y);        {...draw to the right of major tick}
  end;  {else begin}
  Counter:=(Counter+1) mod Major;        {keep track of which length tick to do}
  Y:=Y+Spacing;                          {go to next tick position}
end;  {while}                            {loop if not done}
end;                                     {procedure "Yaxis"}
```

```
$page$ {*******************************************************************}
begin                                {body of program "SinAxes1"}
graphics_init;                       {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin          {output device initialization OK?}
  set_aspect(511,389);               {use the whole screen}
  CharWidth:=2*0.04;                 {char width: 4% of screen width}
  CharHeight:=2*0.08;                {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='VOLTAGE VARIANCE';          {define text to be labelled}
  for X:=-3 to 3 do begin            {make "bold" label}
    move(-(strlen(Text)*CharWidth)/2+X*0.002,0.9);      {center label}
    gtext(Text);                     {label the text}
  end;
  set_text_rot(0,1);                 {vertical labels}
  CharWidth:=2*0.025;                {char width: 2.5% of screen width}
  CharHeight:=2*0.04;                {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='Voltage';                   {define the text to be labelled}
  move(-0.9,-(strlen(Text)*CharWidth)/2);      {start point of centered label}
  gtext(Text);                       {label the text}
  Text:='Time (seconds)';            {define the text to be labelled}
  set_text_rot(1,0);                 {horizontal labels}
  move(-(strlen(Text)*CharWidth)/2,-0.92);     {start point of centered label}
  gtext(Text);                       {label the text}
  set_viewport(0.1,0.99,0.12,0.7);   {define subset of screen}
  move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);       {scale the window for the data}
  Xaxis(1,0.16,-50,150,5,0.001,0.0005); {draw the x-axis}
  Yaxis(0.001,0,0.1,0.2,5,2,1);      {draw the y-axis}
  for X:=1 to 100 do begin           {100 points total}
    Y:=DataPoint(X);                 {get a point from the function}
    if X=1 then move(X,Y)            {move to the first point...}
    else line(X,Y);                  {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                       {terminate the graphics package}
end.                                 {program "SinAxes1"}
```

SinAxes2

```
program SinAxes2(output);
import dgl_lib;                         {get graphics routines}
const
  CrtAddr=            3;                {address of internal CRT}
  ControlWord=        0;                {device control; 0 for CRT}
type
  RoundType=          (Up, Down, Near);        {used by function Round2}
var
  CharWidth:          real;             {width of char is world coords}
  CharHeight:         real;             {height of char is world coords}
  Text:               string[20];       {temporary holding place for text}
  ErrorReturn:        integer;          {variable for initialization outcome}
  I:                  integer;          {return variable from STRWRITE}
  X:                  integer;
  Y:                  real;
  ClipXmin, ClipXmax: real;             {soft clip limits in x}
  ClipYmin, ClipYmax: real;             {soft clip limits in y}
$include 'DGLPRG:DataPoint'$            {function: y:=f(x) }
$page$  {******************************************************************}
procedure ClipLimit(Xmin, Xmax, Ymin, Ymax: real);
{-------------------------------------------------------------------------}
{  This procedure defines the four global variables which specify where the  }
{  soft clip limits are.                                                  }
{-------------------------------------------------------------------------}
begin                                   {body of procedure "ClipLimit"}
if Xmin<Xmax then begin                 { \                          }
  ClipXmin:=Xmin;                       {  \    Force the minimum soft  }
  ClipXmax:=Xmax;                       {   \   clip limit in X to be   }
end                                     {    \  the smaller of the two  }
else begin                              {    /  X values passed into    }
  ClipXmin:=Xmax;                       {   /   the procedure.          }
  ClipXmax:=Xmin;                       {  /                          }
end;                                    { /                           }
if Ymin<Ymax then begin                 { \                          }
  ClipYmin:=Ymin;                       {  \    Force the minimum soft  }
  ClipYmax:=Ymax;                       {   \   clip limit in Y to be   }
end                                     {    \  the smaller of the two  }
else begin                              {    /  Y values passed into    }
  ClipYmin:=Ymax;                       {   /   the procedure.          }
  ClipYmax:=Ymin;                       {  /                          }
end;                                    { /                           }
end;                                    {procedure "ClipLimit"}
$page$  {******************************************************************}
procedure ClipDraw(X1, Y1, X2, Y2: real);
{-------------------------------------------------------------------------}
{    This procedure takes the endpoints of a line, and clips it.  The soft   }
{  clip limits are the real global variables ClipXmin, ClipXmax, ClipYmin,   }
{  and ClipYmax.  These may be defined through the procedure ClipLimit.      }
{-------------------------------------------------------------------------}
```

```
label
  1;
type
  Edges=         (Left,Right,Top,Bottom);          {possible edges to cross}
  OutOfBounds=   set of Edges;                     {set of edges crossed}
var
  Out,Out1,Out2:OutOfBounds;
  X, Y:          real;
{---------------------------------------------------------------------------}
procedure Code(X, Y: real; var Out: OutOfBounds);
begin                                   {nested procedure "Code"}
Out:=[];                                {null set}
if x<ClipXmin then Out:=[left]            {off left edge?}
else if x>ClipXmax then Out:=[right];     {off right edge?}
if y<ClipYmin then Out:=Out+[bottom]      {off the bottom?}
else if y>ClipYmax then Out:=Out+[top];   {off the top?}
end;                                    {nested procedure "Code"}
{---------------------------------------------------------------------------}
begin                                   {body of procedure "ClipDraw"}
Code(X1,Y1,Out1);                       {figure status of point 1}
Code(X2,Y2,Out2);                       {figure status of point 2}
while (Out1<>[]) or (Out2<>[]) do begin {loop while either point out of range}
  if (Out1*Out2)<>[] then goto 1;       {if intersection non-null, no line}
  if Out1<>[] then Out:=Out1
    else Out:=Out2;                     {Out is the non-empty one}
  if left in Out then begin             {it crosses the left edge}
    y:=Y1+(Y2-Y1)*(ClipXmin-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmin;                        {new x is left edge}
  end  {left in Out?}
  else if right in Out then begin       {it crosses right edge}
    y:=Y1+(Y2-Y1)*(ClipXmax-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmax;                        {new x is right edge}
  end  {right in Out?}
  else if bottom in Out then begin      {it crosses the bottom edge}
    x:=X1+(X2-X1)*(ClipYmin-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmin;                        {new y is bottom edge}
  end  {bottom in Out?}
  else if top in Out then begin         {it crosses the top edge}
    x:=X1+(X2-X1)*(ClipYmax-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmax;                        {new y is top edge}
  end;  {top in Out?}
  if Out=Out1 then begin
    X1:=x;   Y1:=y;   Code(x,y,Out1);   {redefine first end point}
  end  {Out=Out1?}
  else begin
    X2:=x;   Y2:=y;   Code(x,y,Out2);   {redefine second end point}
  end;  {else begin}
end;  {while}
move(x1,y1);                            {if we get to this point, the line...}
line(x2,y2);                            {...is completely visible, so draw it}
1:   end;                               {procedure "ClipDraw"}
```

```
$page$ {********************************************************************}
function Round2(N, M: real; Mode: RoundType): real;
{-----------------------------------------------------------------------------}
{  This function rounds "N" to the nearest "M", according to "Mode".  This   }
{  function works only when the argument is in the range of MININT..MAXINT.  }
{-----------------------------------------------------------------------------}
const
  epsilon=      1E-10;                  {roundoff error fudge factor}
var
  Rounded:      real;                   {temporary holding area}
  Negative:     boolean;                {flag: "It is negative?"}
begin                                   {body of "Round2"}
Negative:=(N<0.0);                      {is the number negative?}
if Negative then begin
  N:=abs(N);                            {work with a positive number}
  if Mode=Up then Mode:=Down            {if number is negative, ...}
  else if Mode=Down then Mode:=Up;      {...reverse up and down}
end;
case Mode of                            {should we round the number...}
  Down: Rounded:=trunc(N/M)*M;          {...left on the number line?}
  Up:   begin
          Rounded:=N/M;                 {...right on the number line?}
          if abs(Rounded-round(Rounded))>epsilon then
            Rounded:=(trunc(Rounded)+1.0)*M
          else
            Rounded:=trunc(Rounded)*M;
        end;
  Near: Rounded:=trunc(N/M+M*0.5)*M;    {...to the nearest multiple?}
end;  {case}
if Negative then Rounded:=-Rounded;     {reinstate the sign}
Round2:=Rounded;                        {assign to function name}
end;                                    {function "Round2"}
$page$ {********************************************************************}
procedure XaxisClip(Spacing, Location: real; Major: integer;
  Majsize,Minsize: real);
{-----------------------------------------------------------------------------}
{  This procedure draws an X-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                       }
{    Spacing:   The distance between tick marks on the axis.                  }
{    Location:  The Y-value of the X-axis.                                    }
{    Major:     The number of tick marks to ge before drawing a major tick    }
{               mark.  If Major=5, every fifth tick mark will be major.       }
{    Majsize:   The length, in world units, of the major tick marks.          }
{    Minsize:   The length, in world units, of the minor tick marks.          }
{-----------------------------------------------------------------------------}
var
  X:            real;                   {X position of tick marks}
  SemiMajsize:  real;                   {half of major tick size}
  SemiMinsize:  real;                   {half of minor tick size}
  Counter:      integer;                {keeps track of when to do major ticks}
```

```
begin                                    {body of procedure "XaxisClip"}
SemiMajsize:=MajSize*0.5;                {calculate half of major tick size}
SemiMinsize:=MinSize*0.5;                {calculate half of minor tick size}
Counter:=0;                              {start with a major tick}
ClipDraw(ClipXmin,Location,ClipXmax,Location);  {draw the X-axis itself}
X:=Round2(ClipXmin,Spacing*Major,Down); {round to next lower major}
while X<=ClipXmax do begin               {loop until greater than ClipXmax}
  if Counter=0 then                      {do a major tick mark?}
    ClipDraw(X,Location-SemiMajsize,X,Location+SemiMajsize)
  else
    ClipDraw(X,Location-SemiMinsize,X,Location+SemiMinsize);    {do minor tick}
  Counter:=(Counter+1) mod Major;        {keep track of which length tick to do}
  X:=X+Spacing;                          {go to next tick position}
end;  {while}
end;                                     {procedure "XaxisClip"}
$page$  {******************************************************************}
procedure YaxisClip(Spacing, Location: real; Major: integer;
  Majsize, Minsize: real);
{-------------------------------------------------------------------------}
{  This procedure draws an Y-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                    }
{    Spacing:   The distance between tick marks on the axis.               }
{    Location:  The X-value of the Y-axis.                                 }
{    Major:     The number of tick marks to ge before drawing a major tick  }
{               mark.  If Major=5, every fifth tick mark will be major.    }
{    Majsize:   The length, in world units, of the major tick marks.       }
{    Minsize:   The length, in world units, of the minor tick marks.       }
{-------------------------------------------------------------------------}
var
  Y:            real;                    {Y position of tick marks}
  SemiMajsize:  real;                    {half of major tick size}
  SemiMinsize:  real;                    {half of minor tick size}
  Counter:      integer;                 {keeps track of when to do major ticks}
begin                                    {body of procedure "YaxisClip"}
SemiMajsize:=Majsize*0.5;                {calculate half of major tick size}
SemiMinsize:=Minsize*0.5;                {calculate half of minor tick size}
Counter:=0;                              {start with a major tick}
ClipDraw(Location,ClipYmin,Location,ClipYmax);
Y:=Round2(ClipYmin,Spacing*Major,Down); {round to next lower major}
while Y<=ClipYmax do begin               {loop until greater than Ymax}
  if Counter=0 then                      {should we do a major tick?}
    ClipDraw(Location-SemiMajsize,Y,Location+SemiMajsize,Y)
  else
    ClipDraw(Location-SemiMinsize,Y,Location+SemiMinsize,Y);
  Counter:=(Counter+1) mod Major;        {keep track of which size tick to do}
  Y:=Y+Spacing;                          {go to next tick position}
end;  {while}
end;                                     {procedure "YaxisClip"}
```

```
$page$  {*****************************************************************}
begin                                 {body of program "SinAxes2"}
graphics_init;                        {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin           {output device initialization OK?}
  set_aspect(511,389);                {use the whole screen}
  CharWidth:=2*0.04;                  {char width: 4% of screen width}
  CharHeight:=2*0.08;                 {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='VOLTAGE VARIANCE';           {define text to be labelled}
  for X:=-3 to 3 do begin             {make "bold" label}
    move(-(strlen(Text)*CharWidth)/2+X*0.002,0.9);      {center label}
    gtext(Text);                      {label the text}
  end;
  set_text_rot(0,1);                  {vertical labels}
  CharWidth:=2*0.025;                 {char width: 2.5% of screen width}
  CharHeight:=2*0.04;                 {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);  {install char size}
  Text:='Voltage';                    {define text to be labelled}
  move(-0.97,-(strlen(Text)*CharWidth)/2);      {start point of centered label}
  gtext(Text);                        {label the text}
  Text:='Time (seconds)';             {define text to be labelled}
  set_text_rot(1,0);                  {horizontal labels}
  move(-(strlen(Text)*CharWidth)/2,-0.92);      {start point of centered label}
  gtext(Text);                        {label the text}
  set_viewport(0.1,0.99,0.12,0.7);    {define subset of the screen}
  move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);        {scale the window for the data}
  ClipLimit(0,100,0.16,0.18);         {define the soft clip limits}
  XaxisClip(1,0.16,5,0.0008,0.0004);  {draw the clipped X-axis}
  YaxisClip(0.0005,0,5,2,1);          {draw the clipped Y-axis}
  CharWidth:=1.3;                     {char width: 1.3 user X units wide}
  CharHeight:=0.0008;                 {char height: .0008 user Y units high}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='';                           {erase previous definitions of string}
  for X:=0 to 10 do begin             {eleven X labels}
    strwrite(Text,1,I,X*10:0);        {convert number to string}
    move(X*10-(strlen(Text)*CharWidth)/2,0.1593);      {center the label}
    gtext(Text);                      {label the text}
  end;  {for x}
  Y:=0.16;                            {starting Y position for Y labels}
  repeat
    strwrite(Text,1,X,Y:6:4);         {convert number to string}
    move(-8,Y-0.0002);                {center the text vertically}
    gtext(Text);                      {label the text}
    Y:=Y+0.0025;                      {next Y position}
  until Y>0.18;                       {terminating condition}
  for X:=1 to 100 do begin            {100 points total}
    Y:=DataPoint(X);                  {get a point from the function}
    if X=1 then move(X,Y)             {move to the first point...}
    else line(X,Y);                   {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                        {terminate the graphics package}
end.                                  {program "SinAxes2"}
```

SinClip

```
program SinClip(output);
import dgl_lib;                              {get graphics routines}
const
  CrtAddr=            3;                     {address of internal CRT}
  ControlWord=        0;                     {device control; 0 for CRT}
type
  RoundType=          (Up, Down, Near);          {used by function Round2}
var
  CharWidth:          real;                  {width of char is world coords}
  CharHeight:         real;                  {height of char is world coords}
  Text:               string[20];            {temporary holding place for text}
  ErrorReturn:        integer;               {variable for initialization outcome}
  X:                  integer;
  Y:                  real;
  ClipXmin, ClipXmax:   real;                {soft clip limits in x}
  ClipYmin, ClipYmax:   real;                {soft clip limits in y}
$include 'DGLPRG:DataPoint'$                {function: y:=f(x) }
$page$  {*****************************************************************}
procedure ClipLimit(Xmin, Xmax, Ymin, Ymax: real);
{-----------------------------------------------------------------------------}
{  This procedure defines the four global variables which specify where the  }
{  soft clip limits are.                                                      }
{-----------------------------------------------------------------------------}
begin
if Xmin<Xmax then begin                      { \                            }
  ClipXmin:=Xmin;                            {  \    Force the minimum soft  }
  ClipXmax:=Xmax;                            {   \   clip limit in X to be   }
end                                          {    \  the smaller of the two  }
else begin                                   {    /  X values passed into    }
  ClipXmin:=Xmax;                            {   /   the procedure.          }
  ClipXmax:=Xmin;                            {  /                            }
end;                                         { /                             }
if Ymin<Ymax then begin                      { \                             }
  ClipYmin:=Ymin;                            {  \    Force the minimum soft  }
  ClipYmax:=Ymax;                            {   \   clip limit in Y to be   }
end                                          {    \  the smaller of the two  }
else begin                                   {    /  Y values passed into    }
  ClipYmin:=Ymax;                            {   /   the procedure.          }
  ClipYmax:=Ymin;                            {  /                            }
end;                                         { /                             }
end;
$page$  {*****************************************************************}
procedure ClipDraw(X1, Y1, X2, Y2: real);
{-----------------------------------------------------------------------------}
{    This procedure takes the endpoints of a line, and clips it.  The soft    }
{  clip limits are the real global variables ClipXmin, ClipXmax, ClipYmin,    }
{  and ClipYmax.  These may be defined through the procedure ClipLimit.       }
{-----------------------------------------------------------------------------}
label
  1;
type
  Edges=        (Left,Right,Top,Bottom);         {possible edges to cross}
  OutOfBounds=  set of Edges;                    {set of edges crossed}
var
  Out,Out1,Out2:OutOfBounds;
  X, Y:         real;
```

```
{---------------------------------------------------------------------------}
procedure Code(X, Y: real; var Out: OutOfBounds);
begin                                   {nested procedure "Code"}
Out:=[];                                {null set}
if x<ClipXmin then Out:=[left]            {off left edge?}
else if x>ClipXmax then Out:=[right];     {off right edge?}
if y<ClipYmin then Out:=Out+[bottom]      {off the bottom?}
else if y>ClipYmax then Out:=Out+[top];   {off the top?}
end;                                    {nested procedure "Code"}
{---------------------------------------------------------------------------}
begin                                   {body of procedure "ClipDraw"}
Code(X1,Y1,Out1);                       {figure status of point 1}
Code(X2,Y2,Out2);                       {figure status of point 2}
while (Out1<>[]) or (Out2<>[]) do begin {loop while either point out of range}
  if (Out1*Out2)<>[] then goto 1;       {if intersection non-null, no line}
  if Out1<>[] then Out:=Out1
    else Out:=Out2;                     {Out is the non-empty one}
  if left in Out then begin             {it crosses the left edge}
    y:=Y1+(Y2-Y1)*(ClipXmin-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmin;                        {new x is left edge}
  end  {left in Out?}
  else if right in Out then begin       {it crosses right edge}
    y:=Y1+(Y2-Y1)*(ClipXmax-X1)/(X2-X1);{adjust value of y appropriately}
    x:=ClipXmax;                        {new x is right edge}
  end  {right in Out?}
  else if bottom in Out then begin      {it crosses the bottom edge}
    x:=X1+(X2-X1)*(ClipYmin-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmin;                        {new y is bottom edge}
  end  {bottom in Out?}
  else if top in Out then begin         {it crosses the top edge}
    x:=X1+(X2-X1)*(ClipYmax-Y1)/(Y2-Y1);{adjust value of x appropriately}
    y:=ClipYmax;                        {new y is top edge}
  end;  {top in Out?}
  if Out=Out1 then begin
    X1:=x;   Y1:=y;   Code(x,y,Out1);   {redefine first end point}
  end  {Out=Out1?}
  else begin
    X2:=x;   Y2:=y;   Code(x,y,Out2);   {redefine second end point}
  end;  {else begin}
end;  {while}
move(x1,y1);                            {if we get to this point, the line...}
line(x2,y2);                            {...is completely visible, so draw it}
1:   end;  {procedure "ClipDraw"}       {return}
$page$  {*******************************************************************}
function Round2(N, M: real; Mode: RoundType): real;
{---------------------------------------------------------------------------}
{  This function rounds "N" to the nearest "M", according to "Mode".  This  }
{  function works only when the argument is in the range of MININT..MAXINT.  }
{---------------------------------------------------------------------------}
const
  epsilon=      1E-10;                  {roundoff error fudge factor}
var
  Rounded:      real;                   {temporary holding area}
  Negative:     boolean;                {flag: "It is negative?"}
```

```
begin                                   {body of "Round2"}
Negative:=(N<0.0);                      {is the number negative?}
if Negative then begin
  N:=abs(N);                            {work with a positive number}
  if Mode=Up then Mode:=Down            {if number is negative, ...}
  else if Mode=Down then Mode:=Up;      {...reverse up and down}
end;
case Mode of                            {should we round the number...}
  Down: Rounded:=trunc(N/M)*M;          {...left on the number line?}
  Up:   begin
          Rounded:=N/M;                 {...right on the number line?}
          if abs(Rounded-round(Rounded))>epsilon then
            Rounded:=(trunc(Rounded)+1.0)*M
          else
            Rounded:=trunc(Rounded)*M;
        end;
  Near: Rounded:=trunc(N/M+M*0.5)*M;    {...to the nearest multiple?}
end;  {case}
if Negative then Rounded:=-Rounded;     {reinstate the sign}
Round2:=Rounded;                        {assign to function name}
end;                                    {function "Round2"}
$page$  {*****************************************************************}
procedure XaxisClip(Spacing, Location: real; Major: integer;
  Majsize,Minsize: real);
{------------------------------------------------------------------------------}
{  This procedure draws an X-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                       }
{    Spacing:   The distance between tick marks on the axis.                 }
{    Location:  The Y-value of the X-axis.                                   }
{    Major:     The number of tick marks to ge before drawing a major tick   }
{               mark.  If Major=5, every fifth tick mark will be major.      }
{    Majsize:   The length, in world units, of the major tick marks.         }
{    Minsize:   The length, in world units, of the minor tick marks.         }
{------------------------------------------------------------------------------}
var
  X:            real;
  SemiMajsize:  real;
  SemiMinsize:  real;
  Counter:      integer;                {keeps track of when to do major ticks}
begin                                   {body of procedure "XaxisClip"}
SemiMajsize:=MajSize*0.5;
SemiMinsize:=MinSize*0.5;
Counter:=0;                             {start with a major tick}
ClipDraw(ClipXmin,Location,ClipXmax,Location);
X:=Round2(ClipXmin,Spacing*Major,Down); {round to next lower major}
while X<=ClipXmax do begin
  if Counter=0 then
    ClipDraw(X,Location-SemiMajsize,X,Location+SemiMajsize)
  else
    ClipDraw(X,Location-SemiMinsize,X,Location+SemiMinsize);
  Counter:=(Counter+1) mod Major;
  X:=X+Spacing;
end;  {while}
end;                                    {procedure "XaxisClip"}
```

```
$page$  {*****************************************************************}
procedure YaxisClip(Spacing, Location: real; Major: integer;
  Majsize, Minsize: real);
{--------------------------------------------------------------------------}
{  This procedure draws an Y-axis at any intersection point on the plotting  }
{  surface.  Parameters are as follows:                                     }
{    Spacing:   The distance between tick marks on the axis.               }
{    Location:  The X-value of the Y-axis.                                 }
{    Major:     The number of tick marks to ge before drawing a major tick  }
{               mark.  If Major=5, every fifth tick mark will be major.    }
{    Majsize:   The length, in world units, of the major tick marks.       }
{    Minsize:   The length, in world units, of the minor tick marks.       }
{--------------------------------------------------------------------------}
var
  Y:            real;
  SemiMinsize:  real;
  SemiMajsize:  real;
  Counter:      integer;              {Keeps track of when to do major ticks}
begin                                 {body of procedure "YaxisClip"}
SemiMajsize:=Majsize*0.5;
SemiMinsize:=Minsize*0.5;
Counter:=0;                           {start with a major tick}
ClipDraw(Location,ClipYmin,Location,ClipYmax);
Y:=Round2(ClipYmin,Spacing*Major,Down); {round to next lower major}
while Y<=ClipYmax do begin
  if Counter=0 then
    ClipDraw(Location-SemiMajsize,Y,Location+SemiMajsize,Y)
  else
    ClipDraw(Location-SemiMinsize,Y,Location+SemiMinsize,Y);
  Counter:=(Counter+1) mod Major;
  Y:=Y+Spacing;
end;  {while}
end;                                  {procedure "YaxisClip"}
$page$  {*****************************************************************}
begin                                 {program "SinClip"}
graphics_init;                        {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin           {output device initialization OK?}
  set_aspect(511,389);                {use the whole screen}
  CharWidth:=2*0.04;                  {char width: 4% of screen width}
  CharHeight:=2*0.08;                 {char height: 8% of screen height}
  set_char_size(CharWidth,CharHeight);  {install the character size}
  Text:='VOLTAGE VARIANCE';           {define the text to be labelled}
  for X:=-3 to 3 do begin             {make "bold" label}
    move(-(strlen(Text)*CharWidth)/2+X*0.002,0.9);      {center label}
    gtext(Text);                      {label the text}
  end;
  set_text_rot(0,1);                  {vertical labels}
  CharWidth:=2*0.025;                 {char width: 2.5% of screen width}
  CharHeight:=2*0.04;                 {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='Voltage';                    {define text to be labelled}
  move(-0.9,-(strlen(Text)*CharWidth)/2);       {start point of centered label}
  gtext(Text);                        {label the text}
  Text:='Time (seconds)';             {define text to be labelled}
  set_text_rot(1,0);                  {horizontal labels}
  move(-(strlen(Text)*CharWidth)/2,-0.92);      {start point of centered label}
  gtext(Text);                        {label the text}
```

```
  set_viewport(0.1,0.99,0.12,0.7);      {define subset of window}
  move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);          {scale the window for the data}
  ClipLimit(0,100,0.16,0.18);           {define the soft clip limits}
  XaxisClip(1,0.16,5,0.0008,0.0004);    {draw the clipped X-axis}
  YaxisClip(0.001,0,5,2,1);             {draw the clipped Y-axis}
  for X:=1 to 100 do begin              {100 points total}
    Y:=DataPoint(X);                    {get a point from the function}
    if X=1 then move(X,Y)               {move to the first point...}
    else line(X,Y);                     {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                          {terminate the graphics package}
end.                                    {program "SinClip"}
```

SinLabel1

```
program SinLabel1(output);
import dgl_lib, dgl_inq;                {get graphics routines}
const
  CrtAddr=          3;                  {address of internal CRT}
  ControlWord=      0;                  {device control; 0 for CRT}
var
  ErrorReturn:  integer;                {variable for initialization outcome}
  Strng:        string[7];              {seven characters in 'Voltage'}
  Character:    integer;                {loop counter for labelling}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$            {function: y:=f(x) }
$page$  {*****************************************************************}
begin                                   {body of program "SinLabel1"}
graphics_init;                          {initialize graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin             {output device initialization OK?}
  set_aspect(511,389);                  {use the whole screen}
  move(-0.45,0.9);                      {starting point for the title}
  gtext('VOLTAGE VARIANCE');            {label the plot}
  Strng:='Voltage';                     {the y-axis label}
  move(-0.95,0.3);                      {starting point for the y-axis title}
  for Character:=1 to strlen(Strng) do  {follow every character...}
    gtext(str(Strng,Character,1)+chr(13)+chr(10));      {...with a CR/LF}
  move(-0.3,-0.9);                      {starting point for the x-axis label}
  gtext('Time (seconds)');              {x-axis label}
  set_viewport(0.1,0.99,0.12,0.7);      {define subset of screen}
  move(-1,-1); line(1,-1); line(1,1); line(-1,1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);          {scale the window for the data}
  for X:=1 to 100 do begin              {100 points total}
    Y:=DataPoint(X);                    {get a point from the function}
    if X=1 then move(X,Y)               {move to the first point...}
    else line(X,Y);                     {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                          {terminate the graphics package}
end.                                    {program "SinLabel1"}
```

SinLabel2

```
program SinLabel2(output);
import dgl_lib, dgl_inq;                {get graphics routines}
const
  CrtAddr=       3;                     {address of internal CRT}
  ControlWord=   0;                     {device control; 0 for CRT}
var
  CharWidth:     real;                  {width of character in world coords}
  CharHeight:    real;                  {height of character in world coords}
  Text:          string[20];            {temporary holding place for text}
  ErrorReturn:   integer;               {variable for initialization outcome}
  X:             integer;
  Y:             real;
$include 'DGLPRG:DataPoint'$            {function: y:=f(x) }
$page$  {*************************************************************}
begin                                   {body of program "SinLabel2"}
graphics_init;                          {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin             {output device initialization OK?}
  set_aspect(511,389);                  {use the whole screen}
  CharWidth:=2*0.04;                    {char width: 4% of screen width}
  CharHeight:=2*0.08;                   {char height: 8% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='VOLTAGE VARIANCE';             {define the text to be labelled}
  move(-(strlen(Text)*CharWidth)/2,0.9);{go to start point for centered label}
  gtext(Text);                          {label the text}
  set_text_rot(0,1);                    {vertical labels}
  CharWidth:=2*0.025;                   {char width: 2.5% of screen width}
  CharHeight:=2*0.04;                   {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='Voltage';                      {define the text to be labelled}
  move(-0.9,-(strlen(Text)*CharWidth)/2);       {start point of centered label}
  gtext(Text);                          {label the text}
  set_text_rot(1,0);                    {horizontal labels}
  Text:='Time (seconds)';               {define the text to be labelled}
  move(-(strlen(Text)*CharWidth)/2,-0.92);      {start point of centered label}
  gtext(Text);                          {label the text}
  set_viewport(0.1,0.99,0.12,0.7);      {define subset of screen}
  move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);          {scale the window for the data}
  for X:=1 to 100 do begin              {100 points total}
    Y:=DataPoint(X);                    {get a point from the function}
    if X=1 then move(X,Y)               {move to the first point...}
    else line(X,Y);                     {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                          {terminate the graphics package}
end.                                    {program "SinLabel2"}
```

SinLabel3

```
program SinLabel3(output);
import dgl_lib, dgl_inq;                {get graphics routines}
const
  CrtAddr=      3;                      {address of internal CRT}
  ControlWord=  0;                      {device control; 0 for CRT}
var
  CharWidth:    real;                   {width of character in world coords}
  CharHeight:   real;                   {height of character in world coords}
  Text:         string[20];             {temporary holding place for text}
  ErrorReturn:  integer;                {variable for initialization outcome}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$            {function: y:=f(x) }
$page$  {*************************************************************}
begin                                   {body of program "SinLabel3"}
graphics_init;                          {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin             {output device initialization OK?}
  set_aspect(511,389);                  {use the whole screen}
  CharWidth:=2*0.04;                    {char width: 4% of screen width}
  CharHeight:=2*0.08;                   {char height: 8% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='VOLTAGE VARIANCE';             {define the text to be labelled}
  for X:=-3 to 3 do begin               {make "bold" label}
    move(-(strlen(Text)*CharWidth)/2+X*0.002,0.9);      {center label}
    gtext(Text);                        {label the text}
  end;  {for X}
  set_text_rot(0,1);                    {vertical labels}
  CharWidth:=2*0.025;                   {char width: 2.5% of screen width}
  CharHeight:=2*0.04;                   {char height: 4% of screen height}
  set_char_size(CharWidth,CharHeight);  {install character size}
  Text:='Voltage';                      {define the text to be labelled}
  move(-0.9,-(strlen(Text)*CharWidth)/2);       {start point of centered label}
  gtext(Text);                          {label the text}
  set_text_rot(1,0);                    {horizontal labels}
  Text:='Time (seconds)';               {define the text to be labelled}
  move(-(strlen(Text)*CharWidth)/2,-0.92);      {start point of centered label}
  gtext(Text);                          {label the text}
  set_viewport(0.1,0.99,0.12,0.7);      {define subset of screen}
  move(-1,-1); line(-1,1); line(1,1); line(1,-1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);          {scale the window for the data}
  for X:=1 to 100 do begin              {100 points total}
    Y:=DataPoint(X);                    {get a point from the function}
    if X=1 then move(X,Y)               {move to the first point...}
    else line(X,Y);                     {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                          {terminate the graphics package}
end.                                    {program "SinLabel3"}
```

SinLine

```
program SinLine(output);
import dgl_lib;                          {get graphics routines}
const
  CrtAddr=      3;                       {address of internal CRT}
  Control=      0;                       {device control; 0 for CRT}
var
  ErrorReturn:  integer;                 {variable for initialization outcome}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$             {function: y:=f(x) }
$page$  {*********************************************************************}
begin                                    {body of program "SinLine"}
graphics_init;                           {initialize graphics system}
display_init(CrtAddr,Control,ErrorReturn);      {which output device?}
if ErrorReturn=0 then begin              {output device initialization OK?}
  for X:=1 to 100 do begin               {100 points total}
    Y:=DataPoint(X);                     {get a point from the function}
    if X=1 then move(X/100,Y)            {move to the first point...}
    else line(X/100,Y);                  {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                           {terminate the graphics package}
end.                                     {program "SinLine"}
```

SinViewpt

```
program SinViewpt(output);
import dgl_lib;                          {get graphics routines}
const
  CrtAddr=          3;                   {address of internal CRT}
  ControlWord=      0;                   {device control; 0 for CRT}
var
  ErrorReturn:  integer;                 {variable for initialization outcome}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$             {function: y:=f(x) }
$page$  {*********************************************************************}
begin                                    {body of program "SinViewpt"}
graphics_init;                           {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin              {output device initialization OK?}
  set_aspect(511,389);                   {use the whole screen}
  set_viewport(0.10,0.99,0.12,0.70);     {define subset of screen}
  move(-1,-1); line(1,-1); line(1,1); line(-1,1); line(-1,-1);  {frame}
  set_window(0,100,0.16,0.18);           {scale the window for the data}
  for X:=1 to 100 do begin               {100 points total}
    Y:=DataPoint(X);                     {get a point from the function}
    if X=1 then move(X,Y)                {move to the first point...}
    else line(X,Y)                       {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                           {terminate the graphics package}
end.                                     {program "SinViewpt"}
```

SinWindow

```
program SinWindow(output);
import dgl_lib;                       {get graphics routines}
const
  CrtAddr=          3;                {address of internal CRT}
  ControlWord=      0;                {device control; 0 for CRT}
var
  ErrorReturn:  integer;              {variable for initialization outcome}
  X:            integer;
  Y:            real;
$include 'DGLPRG:DataPoint'$          {function: y:=f(x) }
{*****************************************************************************}
begin                                 {body of program "SinWindow"}
graphics_init;                        {initialize the graphics system}
display_init(CrtAddr,ControlWord,ErrorReturn);  {which output device?}
if ErrorReturn=0 then begin           {output device initialization OK?}
  set_window(0,100,0.16,0.18);        {scale the window for the data}
  for X:=1 to 100 do begin            {100 points total}
    Y:=DataPoint(X);                  {get a point from the function}
    if X=1 then move(X,Y)             {move to the first point...}
    else line(X,Y)                    {...and draw to all the rest}
  end;  {for X:=1 to 100}
end;  {ErrorReturn=0?}
graphics_term;                        {terminate the graphics package}
end.                                  {program "SinWindow"}
```

Graphics Procedure Reference

Appendix **B**

The Pascal Programming Language was designed as a teaching language, and as such was intended to be machine independent. This attribute has its good and bad points. Being machine independent makes the language more easily transportable, but also ensures that it is difficult, if not impossible, to access any innovative hardware features provided by a specific computer system.

To allow easy access to the graphics and I/O features of your Pascal system, a set of procedures and functions are provided in the LIBRARY file on the SYSVOL: disc. This reference describes the syntax and semantics for the procedures and functions provided to access graphics.

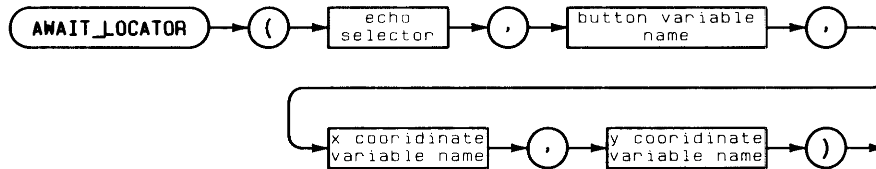
The small block of text labelled IMPORT, immediately below the title of each entry, lists the module which must be declared in an IMPORT statement in order to access the feature. Modules which are needed by these imported modules, if any, are shown in the Module Dependency Table at the end of this reference.

AWAIT_LOCATOR

IMPORT: dgl_lib

This **procedure** waits until activation of the locator button and then reads from the enabled locator device. Various echo methods can be selected.

Syntax



Item	Description/Default	Range Restrictions
echo selector	Expression of TYPE INTEGER	MININT to MAXINT
button variable name	Variable of TYPE INTEGER	–
x coordinate name	Variable of TYPE REAL	–
y coordinate name	Variable of TYPE REAL	–

Procedure Heading

```
PROCEDURE AWAIT_LOCATOR (     Echo    : INTEGER;
                          VAR Button  : INTEGER;
                          VAR WX, WY  : REAL     );
```

Semantics

AWAIT_LOCATOR waits until the locator button is activated and then returns the value of the selected button and the world coordinates of the locator. While the button press is awaited, the locator position can be tracked on the graphic display device. If an invalid button is pressed, the button value will be returned as 0; otherwise it will contain the value of the button pressed. On locators that use a keyboard for the button device (e.g. HP 9826 / HP 9836), the ordinal value of the key pressed is returned.

The **echo selector** selects the type of echo used. Possible values are:

0 - No echo.
1 - Echo on the locator device.
2 - Small cursor
3 - Full cross hair cursor
4 - Rubber band line
5 - Horizontal rubber band line
6 - Vertical rubber band line
7 - Snap horizontal / vertical rubber band line
8 - Rubber band box
9 and above - Device dependent echo on the locator device.

Locator input can be echoed on either a graphics display device or a locator device. The meaning of the various echoes on various devices used as locators and displays is discussed below.

The **button value** is the INTEGER value of the button used to terminate the locator input.

The **x** and **y position** represent the world coordinate point returned from the enabled locator.

AWAIT_LOCATOR implicitly makes the picture current before sending any commands to the locator device. The locator should be enabled (LOCATOR_INIT) before calling AWAIT_LOCATOR. The locator is terminated by the procedure LOCATOR_TERM.

Range and Limit Considerations

If the echo selector is out of range, the call to AWAIT_LOCATOR is completed using an echo selector of 1 and no error is reported. Echoes 2 through 8 require a graphics display to be enabled. If a display is not enabled, the call will be completed with echo 1 and GRAPHICSERROR will return 4.

If the point entered is outside of the current logical locator limits, the transformed point will still be returned in world coordinates.

Starting Position Effects

The location of the starting position is device dependent after this procedure with echo 0 or echo 1. For soft-copy devices it is typically unchanged; however, for plotters the pen position (starting position) will remain at the last position it was moved to by the operator. This is done to reduce pen movement back to the current position after each AWAIT_LOCATOR invocation.

Echo Types

Several different types of echoing can be performed. Some echoes are performed on the locator device while others use the graphics display device. When the echo selector is in the range 2 thru 8, the graphics display device will be used in echoing. All of the echoes on the graphics display start at a point on the graphics display called the locator echo position (see SET_ECHO_POS). For some of these echoes the locator echo position is also used as a fixed reference point. For example, the fixed end of the rubber band line will be at the locator echo position. The echoes available are:

2. Small cursor
   Track the position of the locator on the graphics display device. The initial position of the cursor is at the locator echo position. The point returned is the locator position.
3. Full cross hair cursor
   Designate the position of the locator on the graphics display device with two intersecting lines. One line is horizontal with a length equal to the width of the logical display surface. The other line is vertical with a length equal to the height of the logical display surface. The initial point of intersection is at the current locator echo position. The point returned is the locator position.
4. Rubber band line
   Designate the endpoints of a line. One end is fixed at the locator echo position; the other is designated by the current locator position. The locator position can be told from the locator echo position by the presence of a small cursor (echo 2) at end representing the locator echo position. The point returned is the locator position.

5. Horizontal rubber band line
   Designate a horizontal line. One endpoint of the line is fixed at the locator echo position; the other endpoint has the world Y-coordinate of the locator echo position and the world X-coordinate of the current locator position. The locator position can be told from the locator echo position by the presence of a small cursor (echo 2) at end representing the locator echo position. The point returned will have the X-coordinate of the locator position and the Y-coordinate of the locator echo position.
6. Vertical rubber band line
   Designate a vertical line. One endpoint of the line is fixed at the locator echo position; the other endpoint will have the world X-coordinate of the locator echo position and the world Y-coordinate of the current locator position. The locator position can be told from the locator echo position by the presence of a small cursor (echo 2) at end representing the locator echo position. The point returned will have the X-coordinate of the locator echo position and the Y-coordinate of the locator position.
7. Snap horizontal / vertical rubber band line
   Designate a horizontal / vertical line. One endpoint of the line is fixed at the locator echo position. The other endpoint will be either a horizontal (see echo 5) or vertical (see echo 6) rubber band line, depending on which one produces the longer line. If both lines are of equal length, a horizontal line will be used. The locator position can be told from the locator echo position by the presence of a small cursor (echo 2) at end representing the locator echo position. The point returned is the endpoint of the echoed line.
8. Rubber band box
   Designate a rectangle. The diagonal of the rectangle is the line from the locator echo position to the current locator position. The locator position can be told from the locator echo position by the presence of a small cursor (echo 2) at end representing the locator echo position. The point returned will be the locator position.

Echo selectors of 1 and greater than or equal to 9 produce a device dependent echo on the locator device. Most locator devices support at least one form of echoing. Possible ones include beeping, displaying the value entered, or blinking a light each time a point is entered. If the specified echo is not supported on the enabled locator device, echo 1 will be used.

Echoes on Raster Displays
Raster displays support all the echoes described under "Echo Types."

Echoes on HPGL Plotters
Hard copy plotting devices (such as the 9872 or the 7580) cannot perform all the echoes listed above. The closest approximation possible is used for simulating them. The actual echo performed may also depend on whether the plotter is also being used as the locator. The echoes available on plotters are:

2. Small cursor

   Initially the plotter's pen will be moved to the locator echo position. The pen will then reflect the current locator position (i.e., track) until the locator operation is terminated.
3. Full cross hair cursor
   Simulated by ECHO #2.
4. Rubber band line

   Simulated by ECHO #2.

5. Horizontal rubber band line
   If the plotter is **not** the current locator device, the plotter's pen will initially be moved to the current locator echo position. The pen will then reflect the X coordinate of the current locator position and the Y coordinate of the current locator echo position.

   If the plotter is used as the locator, this echo is simulated by echo 2 except the current locator X coordinate and the locator echo position Y coordinate are returned.
6. Vertical rubber band line
   If the plotter is **not** the current locator device, the plotter's pen position will initially be moved to the current locator echo position. The pen will then reflect the X coordinate of the current locator echo position and the Y coordinate of the current locator position.

   If the plotter is used as the locator, this echo is simulated by echo 2 except the locator echo position X coordinate and the current locator Y coordinate are returned.
7. Snap horizontal / vertical rubber band line
   Designate a horizontal / vertical line. One endpoint of the line is fixed at the locator echo position. The other endpoint will be either a horizontal (see echo 5) or vertical (see echo 6) rubber band line, depending on which one produces the longer line. If both lines are of equal length, a horizontal line will be used. The locator position can be told from the locator echo position by the presence of a small cursor (echo 2) at end representing the locator echo position. The point returned is the endpoint of the echoed line.
8. Rubber band box
   Simulated by echo 2. The point returned will be the locator position.

Absolute Locators (Graphics Tablet or Plotter)
For HPGL graphics tablets the operator positions the stylus to the desired position and depresses it. The button value returned is always one. For an echo selector of 1 the tablet beeper is sounded when the stylus is depressed. An echo selector greater than or equal to 9 uses the same echo as an echo selector of 1. (Some HPGL plotters have the ability of using the physical pen as a locator. See the subsequent section called "HPGL Plotters as Absolute Locators" for details.)

Relative Locators (Knob or Mouse)
When the knob is specified as the locator (LOCATOR_INIT with device selector of 2) the keyboard keys have the following meanings:

Arrow keys	Move the cursor in the direction indicated.
Knob	Move the cursor right and left.
Knob with shift key pressed	Move the cursor up and down.
Mouse	Move the cursor in the direction of mouse movement (mouse left = cursor left; mouse forward = cursor up; etc.).
Number keys $1 \rightarrow 9$	Change the amount the cursor is moved per arrow keypress or knob rotation. 1 provides the least movement and 9 provides the most.

All other keys act as the locator buttons. The ordinal value of the locator button (key) struck is returned in BUTTON.

For an echo selector of 1 the position of the locator is indicated by a small crosshair cursor on the graphics display.

The initial position of the cursor is located at the current starting position of the graphics display. This is the point obtained by the last invocation of await_locator, or the lower left hand corner of the locator limits if no point has been received since LOCATOR_INIT was executed. For back to back AWAIT_LOCATOR calls this would mean the second AWAIT_LOCATOR would begin were the first AWAIT_LOCATOR left the cursor. Echo selectors greater than or equal to 9 have the same effect as an echo selector of 1.

Locator input can be echoed on either a graphics display device or a locator device. Echoes 2 thru 8 are explained above under "Echoes on Raster Displays" and "Echoes on HPGL Plotters". For an echo selector of 0 or 1 the pen tracks the locator position. Echo selectors greater than or equal to 9 have the same effect as an echo selector of 1.

HPGL Plotters as Absolute Locators

The AWAIT_LOCATOR function enables a digitizing mode in the device. For HPGL plotters the operator then positions the pen to the desired position with the cursor buttons or joy stick and then presses the enter key. The pen state (0 for 'up', and 1 for 'down') is returned in the button parameter.

Following locator input (echo on the locator device), the pen position will remain at the last position it was moved to by the operator. This means that the starting position for the next graphics primitive will be wherever the pen was left.

Locator input can be echoed on either a graphics display device or a locator device. Echoes 2 thru 8 are explained above under "Echoes on Raster Displays" and "Echoes on HPGL Plotters". For an echo selector of 0 or 1 the pen tracks the locator position. Echo selectors greater than or equal to 9 have the same effect as an echo selector of 1.

Error Conditions

The graphics system must be initialized and the locator device must be enabled or the call will be ignored. If the echo selector is between 1 and 9 and the graphics display is not enabled, the call will be completed with an echo selector of 1. If any of the preceding errors are encountered, an ESCAPE (-27) is generated, and GRAPHICSERROR will return a non-zero value.

CLEAR_DISPLAY

IMPORT: dgl_lib

This **procedure** clears the graphics display.

Syntax

→(CLEAR_DISPLAY)→

Procedure Heading

```
PROCEDURE CLEAR_DISPLAY;
```

Semantics

The graphics system provides the capability to clear the graphics display of all output primitives at any time in an application program. This procedure has different meaning for different graphics display devices. CLEAR_DISPLAY makes the picture current. The starting position is not effected by this procedure.

HPGL Plotters

Plotters with page advance will be sent a command to advance the paper. On devices such as fixed page plotters, a call to CLEAR_DISPLAY simply makes the picture current.

Raster Displays

On CRT displays, this procedure clears the display to the background color. This means slightly different things on different displays:

Monochrome	If color table location 0 is 0 then the display is cleared to black. Otherwise, the display is cleared to white.
HP 98627A	The display is cleared to the non-dithered color closest to the color represented specified by color table location 0. (e.g., If color table location 0 was Red = .5, Green = .2, Blue = 0, the display would be cleared to red.)
HP Model 36C	The display is cleared to the color represented by color table location 0.

Error conditions:

The graphics system must be initialized and a display must be enabled or the call will be ignored, an ESCAPE (− 27) will be generated, and the GRAPHICSERROR function will return a non-zero value.

CONVERT_WTODMM

IMPORT: dgl_lib

This **procedure** converts from world coordinates to millimetres on the graphics display.

Syntax

Item	Description/Default	Range Restrictions
world x	Expression of TYPE REAL	–
world y	Expression of TYPE REAL	–
metric x name	Variable of TYPE REAL	–
metric y name	Variable of TYPE REAL	–

Procedure Heading

```
PROCEDURE CONVERT_WTODMM (     WX, WY    : REAL;
                           VAR MmX, MmY  : REAL );
```

Semantics

This procedure returns a coordinate pair (**metric X**,**metric Y**) representing the **world X** and **Y** coordinates. The metric X and Y values are the number of millimetres along the X and Y axis from the supplied world coordinate point to the origin of the metric coordinate system on the device. The location of this origin is device dependent.

For raster devices, the metric origin is the lower-left dot. For HPGL plotters, it is the lower-left corner of pen movement.

Since the origin of the world coordinate system need not correspond to the origin of the physical graphics display, converting the point (0.0,0.0) in the world coordinate system may not result in the value (0.0,0.0) offset from the physical display device's origin.

CONVERT_WTODMM will take any world coordinate point, inside or outside the current window, and convert it to a point offset from the physical display device's origin.

Error conditions:

The graphics system must be initialized and the graphics display must be enabled or the call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR will return a non-zero value.

CONVERT_WTOLMM

IMPORT: dgl_lib

This **procedure** converts from world coordinates to millimetres on the locator surface.

Syntax

Item	Description/Default	Range Restrictions
world x	expression of TYPE REAL	–
world y	expression of TYPE REAL	–
metric x name	variable of TYPE REAL	–
metric y name	variable of TYPE REAL	–

Procedure Heading

```
PROCEDURE CONVERT_WTOLMM (      WX, WY   : REAL;
                            VAR MmX, MmY : REAL );
```

Semantics

This procedure returns a coordinate pair (**metric x**,**metric y**) representing the **world X** and **Y** coordinates. The metric x and y values are the number of millimetres along the X and Y axis from the supplied world coordinate point to the origin of the metric coordinate system on the device. The location of this origin is device dependent.

For raster devices, the metric origin is the lower-left dot. For HPGL plotters, it is the lower-left corner of pen movement.

Since the origin of the world coordinate system need not correspond to the origin of the physical locator device, converting the point (0.0,0.0) in the world coordinate system does not necessarily result in the value (0.0,0.0) offset from the physical locator device's origin.

CONVERT_WTOLMM will take any world coordinate point, inside or outside the current window, and convert it to a point offset from the physical locator origin.

Error Conditions

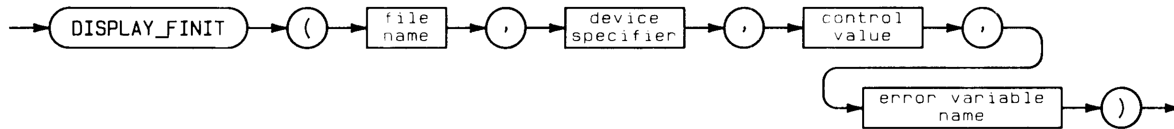
The graphics system must be initialized, the graphics device must be enabled, and the locator must be initialized or the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

DISPLAY_FINIT

IMPORT: dgl_lib

This **procedure** enables the output of the graphics library to be sent to a file.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
file name	Expression of TYPE *Gstring255*; can be a STRING of any length up to 255 characters.	Must be a valid file name (see "The File System")	–
device specifier	Expression of TYPE *Gstring255*; can be a STRING of any length up to 255 characters. First six characters are significant.	9872A, 9872B, 9872C, 9872S, 9872T, 7470A, 7475A, 7550A and 7586B	–
control value	Expression of TYPE INTEGER	MININT thru MAXINT	see below
error variable name	Variable of TYPE INTEGER	–	–

Procedure Heading

```
PROCEDURE DISPLAY_FINIT (     File_Name   : Gstring255,
                                Device_Name: Gstring255,
                                Control    : INTEGER,
                            var Ierr       : INTEGER );
```

Semantics

DISPLAY_FINIT allows output from the graphics library to be sent to a file. This file can then be sent a graphics display device by use of the operating system's file system (e.g. FILER, or SRM spooler). The contents of the file are device dependent, and MUST be sent only to devices of the type indicated in device name when the file was created.

The **file name** specifies the name of the file to send device dependent commands to.

The **device specifier** tells the graphics system the type of device that the file will be sent to. Only some types of devices may be use this command. For example raster devices (i.e. the internal display) may not use this command. For the currently supported devices, see the range restrictions under Syntax, above.

The **control value** is used to control characteristics of the graphics display device and should be set according to the display device the file is intended for. See "Control Values," below, for the meaning of the control value.

The **error variable name** will contain a value indicating whether the graphics display device was successfully initialized.

Value	Meaning
0	The graphics display device was successfully initialized.
1	The graphics display device (indicated by *device name*) is not supported by the graphics library.
2	Unable to open the file specified. File error is returned in Escapecode, and Ioresult (see the Pascal Language System User's manual).

DISPLAY_FINIT enables a file as the logical graphics display. The file can be of any type, although the current spooling mechanisms can only handle TEXT and ASCII files. The file need not exist before this procedure is called. If this procedure is successful the file will be closed with 'LOCK' when DISPLAY_TERM is executed.

This procedure initializes and enables the graphics display for graphics output. Before the device is initialized the device status is 0, the device address is 0, and the device name is the default name. The default name is ' ' (six ASCII blanks).

When the device is enabled the device status is set to 1 (enabled) and the internal device specifier used by the graphics library is set to the file name provided by the user. The device name is set to the supplied device name. This information is available by calling INQ_WS with operation selectors of 11050 and 12050.

Initialization includes the following operations:

- The graphics display surface is cleared (e.g., CRT erased, plotter page advanced) if Bit 7 of CONTROL is not set.
- The starting position is set to a device dependent location.
- The logical display limits are set to the default limits for the device.
- The aspect ratio of the virtual coordinate system is applied to the logical display limits to define the limits of the virtual coordinate system.
- All primitive attributes are set to the default values.
- The locator echo position is set to its default value.

Only one graphics output device can be initialized at a time. If a graphics display device is currently enabled, the enabled device will be terminated (via DISPLAY_TERM) and the call will continue.

A call to MOVE or INT_MOVE should be made after this call to update the starting position and in so doing, place the physical pen or beam at a known location on the graphics display device.

The Control Value

The control value is used to control characteristics of the graphics display device. Bits should be set according to the following bit map. All unused bits should be set to 0.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits	Meaning
0 thru 6	Currently unused. Should be set to 0.
7	If this bit is set (BIT 7 = 1), it will inhibit clearing of the graphics display as part of the DISPLAY_FINIT procedure. Some devices have the ability to **not** clear the graphics display, or **not** to perform a page advance during device initialization. This bit is ignored on devices that do not support the feature.
8 thru 15	Not used by DISPLAY_FINIT.

HPGL Plotter Initialization

When an HPGL device is initialized the following device dependent actions are performed, in addition to the general initialization process:

- Pen velocity, force, and acceleration are set to the default for that device.
- ASCII character set is set to 'ANSI ASCII'.
- Paper cutter is enabled (HP 9872S / HP 9872T).
- Advance page option is enabled (HP 9872S / HP 9872T / HP 7550A).
- Paper is advanced one full page (HP 9872S / HP 9872T / HP 7550A) (unless DISPLAY_INIT CONTROL bit 7 is set).
- The automatic pen options are set (HP 7580 / HP 7585 / HP 7586B / HP 7550A).

The default initial dimensions for the HPGL plotters supported by the graphics library are:

Plotter	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
9872	400	285	16000	11400	.7125	40.0
7580	809.5	524.25	32380	20970	.6476	40.0
7585	1100	891.75	44000	35670	.8107	40.0
7586	1182.8	898.1	47312	35924	.7593	40.0
7470	257.5	191.25	10300	7650	.7427	40.0
7550	411.25	254.25	16450	10170	.6182	40.0
7475	416	259.125	16640	10365	.6229	40.0

Any device not in this list is **not** supported.

The default logical display surface is set equal to the maximum physical limits of the device. The view-surface is always justified in the lower left corner of the current logical display surface (corner nearest the turret for the HP 7580 and HP 7585 plotters). The physical origin of the graphics display is at the lower left boundary of pen movement.

Error Conditions

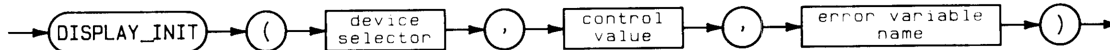
If the graphics system is not initialized, the call is ignored, an ESCAPE (– 27) is generated, and GRAPHICSERROR returns a non-zero value.

DISPLAY_INIT

IMPORT: dgl_lib

This **procedure** enables a device as the logical graphics display.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
device selector	Expression of TYPE INTEGER	MININT to MAXINT	
control value	Expression of TYPE INTEGER	MININT to MAXINT	–
error variable name	Variable of TYPE INTEGER	–	–

Procedure Heading

```
PROCEDURE DISPLAY_INIT (     Dev_Adr  : INTEGER,
                             Control  : INTEGER,
                         VAR IErr     : INTEGER );
```

Semantics

DISPLAY_INIT enables a device as the logical graphics display. It initializes and enables the graphics display device for graphics output.

Before the device is initialized the device status is 0, the device address is 0, and the device name is the default name. The default name is ' ' (six ASCII blanks).

When the device is enabled the device status is set to 1 (enabled) and the internal device specifier used by the graphics library is set equal to the device selector provided by the user. The device name is set to the device being used. This information is available by calling INQ_WS with operation selectors 11050 and 12050.

The **device selector** specifies the physical address of the graphics output device.

- address =3 Primary internal graphics CRT (HP Series 200) (i.e., the display designated as the console where the command line is displayed)
- address =6 Secondary internal graphics CRT (HP Series 200), if present (i.e., any display other than the console that does not require a select code and/or bus address to access it)
- 8 <= device selector <= 31 Interface Card Select Code (HP 98627A default = 28)
- 100 <= device selector <= 3199 composite HPIB/device address

The **control value** is used to control device dependent characteristics of the graphics display device.

The **error variable name** will contain a value indicating whether the graphics display device was successfully initialized.

Value	Meaning
0	The graphics display device was successfully initialized.
2	Unrecognized device specified. Unable to communicate with a device at the specified address, non-existent interface card or non-graphics system supported interface card.

If an error is encountered, the call will be ignored.

The graphics library attempts to directly identify the type of device by using its device selector in some way. The meanings for device address are listed above.

At the time that the graphics library is initialized, all devices which are to be used must be connected, powered on, ready, and accessible via the supplied device selector. Invalid device selectors or unresponsive devices result in that device not being initialized and an error being returned.

Only one graphics output device maybe initialized at a time. If a graphics display device is currently enabled, the enabled device will be terminated (via DISPLAY_TERM) and the call will continue.

A call to MOVE or INT_MOVE should be made after this call to update the starting position and in so doing, place the physical pen or beam at a known location on the graphics display device.

The Control Value

Used to control characteristics of the graphics display device. Bits should be set according to the following bit map. All unused bits should be set to 0.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits	Meaning
0 thru 6	Currently unused. Should be set to 0.
7	If this bit is set (BIT 7 = 1), it will inhibit clearing of the graphics display as part of the DISPLAY_FINIT procedure. Some devices have the ability to **not** clear the graphics display, or **not** to perform a page advance during device initialization. This bit is ignored on devices that do not support the feature.
8 thru 15	Bits 8 though 15 are used by some devices to control device dependent features of those devices.

Bits 8,9, and 10 of DISPLAY_INIT's CONTROL parameter determine the type of display for the HP 98627A card and the default dimensions assumed by the graphics system.

CONTROL	Bits 10 9 8	Description	
256	001	US STD	(512 x 390, 60 hz refresh)
512	010	EURO STD	(512 x 390, 50 hz refresh)
768	011	US TV	(512 x 474, 15.75 Khz horizontal refresh, interlaced)
1024	100	EURO TV	(512 x 512, 50 hz vertical refresh, interlaced)
1280	101	HI RES	(512 x 512, 60 hz)
1536	110	Internal	(HP) use only

Out of range values are treated as if CONTROL = 256.

When using a Model 237 display that is designated the console, bit 8 of DISPLAY_INIT's CONTROL parameter determines if the entire screen will be used for graphics. A value of 256 (i.e., bit 8 = 1) turns off the echo of the type-ahead buffer, and allocates the entire screen for graphics. The type-ahead buffer echo is re-enabled by the DISPLAY_TERM procedure call.

General Initialization Operations
Initialization includes the following operations:

- The graphics display surface is cleared (e.g., CRT erased, plotter page advanced) unless Bit 7 of the control value is set.
- The starting position is set to a device dependent location. (This is undefined for HPGL plotters.)
- The logical display limits are set to the default limits for the device.
- The aspect ratio of the virtual coordinate system is applied to the logical display limits to define the limits of the virtual coordinate system.
- All primitive attributes are set to the default values.
- The locator echo position is set to its default value.
- If the display and locator are the same physical device, the logical locator limits are set to the limits of the view surface.

Raster Display Initialization
When a raster display is initialized the following device dependent actions are performed, in addition to the general initialization process:

- The starting position is in the lower left corner of the display.
- Graphics memory is cleared if bit 7 of the control word is 0.
- Initialize the color table to default values. If the device has retroactive color definition (Model 36C) and the color table has been changed from the default colors, the colors of an image will change even if bit 7 is set to 1.
- The graphics display is turned on.
- The view surface is centered within the logical display limits.

- The drawing mode (see OUTPUT_ESC) is set to dominate.
- The DISPLAY_INIT CONTROL parameter is used as specified above.

The following table describes the internal raster displays for Series 200 computers:

Computer	Wide mm	High mm	Wide points	High points	Memory Planes	Color Map
Model 216	160	120	400	300	1	no
Model 217	230	175	512	390	1	no
Model 220 (HP82913A)	210	158	400	300	1	no
Model 220 (HP82912A)	152	114	400	300	1	no
Model 226	120	88	400	300	1	no
Model 236	210	160	512	390	1	no
Model 236 Color	217	163	512	390	4	yes
Model 237	312	234	1024	768	1	no

The HP 98627A is a 3 plane non-color mapped color interface card which connects to an external RGB monitor. Bits 8,9, and 10 of DISPLAY_INIT's CONTROL parameter determine the type of display for the HP 98627A card and the default dimensions assumed by the graphics system.

CONTROL	Bits 10 9 8	Description	
256	001	US STD	(512 x 390, 60 hz refresh)
512	010	EURO STD	(512 x 390, 50 hz refresh)
768	011	US TV	(512 x 474, 15.75 Khz horizontal refresh, interlaced)
1024	100	EURO TV	(512 x 512, 50 hz vertical refresh, interlaced)
1280	101	HI RES	(512 x 512, 60 hz)
1536	110	Internal	(HP) use only

Out of range values are treated as if CONTROL = 256.

The physical size of the HP 98627A display (needed by the SET_DISPLAY_LIM procedure) may be given to the graphics system by an escape function (OPCODE = 250). The physical limits assumed until the escape function is given are:

CONTROL =	256	153.3mm wide and 116.7mm high.
	512	153.3mm wide and 116.7mm high.
	768	153.3mm wide and 142.2mm high.
	1280	153.3mm wide and 153.3mm high.

The default logical display surface of the graphics display device is the maximum physical limits of the screen. The physical origin is the lower left corner of the display.

The view surface is always centered within the current logical display surface.

HPGL Plotter Initialization

When an HPGL device is initialized the following device dependent actions are performed, in addition to the general initialization process:

- Pen velocity, force, and acceleration are set to the default for that device.
- ASCII character set is set to 'ANSI ASCII'.
- Paper cutter is enabled (HP 9872S / HP 9872T).
- Advance page option is enabled (HP 9872S / HP 9872T / HP 7550A /HP 7586B).
- Paper is advanced one full page (HP 9872S / HP 9872T / HP 7550A / HP 7586B) (unless DISPLAY_INIT CONTROL bit 7 is set).
- The automatic pen options are set (HP 7580 / HP 7585).

The default initial dimensions for the HPGL plotters supported by the graphics library are:

Plotter	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
9872	400	285	16000	11400	.7125	40.0
7580	809.5	524.25	32380	20970	.6476	40.0
7585	1100	891.75	44000	35670	.8107	40.0
7586	1182.8	898.1	47312	35924	.7593	40.0
7470	257.5	191.25	10300	7650	.7427	40.0
7550	411.25	254.25	16450	10170	.6182	40.0
7475	416	259.125	16640	10365	.6229	40.0

The maximum physical limits of the graphics display for an HPGL device not listed above are determined by the default settings of P1 and P2. The default settings of P1 and P2 are the values they have after an HPGL 'IN' command. Refer to the specific device manual for additional details.

The default logical display surface is set equal to the area defined by P1 and P2 at the time DISPLAY_INIT is invoked. The view surface is always justified in the lower-left corner of the current logical display surface (corner nearest the turret for the HP 7580, HP 7585 and HP 7586 plotters). The physical origin of the graphics display is at the lower-left boundary of pen movement.

Note

If the paper is changed in an HP 7580, HP 7585 or HP 7586 plotter while the graphics display is initialized, it should be the same size of paper that was in the plotter when DISPLAY_INIT was called. If a different size of paper is required, the device should be terminated (DISPLAY_TERM) and re-initialized after the new paper has been placed in the plotter.

Error Conditions

The graphics system must be initialized or the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

DISPLAY_TERM

IMPORT: dgl_lib

This **procedure** disables the enabled graphics display device.

Syntax

Procedure Heading

```
PROCEDURE DISPLAY_TERM;
```

Semantics

DISPLAY_TERM terminates the device enabled as the graphics display. DISPLAY_TERM completes all remaining display operations and disables the logical graphics display. It makes the picture current and releases all resources being used by the device. The device name is set to the default name ' ' (six ASCII blanks), the device status is set to 0 (not enabled) and the device address is set to 0. DISPLAY_TERM does not clear the graphics display.

The graphics display device should be disabled before the termination of the application program. DISPLAY_TERM is the complementary routine to DISPLAY_INIT.

Error Conditions

The graphics system should be initialized and the display should be enabled or the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

GRAPHICSERROR

IMPORT: dgl_lib

This **function** returns and integer error code and can be used to determine the cause of a graphics escape.

Syntax

→(GRAPHICSERROR)→

Function Heading

```
FUNCTION GRAPHICSERROR: INTEGER;
```

Semantics

When an error occurs that uses the escape function, escape-code −27 is used. After the escape is trapped and it has been determined that the graphics library is the source of the error (the escape code equal to −27), GRAPHICSERROR can be used to determine the cause of the error. The function returns the value of the last error generated and then clears the value of the return error. A user who is trapping errors and wishes to keep the value of the error must save it in some variable.

The following list of returned values and the error they represent can be used to interpret the value returned by GRAPHICSERROR.

Value	Meaning
0	No errors since the last call to GRAPHICSERROR or since the last call to GRAPHICS_INIT.
1	The graphics system is not initialized. ACTION: CAll ignored.
2	The graphics display is not enabled. ACTION: Call ignored.
3	The locator device is not enabled. ACTION: Call ignored.
4	Echo value requires a graphics display to be enabled. ACTION: Call completes with echo value = 1.
5	The graphics system is already initialized. ACTION: Call ignored.
6	Illegal aspect ratio specified. X-SIZE and Y-SIZE must be greater than 0. ACTION: Call ignored.
7	Illegal parameters specified. ACTION: Call ignored.
8	The parameters specified are outside the physical display limits. ACTION: Call ignored.
9	The parameters specified are outside the limits of the window. ACTION: Call ignored.
10	The logical locator and the logical display are the same physical device. The logical locator limits cannot be defined explicitly, they must correspond to the logical view surface limits. ACTION: Call ignored.

11	The parameters specified are outside the current virtual coordinate system boundary. ACTION: Call ignored.
13	The parameters specified are outside the physical locator limits. ACTION: Call ignored.
14	Color table contents cannot be inquired or changed. ACTION: Call ignored.
18	The number of points specified for a polygon or polyline operation is less than or equal to zero. ACTION: Call ignored.

GRAPHICS_INIT

IMPORT: dgl_lib

This **procedure** initializes the graphics system.

Syntax

Procedure Heading

```
PROCEDURE GRAPHICS_INIT;
```

Semantics

GRAPHICS_INIT initializes the graphics system. It must be the first graphics system call made by the application program. Any procedure call other than GRAPHICS_INIT will be ignored. GRAPHICS_INIT performs the following operations:

- Get dynamic storage space for the graphics library.
- Sets the aspect ratio to 1.
- Sets the virtual coordinate and viewport limits to range from 0 to 1.0 in the X and Y directions.
- Sets the world coordinate limits to range from −1.0 to 1.0 in the X and Y directions.
- Sets the starting position to (0.0,0.0) in world coordinate system units.
- Sets all attributes equal to their default values.

GRAPHICS_INIT does not enable any logical devices. The graphics system is terminated with a call to GRAPHICS_TERM. Calling GRAPHICS_INIT while the graphics system is initialized will result in an implicit call to GRAPHICS_TERM, before the system is reinitialized.

GRAPHICS_TERM

IMPORT: dgl_lib

This **procedure** terminates the graphics system.

Syntax

Procedure Heading

```
PROCEDURE GRAPHICS_TERM;
```

Semantics

GRAPHICS_TERM terminates the graphics system. Termination includes terminating both the graphics display and the locator devices. GRAPHICS_TERM does not clear the graphics display.

GRAPHICS_TERM should be called as the last graphics system call in the application program.

GRAPHICS_TERM releases dynamic memory allocated during GRAPHICS_INIT. In order that this memory actually be returned the compiler option $HEAP_DISPOSE ON$ must be used.

Error Conditions

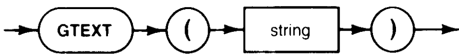
If the graphics system is not initialized, the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

GTEXT

IMPORT: dgl_types dgl_lib

This **procedure** draws characters on the graphics display.

Syntax



Item	Description/Default	Range Restrictions
string	Expression of TYPE *Gstring255*. Can be a string of any length up to 255 characters	length <= 255 characters

Procedure Heading

```
PROCEDURE GTEXT ( String : Gstring255 );
```

Semantics

The **string** contains the characters to be output.

GTEXT produces characters on the graphics display. A series of vectors representing the characters in the string is produced by the graphics system.

When the text string is output, the starting position will represent the lower left-hand corner of the first character in STRING. Text is normally output from left to right and is printed vertically with no slant.

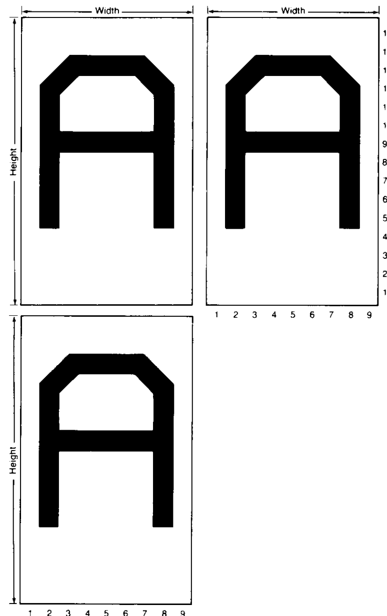
After completion of this call, the starting position is left in a device dependent location such that successive calls to GTEXT will produce a continuous line of text (i.e., `GTEXT('H'); GTEXT('I');` is equivalent to `GTEXT('HI');`).

The attributes of color, line-style, line-width, text rotation, and character size apply to text primitives. However, the text will appear with these attributes only if the graphics device is capable of applying them to text.

Characters

The character sets provided by the graphics system are the same ones used by the CRT in alpha mode, namely the standard character set plus either the Roman extension character set (for all non-Katakana machines) or the Katakana character set (for Katakana machines).

Characters are defined within a cell that has an aspect ratio of 9/15. The character cells are adjacent, both horizontally and vertically, as shown here.



Control Codes

The following control codes are supported by GTEXT:

Control Character	Program Access	Keyboard Access	Action
backspace	CHR(8)	CTRL-H	Move one character cell to the left along the text direction vector (defined by SET_CHAR_SIZE).
linefeed	CHR(10)	CTRL-J	Move down the height of one character cell.
carriage return	CHR(13)	CTRL-M	Move back the length of the text just completed.

Any other control characters are ignored.

The current position is maintained to the resolution of the display device. A text size less-than-or-equal-to the resolution of the display device will result in all the characters in a GTEXT call, or a series of GTEXT calls, being written to the same point on the device.

The current position returned by an INQ_WS is **not** updated by calls to GTEXT. If you want to know the current position after a GTEXT, you must do a MOVE, or some other call which updates the current position.

Error Conditions

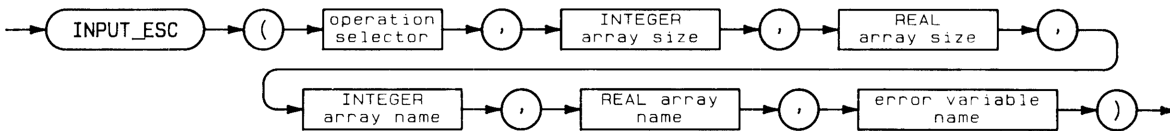
If the graphics system is not initialized or a display is not enabled, the call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR will return a non-zero value.

INPUT_ESC

IMPORT: dgl_lib

This **procedure** allows the user to obtain device dependent information from the graphics system.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
operation selector	Expression of TYPE INTEGER	MININT to MAXINT	–
INTEGER array size	Expression of TYPE INTEGER	MININT to MAXINT	>0
REAL array size	Expression of TYPE INTEGER	MININT to MAXINT	>0
INTEGER array name	Variable of TYPE ANYVAR should be array of INTEGERs	–	–
REAL array name	Variable of TYPE ANYVAR should be array of REAL	–	–
error variable name	Variable of TYPE INTEGER	–	–

Procedure Heading

```
PROCEDURE INPUT_ESC (        Opcode : INTEGER;
                     Isize : INTEGER;
                     Rsize : INTEGER;
                     ANYVAR Ilist : Gint_list;
                     ANYVAR Rlist : Greal_list;
                     VAR Ierr  : INTEGER    );
```

Semantics

The **operation selector** determines the device dependent inquiry escape function being invoked.

The **INTEGER array size** is the number of INTEGER parameters to be returned in the INTEGER array by the escape function. The correct value for this can be found in the hundred's place of the operation selector (see the table below).

The **REAL array size** is the number of REAL parameters to be returned in the REAL array by the escape function. The correct value for this can be found in the thousand's place of the operation selector (see the table below).

The **INTEGER array** is the array in which zero or more INTEGER parameters are returned by the escape function.

The **REAL array** is the array in which zero or more REAL parameters are returned by the escape function.

The **error variable** will contain a code indicating whether the input escape function was performed.

Value	Meaning
0	Inquiry escape function successfully completed.
1	Inquiry operation (operation selector) not supported by the graphics display device.
2	INTEGER array size is not equal to the number of INTEGER parameters to be returned.
3	REAL array size is not equal to the number of REAL parameters to be returned.

If the error variable contains a non-zero value, the call has been ignored.

INPUT_ESC allows application programs to access special device features on a graphics display device. The type of information returned from the graphics display device is determined by the value of operation selector. Possible inquiry escape functions may return the status or the options supported by a particular graphics display device.

Inquiry escape functions only apply to the graphics display device. INPUT_ESC implicitly makes the picture current before the escape function is performed.

HPGL Plotter Operation Selectors

The following inguiry is supported:

Operation Selector	Meaning
2050	Inquire about current turret. INTEGER array [1] = −1 >> Turret mounted, but its type is unknown INTEGER array [1] = 0 >> No turret mounted INTEGER array [1] = 1 >> Fiber tip pens INTEGER array [1] = 2 >> Roller ball pens INTEGER array [1] = 3 >> Capillary pens INTEGER array [2] = 0 >> No turret mounted or turret has no pens INTEGER array [2] = n >> Sum of these values: 1: Pen in stall #1 2: Pen in stall #2 4: Pen in stall #3 8: Pen in stall #4 16: Pen in stall #5 32: Pen in stall #6 64: Pen in stall #7 128: Pen in stall #8

For example, if INTEGER array[2] = 3, pens would only be contained in stalls 1 and 2.

Operation selector 2050 is supported on the HP 7475, HP 7550, HP 7580, HP 7585 and HP 7586 plotters.

Error Conditions

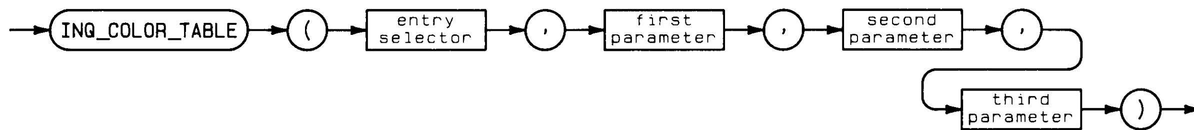
If the graphics system is not initialized or a display is not enabled, the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

INQ_COLOR_TABLE

IMPORT: dgl_lib
dgl_inq

This **procedure** inquires the color modeling parameters for an index into the device-dependent color capability table.

Syntax



Item	Description/Default	Range Restrictions
entry selector	Expression of TYPE INTEGER	>0
first parameter name	Variable of TYPE REAL	–
second parameter name	Variable of TYPE REAL	–
third parameter name	Variable of TYPE REAL	–

Procedure Heading

```
PROCEDURE INQ_COLOR_TABLE (     Index : INTEGER;
                            VAR Colp1 : REAL;
                            VAR Colp2 : REAL;
                            VAR Colp3 : REAL     );
```

Semantics

This routine inquires the color modelling parameters for the specified location in a device-dependent color capability table.

The **entry selector** specifies the location in the color capability table. The parameters returned are for the specified location. The size of the color capability table is device dependent. For raster displays in Series 200 computers, 32 entries are available.

The **first parameter** represents red intensity if the RGB model has been selected with the SET COLOR statement, or hue if the HSL model has been selected.

The **second parameter** represents green intensity if the RGB model has been selected with the SET COLOR statement, or saturation if the HSL model has been selected.

The **third parameter** represents blue intensity if the RGB model has been selected, or luminosity if the HSL model has been selected.

A more detailed description of the color models and the meaning of their parameters can be found under the procedure definition of SET_COLOR_MODEL.

Note

The color table stores color specifications as RGB values. The conversion from RGB to HSL is a one-to-many transformation, and the following arbitrary assignments may be made during the conversion:

```
IF Luminosity = 0
   THEN   Hue = 0
          Saturation = 0

IF Saturation = 0
   THEN   Hue = 0
```

Error Conditions

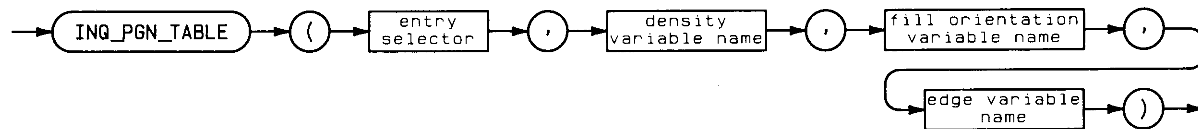
If the graphics system is not initialized, a display device is not enabled, the color table contents cannot be inquired, or the color table entry selector is out of range, the call is ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

INQ_PGN_TABLE

IMPORT: dgl_lib
dgl_inq

This **procedure** inquires the polygon style attributes for an entry in the polygon style table.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
entry selector	Expression of TYPE INTEGER	MININT thru MAXINT	Device dependent
density variable name	Variable of TYPE REAL	–	–
fill orientation variable name	Variable of TYPE REAL	–	–
edge variable name	Variable of TYPE INTEGER	–	–

Procedure Heading

```
PROCEDURE INQ_PGN_TABLE (     Index   : INTEGER;
                          VAR Densty  : REAL;
                          VAR Orient  : REAL;
                          VAR Edge    : INTEGER );
```

Semantics

The **entry selector** specifies the entry in the polygon style table the inquiry is directed at.

The **density variable** will contain a value between -1 and 1. This magnitude of this value is the ratio of filled area to non-filled area. Zero means the polygon interior is not filled. One represents a fully filled polygon interior. All non-zero values specify the density of continuous lines used to fill the interior. Negative values are used to specify crosshatching. Calculations for fill density are based on the thinnest line possible on the device and on continuous line-style. If the interior line-style is not continuous, the actual fill density may not match that found in the polygon style table.

The **fill orientation variable** will contain a value from -90 through 90. This value represents the angle (in degrees) between the lines used for filling the polygon and the horizontal axis of the display device. The interpretation of fill orientation is device-dependent. On devices that require software emulation of polygon styles, the angle specified will be adhered to as closely as possible, within the line-drawing capabilities of the device. For hardware generated polygon styles, the angle specified will be adhered to as closely as is possible given the hardware simulation of the requested density. If crosshatching is specified, the fill orientation specifies the angle of orientation of the first set of lines in the crosshatching, and the second set of lines is always perpendicular to this.

The **edge variable** will contain a 0 if the polygon edge is not to be displayed and a 1 if the polygon edge is to be displayed. If polygon edges are displayed, they adhere to the current line attributes of color, line-style, and line-width, in effect at the time of polygon display.

All current devices support 16 entries in the polygon table. The polygon styles defined in the default tables are defined to exploit the hardware capabilities of the devices they are defined for.

Error Conditions

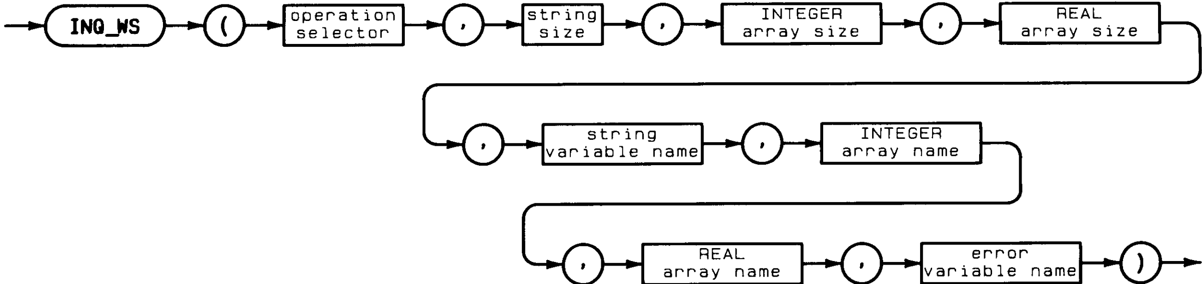
The graphics system must be initialized, a display must be enabled, and the entry selector must be in range or the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

INQ_WS

IMPORT: dgl_lib
dgl_inq

This **procedure** allows the user to determine characteristics of the graphics system.

Syntax



Item	Description/Default	Range Restrictions
operation selector	Expression of TYPE INTEGER	see below
string size	Expression of TYPE INTEGER	see below
integer array size	Expression of TYPE INTEGER	see below
REAL array size	Expression of TYPE INTEGER	see below
string name	Variable of TYPE PACKED ARRAY OF CHAR	–
INTEGER array name	Variable of TYPE ARRAY OF INTEGER	–
REAL array name	Variable of TYPE ARRAY OF REAL	–
error variable name	Variable of TYPE INTEGER	–

Procedure Heading

```
PROCEDURE INQ_WS (        Opcode  : INTEGER;
                          Ssize   : INTEGER;
                          Isize   : INTEGER;
                          Rsize   : INTEGER;
                   ANYVAR Slist   : Gchar_list;
                   ANYVAR Ilist   : Gint_list;
                   ANYVAR Rlist   : Greal_list;
                   VAR    Ierr    : INTEGER);
```

Semantics

The **operation selector** is an integer from the list of operation selectors given below. It is used to specify the topic of the inquiry to the system.

The **string size** is used to specify the maximum number of characters that are to be returned in the string array by the function specified by the operation selector. If there is a 1 in the ten-thousand's place a string value will be returned. The number of characters in the string is returned in the first entry in the INTEGER arrray.

The **INTEGER array size** is the number of integer parameters that are returned in the integer array by the function specified by OPCODE. The thousand's digit of the operation selector is the number of elements the INTEGER array must contain.

The **REAL array size** is the number of REAL parameters that are returned in the REAL array by the function specified by OPCODE. The hundred's digit of the operation selector is the number of elements the REAL array must contain.

The **string array** is a PACKED ARRAY OF CHAR which will contain a string or strings that represents characteristics of the work station specified by the value of operation selector. The application program must ensure that string array is dimensioned to contain all of the values returned by the selected function.

The **INTEGER array** will contain integer values that represent characteristics of the work station specified by the value of OPCODE. The application program must ensure that the integer array is dimensioned to contain all of the values returned by the selected function.

The **REAL array** will contain REAL values that represent characteristics of the work station specified by the value of OPCODE. The application program must ensure that the REAL array is dimensioned to contain all of the values returned by the selected function.

The **error variable** will return an integer indicating whether the inquiry was successfully performed.

Value	Meaning
0	The inquiry was successfully performed.
1	The operation selector was invalid.
2	The INTEGER array size was not equal to the number INTEGER parameters requested by the operation selector.
3	The REAL array size was not equal to the number of REAL parameters requested by the operation selector.
4	The string array was not large enough to hold the string requested by the operation selector.

The procedure INQ_WS returns current information about the graphics system to the application program. The type of information desired is specified by a unique value of OPCODE. The thousands digit of the operation selector specifies the number of integer values returned in the integer array and the hundreds digit specifies the number of REAL values returned in the REAL array. A 1 in the ten-thousand's place indicates that a value will be returned in the string.

One use of INQ_WS is device optimization: the use of inquiry to enhance the application's utilization of the output device. An example of this is using color to distinguish between lines when a device supports colors, and using line-styles when color is not available. Another example is maximizing the aspect ratio used, based on the maximum aspect ratio of the display device.

Device dependent information returned by the procedure is undefined if the device being inquired from is not enabled (e.g., inquire number of colors supported, operation selector 1053, only returns valid information when the display is enabled).

If the graphics system is not initialized, the call will be ignored and GRAPHICSERROR will return a non-zero value.

Supported Operation Selectors

The operation selectors supported by the system and their meaning is listed below:

Operation Selector	Meaning
250	Current cell size used for text. REAL Array[1] = Character cell width in world coordinates REAL Array[2] = Character cell height in world coordinates
251	Marker size. REAL Array[1] = Marker width in world coordinates REAL Array[2] = Marker height in world coordinates
252	Resolution of graphics display REAL Array[1] = Resolution in X direction (points/mm) REAL Array[2] = Resolution in Y direction (points/mm)
253	Maximum dimensions of the graphics display. REAL Array[1] = Maximum size in X direction (MM) REAL Array[2] = Maximum size in Y direction (MM)
254	Aspect ratios REAL Array[1] = Current aspect ratio of the virtual coordinate system. REAL Array[2] = Aspect ratio of logical limits.
255	Resolution of locator device REAL Array[1] = Resolution in X direction (points/mm) REAL Array[2] = Resolution in Y direction (points/mm)
256	Maximum dimensions of the locator display. REAL Array[1] = Maximum size in X direction (MM) REAL Array[2] = Maximum size in Y direction (MM)
257	Current locator echo position REAL array[1] = X world coordinate position REAL array[2] = Y world coordinate position
258	Current virtual coordinate limits REAL array[1] = Maximum X virtual coordinate REAL array[2] = Maximum Y virtual coordinate
259	Starting position. The information returned may not be valid (not updated) following a text call, an escape function call, changes to the viewing transformation or after initialization of the graphics display device. REAL array[1] = X world coordinate position REAL array[2] = Y world coordinate position
450	Current window limits REAL array[1] = Minimum X world coordinate position REAL array[2] = Maximum X world coordinate position REAL array[3] = Minimum Y world coordinate position REAL array[4] = Maximum Y world coordinate position
451	Current viewport limits REAL array[1] = Minimum X virtual coordinate REAL array[2] = Maximum X virtual coordinate REAL array[3] = Minimum Y virtual coordinate REAL array[4] = Maximum Y virtual coordinate

Operation Selector	Meaning
1050	Does graphics display device support clipping at physical limits? INTEGER Array[1] = 0 - No INTEGER Array[1] = 1 - Yes, to the view-surface boundaries INTEGER Array[1] = 2 - Yes, but only to the physical limits of the display surface.
1051	Justification of the view surface within the logical display limits. INTEGER Array[1] = 0 - View-surface is centered within the logical display limits INTEGER Array[1] = 1 - View surface is positioned in the lower left corner of the logical display limits.
1052	Can the graphics display draw in the background color? Drawing in the background color can be used to 'erase' previously drawn primitives. INTEGER Array[1] = 0 - No INTEGER Array[1] = 1 - Yes
1053	The total number of non-dithered colors supported on the graphics display. The number returned does not include the background color. (Compare operation selectors 1053, 1054, and 1075.) INTEGER Array[1] = number of distinct colors supported.
1054	Number of distinct non-dithered colors which can appear on the graphics display at one time. The number returned does not include the background color. INTEGER Array[1] = number of distinct colors which can appear on the display device at one time.
1056	Number of line-styles supported on the graphics display. INTEGER Array[1] = number of hardware line-styles supported.
1057	Number of line-widths supported on the graphics display. INTEGER Array[1] = number of line-widths supported.
1059	Number of markers supported on the graphics display. INTEGER Array[1] = # of distinct markers supported.
1060	Current value of color attribute. INTEGER Array[1] = Current value of color attribute.
1062	Current value of line-style attribute INTEGER Array[1] = Current value of line-style attribute.
1063	Current value of line-width attribute. INTEGER Array[1] = Current value.
1064	Current timing mode. INTEGER Array[1] = 0 - Immediate visibility INTEGER Array[1] = 1 - System buffering
1065	Number of entries in the polygon style table. INTEGER Array[1] = # styles.
1066	Current polygon interior color index. INTEGER Array[1] = Index

Operation Selector	Meaning
1067	Current polygon style index. INTEGER Array[1] = Index
1068	Maximum number of polygon vertices that a display device can process. INTEGER Array[1] = 0 — No hardware support. = N (0<n<32767) — Number of vertices supported. = 32767 — The graphics display device uses all available memory to process polygons (the maximum number of vertices is determined by current free memory).
1069	Does the graphics device support immediate, retroactive change of polygon style for polygons already displayed? INTEGER Array[1] = 0 - No. INTEGER Array[1] = 1 - Yes.
1070	Does the graphics device support hardware (or low-level device handler) generation of polygons using INT_POLYGON_DD? INTEGER Array[1] = 0 - No INTEGER Array[1] = 1 - Yes
1071	Does the graphics device support immediate, retroactive change for primitives already displayed? INTEGER Array[1] = 0 - No INTEGER Array[1] = 1 - Yes
1072	Can the background color of the display be changed? INTEGER Array[1] = 0 - No INTEGER Array[1] = 1 - Yes
1073	Can entries in the color table be redefined using SET_COLOR_TABLE? INTEGER Array[1] = 0 - No INTEGER Array[1] = 1 - Yes
1074	Current color model in use. INTEGER Array[1] = 1 - RGB INTEGER Array[1] = 2 - HSL
1075	Number of entries in the color capability table. The number returned does not include the background color. INTEGER Array[1] = # entries
1076	Current polygon interior line-style. INTEGER Array[1] = Current interior line-style
11050	Graphics display device association. String = Name of device path. (Internal device specifier.) INTEGER Array[1] = Number of characters in the device path.
11052	Locator device association. String = Name of device path. (Internal device specifier.) INTEGER Array[1] = Number of characters in the device path.

Operation Selector	Meaning
12050	Graphics display device information. String = Name of graphics display device. INTEGER Array[1] = Number of characters in the device name. INTEGER Array[2] = Status = 0 Graphics display is not enabled. = 1 Graphics display is enabled.
13052	Graphics locator device information. String = Name of the locator device. INTEGER Array[1] = Number of characters in the device name. INTEGER Array[2] = Status = 0 Locator device is not enabled. = 1 Locator device is enabled. INTEGER Array[3] = Number of buttons on the locator device.

Error Conditions

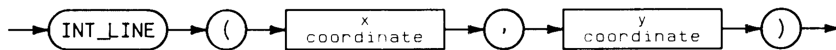
If the graphics system is not initialized, the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

INT_LINE

IMPORT: dgl_types
dgl_lib

This **procedure** draws a line from the starting position to the world coordinate specified.

Syntax



Item	Description/Default	Range Restrictions
x coordinate	Expression of TYPE *Gshortint*. This is subrange of INTEGER	−32 768 to 32 767
y coordinate	Expression of TYPE *Gshortint*. This is subrange of INTEGER	−32 768 to 32 767

Procedure Heading

```
PROCEDURE INT_LINE ( Iwx, Iwy : Gshortint );
```

Semantics

The **x** and **y coordinate** pair is the ending of the line to be drawn in the world coordinate system.

A line is drawn from the starting position to the world coordinate specified by the x and y coordinates. The starting position is updated to this point at the completion of this call.

The primitive attributes of line style (see SET_LINE_STYLE), line width (see SET_LINE_WIDTH), and color (see SET_COLOR) apply to lines drawn using INT_LINE.

This procedure is the same as the LINE procedure, with the exception that the parameters are of type *Gshortint* (−32 768..32 767). When used with some displays this procedure may perform about 3 times faster than the LINE procedure. For all other displays this procedure has about the same performance as the LINE procedure.

The INT_LINE procedure only has increased performance when the following conditions exist:

- The display must be a raster device.
- The window bounds within the range −32 768 to 32 767.
- The window must be less then 32 767 units wide and high.

INT operations are provided for efficient vector generation. Although their use can be mixed with other, non-integer operations, one dot roundoff errors may result with mixed use since different algorithms are used to implement each.

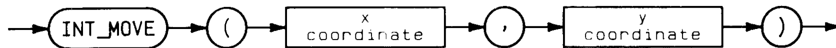
Drawing to the starting position generates the shortest line possible. Depending on the nature of the current line-style, nothing may appear on the graphics display surface. See SET_LINE_STYLE for a complete description of how line-style affects a particular point or vector.

INT_MOVE

IMPORT: dgl_types
dgl_lib

This **procedure** sets the starting position to the world coordinate position specified.

Syntax



Item	Description/Default	Range Restrictions
x coordinate	Expression of TYPE *Gshortint*. This is subrange of INTEGER	−32 768 to 32 767
y coordinate	Expression of TYPE *Gshortint*. This is subrange of INTEGER	−32 768 to 32 767

Procedure Heading

```
PROCEDURE INT_MOVE ( Iwx, Iwy : INTEGER );
```

Semantics

The **x** and **y coordinate** pair define the new starting position in world coordinates.

INT_MOVE specifies where the next graphical primitive will be output. It does this by setting the value of the starting position to the world coordinate system point specified by the x and y coordinate values and then moving the pen (or its logical equivalent) to that point.

The starting position corresponds to the location of the physical pen or beam in all but four instances: after a change in the viewing transformation, after initialization of a graphical display device, after the output of a text string, or after the output of an escape function. A call to MOVE or INT_MOVE should therefore be made after any one of the following calls to update the value of the starting position and in so doing, place the physical pen or beam at a known location: SET_ASPECT, DISPLAY_INIT, SET_DISPLAY_LIM, OUTPUT_ESC, TEXT, SET_VIEWPORT, and SET_WINDOW.

This procedure is the same as the MOVE procedure, with the exception that the parameters are of type *Gshortint* (−32 768..32 767). When used with the same display, this procedure can perform about 3 times faster than the MOVE procedure. For all other displays this procedure has about the same performance as the MOVE procedure.

The INT_MOVE procedure only has increased performance when the following conditions exist:

- The display must be a raster device.
- The window bounds within the range −32 768 to 32 767.
- The window must be less than 32767 units wide and high.

INT operations are provided for efficient vector generation. Although their use can be mixed with non-integer operations, one dot roundoff errors may result with mixed use since different algorithms are used to implement each.

Error Conditions

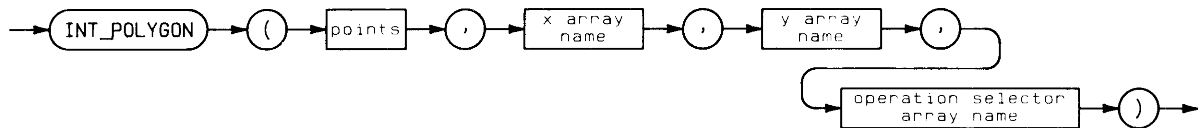
The graphics system must be initialized and a graphics display must be enabled or the call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

INT_POLYGON

IMPORT: dgl_types
dgl_lib
dgl_poly

This **procedure** displays a polygon-set starting and ending at the specified point adhering to the specified polygon style exactly as specified (i.e., device-independent results).

Syntax



Item	Description/Default	Range Restrictions
points	Expression of TYPE INTEGER	MININT thru MAXINT
x array name	Array of TYPE *Gshortint*. *Gshortint* is a subrange of INTEGER.	−32 768 to 32 767
y array name	Array of TYPE *Gshortint*. *Gshortint* is a subrange of INTEGER.	−32 768 to 32 767
operation selector array name	Array of TYPE *Gshortint*. *Gshortint* is a subrange of INTEGER.	−32 768 to 32 767

Procedure Heading

```
PROCEDURE INT_POLYGON (        Npoint   : INTEGER;
                       ANYVAR  Xvec     : Gshortint_list;
                       ANYVAR  Yvec     : Gshortint_list;
                       ANYVAR  Opcodes  : Gshortint_list);
```

Semantics

Points is the number of vertices in the polygon set.

The **x** and **y coordinate arrays** contain the world coordinate values for each vertex of the polygon-set. The vertices must be in order. The vertices for the first sub-polygon must be at the beginning of these arrays, followed by the vertices for the second sub-polygon, etc. So, the coordinate arrays must contain a total number of vertices that equals points.

The **operation selector array** contains a series of integer operation selectors defining which vertices start new polygons, and defining which edges should be displayed.

Value	Meaning
0	Don't display the line for the edge extending to this vertex from the previous vertex.
1	Display the line for the edge extending to this vertex from the previous vertex.
2	This vertex is the first vertex of a sub-polygon. Succeeding vertices are part of a sub-polygon until a new start-of-polygon operation selector (2) is encountered. (Or the end of the arrays is encountered.)

Note

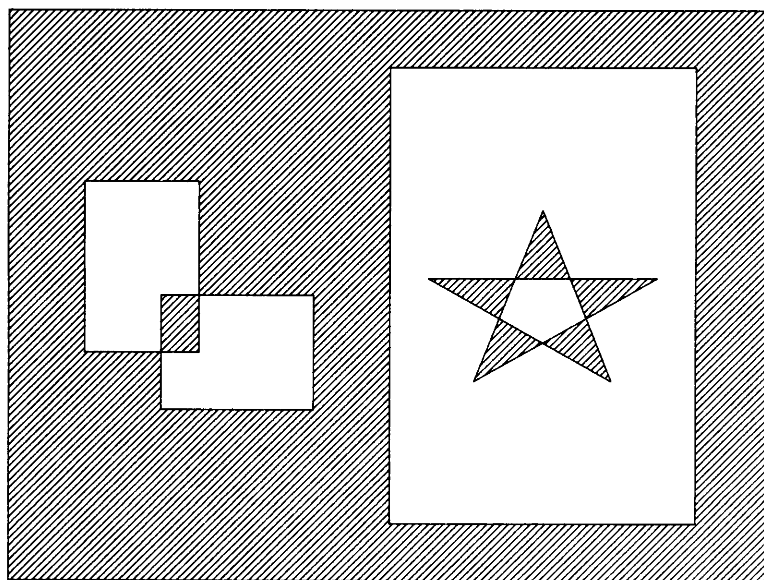
The first entry in the operation selector array **must** be 2, since it is the first vertex of a sub-polygon.

INT_POLYGON is used to output a polygon-set, specified in world coordinates, adhering exactly to the polygon style attributes that are currently specified. A polygon-set is a set of polygons (called "sub-polygons") that are treated graphically as one polygon. This is accomplished by "stacking" the sub-polygons. The subpolygons in a polygon-set may intersect or overlap each other.

The edge of a sub-polygon is defined as the line sequence that connects its vertices in the order specified. If the last vertex specified for a sub-polygon is not the same as the first, they are automatically connected.

When a polygon-set is displayed, the primitive attributes for polygons and lines define its appearance. In particular, the interior of the polygon-set will be filled according to the attributes of polygon style, polygon interior color and polygon interior line-style. If the edges are to be displayed as specified in the polygon style, the edges will adhere to the current line attributes of color, line-style and line-width. A dot will disappear on an edged polygon if the edge is done with a complementing line.

The filling of polygons also depends on how the sub-polygons "nest" within each other. An "even-odd" rule is used for determining which areas will be filled. Moving across the screen, count the edges of the polygon. Odd-numbered edges will turn the fill on and even-numbered edges will turn the fill off. The picture below will help clear up how the fills work.



Polygon Filling

Refer to SET_PGN_TABLE, SET_PGN_STYLE, SET_PGN_COLOR, SET_PGN_LS for a more detailed description of how attributes affect polygons.

As stated above, the values in the operation selector array define how the edges of the sub-polygons are displayed. The edge from the (I-1)th vertex to the Ith vertex will only be displayed if the Ith entry in the operation selector array equals 1. To display the edge from the last vertex to the first vertex of a sub-polygon, the first vertex must be explicitly respecified after all the other vertices of the sub-polygon, with an operation selector equal to 1. Otherwise the edge from the last vertex to the first will not be drawn. It will, however, automatically be connected for polygon filling.

If it is within the capabilities of the device, filling of the sub-polygon will be done to the sub-polygon edges regardless of whether the edges are displayed. If an entry in the operation selector array does not equal 0, 1, or 2, it will be treated as if it were equal to 0 and the edge will not be drawn.

When INT_POLYGON is used, the current position is updated to the end of the last sub-polygon specified in the polygon-set. The end of the last sub-polygon is defined to be the first (implicit last) vertex of the subpolygon. So, if there is only one vertex in a polygon-set this call degenerates to an update of the current position to the first coordinate set in the x and y point arrays (x coordinate array[1], y coordinate array[1]).

It is the application program's responsibility to ensure that the arrays are all dimensioned to at least the number of elements specified by points and that at least that many values are contained in each array.

Polygons are defined to be closed surfaces. When a sub-polygon extends beyond a clipping edge the closed nature of the sub-polygon is destroyed. As with other primitives, unpredictable results may occur if the sub-polygon extends beyond the clipping window.

This procedure is the same as the POLYGON procedure, with the exception that the parameters are of type *Gshortint* (− 32 768..32 767). When used with some displays this procedure may perform about 3 times faster than the POLYGON procedure. For all other displays this procedure has about the same performance as the POLYGON procedure.

The INT_POLYGON procedure only has increased performance when the following conditions exist:

- The display must be a raster device.
- The window bounds are within the range − 32 768 through 32 767.
- The window must be less than 32 767 units wide and high.

INT_POLYGON is provided for efficient vector generation. Although its use can be mixed with MOVE, LINE, POLYLINE, and POLYGON, one dot roundoff errors may result with mixed use since different algorithms are used to implement each.

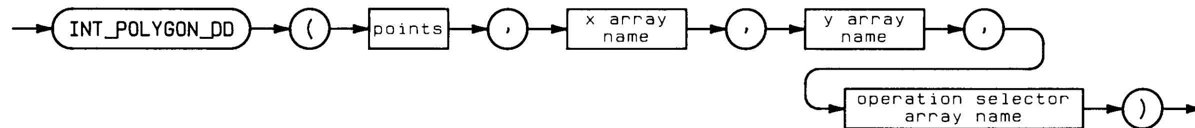
Error Conditions
The graphics system must be initialized, a graphics display must be enabled, all parameters must be within specified limits and the number of points specified must be greater than 0 or the call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

INT_POLYGON_DD

IMPORT: dgl_types
dgl_lib
dgl_poly

This **procedure** displays a polygon-set starting and ending at the specified point adhering to the specified polygon style in a device-dependent fashion.

Syntax



Item	Description/Default	Range Restrictions
points	Expression of TYPE INTEGER	MININT thru MAXINT
x array name	Array of TYPE *Gshortint*. *Gshortint* is a sub-range of INTEGER.	−32 768 to 32 767
y array name	Array of TYPE *Gshortint*. *Gshortint* is a sub-range of INTEGER.	−32 768 to 32 767
operation selector array name	Array of TYPE *Gshortint*. *Gshortint* is a sub-range of INTEGER.	−32 768 to 32 767

Procedure Heading

```
PROCEDURE INT_POLYGON_ DD (        Npoint    : INTEGER;
                           ANYVAR  Xvec      : Gshortint_list;
                           ANYVAR  Yvec      : Gshortint_list;
                           ANYVAR  Opcodes   : Gint_list      );
```

Semantics

Points is the number of vertices in the polygon set.

The **x** and **y coordinate arrays** contain the world coordinate values for each vertex of the polygon-set. The vertices must be in order. The vertices for the first sub-polygon must be at the beginning of these arrays, followed by the vertices for the second sub-polygon, etc. So, the coordinate arrays must contain a total number of vertices that equals points.

The **operation selector array** contains a series of integer operation selectors defining which vertices start new polygons, and defining which edges should be displayed.

Value	Meaning
0	Don't display the line for the edge extending to this vertex from the previous vertex.
1	Display the line for the edge extending to this vertex from the previous vertex.
2	This vertex is the first vertex of a sub-polygon. Succeeding vertices are part of a sub-polygon until a new start-of-polygon operation selector (2) is encountered. (Or the end of the arrays is encountered.)

Note

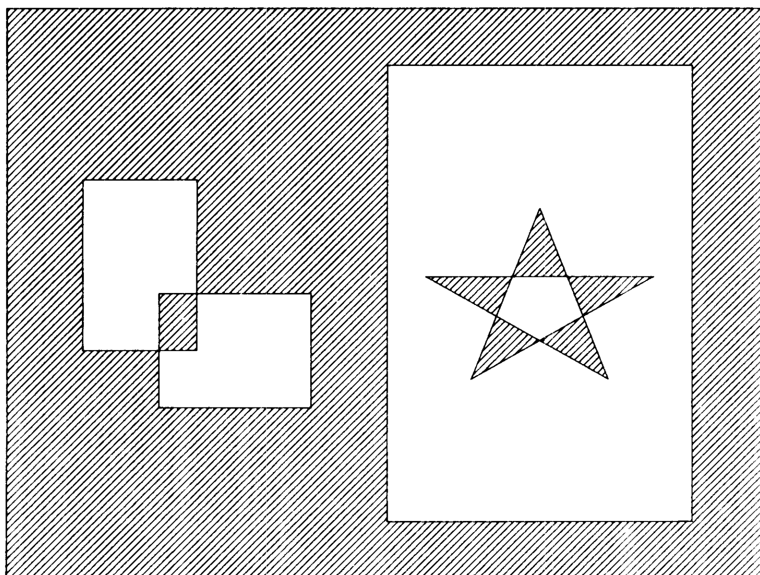
The first entry in the operation selector array **must** be 2, since it is the first vertex of a sub-polygon.

INT_POLYGON_DD is used to output a polygon-set, specified in world coordinates, adhering within the capabilities of the device to the polygon style attributes that are currently specified. A polygon-set is a set of polygons (called "sub-polygons") that are treated graphically as one polygon. The subpolygons in a polygon-set may intersect or overlap each other.

The edge of a sub-polygon is defined as the line sequence that connects its vertices in the order specified. If the last vertex specified for a sub-polygon is not the same as the first, they are automatically connected.

When a polygon-set is displayed, the primitive attributes for polygons and lines define its appearance. In particular, the interior of the polygon-set will be filled according to the attributes of polygon style, polygon interior color and polygon interior line-style. If the edges are to be displayed as specified in the polygon style, the edges will adhere to the current line attributes of color, line-style and line-width.

The filling of polygons also depends on how the sub-polygons "nest" within each other. An "even-odd" rule is used for determining which areas will be filled. Moving across the screen, count the edges of the polygon. Odd-numbered edges will turn the fill on and even-numbered edges will turn the fill off. The picture below will help clear up how the fills work.



Polygon Filling

Refer to SET_PGN_TABLE, SET_PGN_STYLE, SET_PGN_COLOR, SET_PGN_LS for a more detailed description of how attributes affect polygons.

As stated above, the values in the operation selector array define how the edges of the sub-polygons are displayed. The edge from the (I-1)th vertex to the Ith vertex will only be displayed if the Ith entry in the operation selector array equals 1. To display the edge from the last vertex to the first vertex of a sub-polygon, the first vertex must be explicitly respecified after all the other vertices of the sub-polygon, with an operation selector equal to 1. Otherwise the edge from the last vertex to the first will not be drawn. It will, however, automatically be connected for polygon filling.

If it is within the capabilities of the device, filling of the sub-polygon will be done to the sub-polygon edges regardless of whether the edges are displayed. If an entry in the operation selector array does not equal 0, 1, or 2, it will be treated as if it were equal to 0, i.e., the edge will not be drawn.

When INT_POLYGON_DD is used, the current position is updated to the end of the last sub-polygon specified in the polygon-set. The end of the last sub-polygon is defined to be the first (implicit last) vertex of the subpolygon. So, if there is only one vertex in a polygon-set this call degenerates to an update of the current position to the first coordinate set in the x and y point arrays (x coordinate array[1], y coordinate array[1]).

It is the application program's responsibility to ensure that the arrays are all dimensioned to at least the number of elements specified by points and that at least that many values are contained in each array.

Device capabilities vary widely. Not all devices are able to draw polygon edges as requested. If a device is not able to draw polygon edges as requested, they will be simulated in software. The simulation will always adhere to the edge value in SET_PGN_STYLE and the operation selector in INT_POLYGON_DD, but the line-style and color of the edge will depend on the capability of the device to produce lines with those attributes.

Polygon fill capabilities can vary widely between devices. A device may have no filling capabilities at all, may be able to perform only solid fill, or may be able to fill polygons with different fill densities and at different fill line orientations. INT_POLYGON_DD tries to match the device capabilities to the request. If the device cannot fill the request at all, then no simulation is done and the polygon will not be filled. For HPGL plotters, the fill is simulated. For raster devices, if the density is greater than 0.5, a solid fill is used, otherwise, the fill is simulated.

In the case where the polygon style specifies non-display of edged, this would result in no visible output although visible output had been specified. To provide some visible output in this case, INT_POLYGON_DD will outline the polygon using the color and line-style specified for the fill lines. However, only those edge segments specified as displayable by the operation selector array will be drawn. Therefore, if all edge segments are specified as non-displayed, there will still be no visible output.

Regardless of the capabilities of the device, INT_POLYGON_DD sets the starting position to the first vertex of the last member polygon specified in the call. If there is only one polygon specified, the starting position will therefore be set to the first vertex specified.

Polygons are defined to be closed surfaces. When a sub-polygon extends beyond a clipping edge the closed nature of the sub-polygon is destroyed. As with other primitives, unpredictable results may occur if the sub-polygon extends beyond the clipping window.

This procedure is the same as the procedure POLYGON_DEV_DEP, with the exception that the parameters are of type *Gshortint* (– 32 768..32 767). When used with some displays this procedure may perform about 3 times faster than the POLYGON_DEV_DEP procedure. For all other displays this procedure has about the same performance as the POLYGON_DEV_DEP procedure.

The INT_POLYGON_DD procedure only has increased performance when the following conditions exist:

- The display is a raster device.
- The window bounds are within the range – 32 768 through 32 767.
- The window is less then 32 767 units wide and high.

INT_POLYGON_DD is provided for efficient vector generation. Although its use can be mixed with MOVE, LINE, POLYLINE, and POLYGON_DEV_DEP, one dot roundoff errors may result with mixed use since different algorithms are used to implement each.

Error Conditions

The graphics system must be initialized, a graphics display must be enabled, all parameters must be within specified limits and the number of points (Points) must be greater than 0 or the call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR will return a non-zero value.

INT_POLYLINE

IMPORT: dgl_types
dgl_lib

This **procedure** draws a connected line sequence starting at the specified point.

Syntax

Item	Description/Default	Range Restrictions
points	Expression of TYPE INTEGER	MININT thru MAXINT
x array name	Array of TYPE *Gshortint*. *Gshortint* is a subrange of INTEGER.	−32 768 to 32 767
y array name	Array of TYPE *Gshortint*. *Gshortint* is a subrange of INTEGER.	−32 768 to 32 767

Procedure Heading

```
PROCEDURE INT_POLYLINE (        Npts         : INTEGER;
                         ANYVAR Xvec, Yvec   : Gshortint_list )
```

Semantics

Points is the number of vertices in the polygon set.

The **x** and **y coordinate arrays** contain the world coordinate values for each vertex of the polyline-set. The vertices must be in order. The vertices for the first sub-polyline must be at the beginning of these arrays, followed by the vertices for the second sub-polyline, etc. So, the coordinate arrays must contain a total number of vertices that equals points.

The procedure INT_POLYLINE provides the capability to draw a series of connected lines starting at the specified point. A complete object can be drawn by making one call to this procedure. This call first sets the starting position to be the first elements in the x and y coordinate arrays. The line sequence begins at this point and is drawn to the second element in each array, then to the third and continues until points-1 lines are drawn.

This procedure is equivalent to the following sequence of calls:

```
INT_MOVE (X_coordinate_array[1],Y_coordinate_array[1]);
INT_LINE (X_coordinate_array[2],Y_coordinate_array[2]);
INT_LINE (X_coordinate_array[3],Y_coordinate_array[3]);
          :
          :
INT_LINE (X_coordinate_array[Points],Y_coordinate_array[Points]);
```

The starting position is set to (X_coordinate_array[Points], Y_coordinate_array[Points]) at the completion of this call.

Specifying only one element, or Points equal to 1, causes a move to be made to the world coordinate point specified by the first entries in the two coordinate arrays.

It is the application program's responsibility to ensure that the arrays are all dimensioned to at least the number of elements specified by points and that at least that many values are contained in each array.

Depending on the nature of the current line-style nothing may appear on the graphics display. See SET_LINE_STYLE for a complete description of how line-style affects a particular point or vector.

The primitive attributes of color, line-style, and line-width apply to polylines.

This procedure is the same as the POLYLINE procedure, with the exception that the parameters are of type *Gshortint* (− 32 768..32 767). When used with some displays this procedure may perform about 3 times faster than the POLYLINE procedure. For all other displays this procedure has about the same performance as the POLYLINE procedure.

The INT_POLYLINE procedure only has increased performance when the following conditions exist:

- The display must be a raster device.
- The window bounds within the range − 32 768 to 32 767.
- The window must be less then 32 767 units wide and high.

INT_POLYLINE is provided for efficient vector generation. Although its use can be mixed with MOVE, LINE, and POLYLINE, one dot roundoff errors may result with mixed use since different algorithms are used to implement each.

Error Conditions

The graphics system must be initialized, a graphics display must be enabled, all parameters must be within specified limits and the number of points (points) must be greater than 0 or the call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

LOCATOR_INIT

IMPORT: dgl_lib

This **procedure** enables the locator device for input.

Syntax

Item	Description/Default	Range Restrictions
device selector	Expression of TYPE INTEGER	MININT TO MAXINT
error variable name	Variable of TYPE INTEGER	–

Procedure Heading

```
PROCEDURE LOCATOR_INIT (     Dev_Adr  : INTEGER;
                         VAR Ierr     : INTEGER );
```

Semantics

The **device selector** specifies the physical addresses of the graphics locator device.

Device Selector	Locator Device Selected
2	Relative locator, such as knob or mouse
100..3199	HP-IB device at specified select code and address

The **error variable** will contain a value indicating whether the locator device was successfully enabled.

Value	Meaning
0	The locator device was successfully initialized.
2	Unrecognized device specified. Unable to communicate with a device at the specified address, non-existent interface card or non-graphics system supported interface card.

If the error variable contains a non-zero value, the call has been ignored.

LOCATOR_INIT enables the logical locator device for input. Enabling the locator includes associating the logical locator device with a physical device and initializing the device. The device name is set to the name of the physical device, the device status is set to 1 (enabled) and the internal device selector used by the graphics library is set equal to the device selector provided by the user. This information is available by calling INQ_WS with operation selectors 11052 and 13052.

LOCATOR_INIT implicitly makes the picture current before attempting to initialize the device.

LOCATOR_INIT enables the logical locator device for input. Enabling the locator includes associating the logical locator device with a physical device and initializing the device.

The graphics library attempts to directly identify the type of device by using its device address in some way. The meanings of the device address are defined above.

At the time that the graphics library is initialized, all devices which are to be used must be connected, powered on, ready, and accessible via the specified physical address. Invalid addressed or unresponsive devices result in that device not being initialized and an error being returned.

The locator device must be enabled before it is used for input. The locator device is disabled by calling LOCATOR_TERM.

If the graphics display and the locator are not the same physical device (e.g. HP 9826 display and HP 9111 locator), then the logical locator limits will be set to the default values for the particular locator used. If the graphics display and locator are the same physical device (e.g., HP 9826 display and HP 9826 knob locator), then the logical locator limits are set to the current view surface limits.

The locator echo position is set to the default value (see SET_ECHO_POS).

Only one locator device may be enabled at a time. If a locator is currently enabled, then the enabled device will be terminated (via LOCATOR_TERM) and the call will continue. The locator device should be disabled before the termination of the application program. LOCATOR_INIT is the complementary routine to LOCATOR_TERM.

Absolute Locator Limits (HPGL Plotter or Graphics Tablet)
When the locator device is initialized on an HPGL plotter or graphics tablet, the graphics display is left unaltered. HPGL devices are initialized to the following defaults when LOCATOR_INIT is executed:

Plotter/ Tablet	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
9872	400	285	16000	11400	.7125	40.0
7580	809.5	524.25	32380	20970	.6476	40.0
7585	1100	891.75	44000	35670	.8107	40.0
7586	1182.8	898.1	47312	35924	.7593	40.0
7470	257.5	191.25	10300	7650	.7427	40.0
7550	411.25	254.25	16450	10170	.6182	40.0
7475	416	259.125	16640	10365	.6229	40.0
9111	300.8	217.6	12032	8704	.7234	40.0

The maximum physical limits of the locator for a HPGL device not listed above are determined by the default settings of P1 and P2. The default settings of P1 and P2 are the values they have after an HPGL 'IN' command. Refer to the specific device manual for additional details.

The default logical display surface is set equal to the area defined by P1 and P2 at the time LOCATOR_INIT is invoked.

Note

If the paper is changed in an HP 7580 or HP 7585 plotter while the graphics locator is initialized, it should be the same size of paper that was in the plotter when LOCATOR_INIT was called. If a different size of paper is required, the device should be terminated (LOCATOR_TERM) and re-initialized after the new paper has been placed in the plotter.

No locator points are returned while the pen control buttons are depressed on HPGL plotters.

Relative Locators (Knob or Mouse)

When the locator device is initialized, the graphics display is left unaltered. The default initialization characteristics for the knob on various Series 200 computers is listed below:

Computer	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
Model 216	160	120	400	300	.75	2.5
Model 217	230	175	512	390	.7617	2.226
Model 220 (HP82913A)	210	158	400	300	.75	1.905
Model 220 (HP82912A)	152	114	400	300	.75	2.632
Model 226	120	88	400	300	.75	3.333
Model 236	210	160	512	390	.7617	2.438
Model 236 Color	217	163	512	390	.7617	2.39
Model 237	312	234	1024	768	.75	3.282

The knob uses the current display limits as its locator limits for locator echoes 2 though 8. For all other echoes the above limits are used. An example of when the two limits may differ follows:

The knob locator is initialized on a Model 226. The graphics display is an HP 98627A color output card. The resolution of the locator is 0 through 399 in the X dimension, and 0 through 299 in the Y dimension. The resolution of the display is 0 through 511 in the X dimension, and 0 through 389 in the Y dimension. When AWAIT_LOCATOR is used with echo 4, the locator will effectively have the HP 98627A resolution for the duration of the AWAIT_LOCATOR call. However, if echo 1 is used with AWAIT_LOCATOR, the cursor will appear on the Model 226 and the locator has a resolution of 0 through 399 and 0 through 299. Note that all conversion routines and inquiries will use the Model 226 limits.

The physical origin of the locator device is the lower left corner of the display.

Error Conditions

The graphics system must be initialized or this call will be ignored, an ESCAPE (-27) will be generated, and GRAPHICSERROR will return a non-zero value.

LOCATOR_TERM

IMPORT: dgl_lib

This **procedure** disables the enabled locator device.

Syntax

→(LOCATOR_TERM)→

Procedure Heading

```
PROCEDURE LOCATOR_TERM;
```

Semantics

LOCATOR_TERM terminates and disables the enabled locator device. It transmits any termination sequence required by the device and releases all resources being used by the device. The device name is set to the default device name (' '), the device status is set to 0 (not enabled) and the device address is set to 0.

LOCATOR_TERM is the complementary routine to LOCATOR_INIT.

If a locator device is used, LOCATOR_TERM should be called before the application program is terminated.

Error Conditions

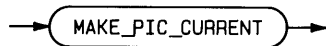
The graphics system must be initialized and a locator device enabled or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

MAKE_PIC_CURRENT

IMPORT: dgl_lib

This **procedure** makes the picture current.

Syntax



Procedure Heading

```
PROCEDURE MAKE_PIC_CURRENT;
```

Semantics

The graphics display surface can be made current at any time with a call to MAKE_PIC_CURRENT. This insures that all previously generated primitives have been sent to the graphics display device. Due to operating system delays, all picture changes may not have been displayed on the graphics display upon return to the calling program. MAKE_PIC_CURRENT is most often used in system buffering mode (see SET_TIMING) to make sure that all output has been sent to the graphics display device when required.

Before performing any non-graphics library input or output to an active graphics device, (e.g., a Pascal read or write), it is essential that all of the previously generated output primitives be sent to the device. If immediate visibility is the current timing mode, all primitives will be sent to the device before completion of the call to generate them, but if system buffering is used, MAKE_PIC_CURRENT should be called before performing any non-graphics system I/O.

The following routines implicitly make the picture current:

AWAIT_LOCATOR	DISPLAY_TERM	INPUT_ESC
LOCATOR_INIT	SAMPLE_LOCATOR	

A call to MAKE_PIC_CURRENT can be made at any time within an application program to insure that the image is fully displayed. MAKE_PIC_CURRENT does not modify the current timing mode.

Error Conditions

The graphics system must be initialized and a display must be enabled or this call will be ignored, an ESCAPE (-27) will be generated, and GRAPHICSERROR will return a non-zero value.

MARKER

IMPORT: dgl_lib

This **procedure** outputs a marker symbol at the starting position.

Syntax

Item	Description/Default	Range Restrictions	Recommended Range
marker selector	Expression of TYPE INTEGER	MININT TO MAXINT	1 thru 19

Procedure Heading

```
PROCEDURE MARKER ( Marker_num : INTEGER );
```

Semantics

The **marker selector** determines which marker will be output. There are 19 defined invariant marker symbols (1-19). They are defined as follows:

1 - '.'	7 - rectangle	13 - '3'
2 - '+'	8 - diamond	14 - '4'
3 - '*'	9 - rectangle with cross	15 - '5'
4 - 'O'	10 - '0'	16 - '6'
5 - 'X'	11 - '1'	17 - '7'
6 - triangle	12 - '2'	18 - '8'
		19 - '9'

Marker numbers 20 and larger are device dependent.

MARKER outputs the marker designated by the marker selector, centered about the starting position. The starting position is left unchanged at the completion of this call.

If the marker selector specified is greater than the number of distinct marker symbols that are supported by a device, then marker number 1 ('.') will be used. INQ_WS can be used to inquire the number of distinct marker symbols that are available on a particular graphics display device. Depending on a particular display device's capabilities, the graphics library uses either hardware or software to generate the marker symbols.

The size and orientation of markers is fixed and not affected by the viewing transformation. The size of markers is device dependent and cannot be changed.

Only the primitive attributes of color and highlighting apply to markers. However, the marker will appear with these attributes only if the device is capable of applying them to markers.

Error Conditions

The graphics system must be initialized and a display device enabled or the call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

MOVE

IMPORT: dgl_lib

This **procedure** sets the starting position to the world coordinate specified.

Syntax

Item	Description/Default	Range Restrictions
x coordinate	Expression of TYPE REAL	–
y coordinate	Expression of TYPE REAL	–

Procedure Heading

```
PROCEDURE MOVE ( Wx, Wy : REAL );
```

Semantics

MOVE specifies where the next graphical primitive will be output. It does this by setting the value of the starting position to the world coordinate system point specified by the X,Y coordinate values and then moving the physical beam or pen to that point.

The **x** and **y coordinate** pair is the new starting position in world coordinates.

The starting position corresponds to the location of the physical pen or beam in all but four instances: after a change in the viewing transformation, after initialization of a graphical display device, after the output of a text string, or after the output of a graphical escape function. A call to MOVE or INT_MOVE should therefore be made after any one of the following calls to update the value of the starting position and in so doing, place the physical pen or beam at a known location: SET_ASPECT, DISPLAY_INIT, SET_DISPLAY_LIM, OUTPUT_ESC, TEXT, SET_VIEWPORT, and SET_WINDOW.

Error Conditions

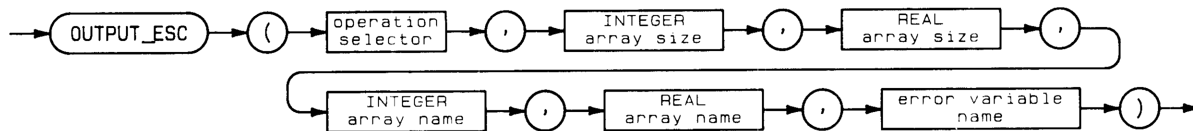
The graphics system must be enabled and a display device enabled or this call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR will return a non-zero value.

OUTPUT_ESC

IMPORT: dgl_lib

This **procedure** performs a device dependent escape function to inquire from the graphics display device.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
operation selector	Expression of TYPE INTEGER	MININT to MAXINT	–
INTEGER array size	Expression of TYPE INTEGER	MININT to MAXINT	>0
REAL array size	Expression of TYPE INTEGER	MININT to MAXINT	>0
INTEGER array name	Any valid variable. Should be INTEGER array	–	–
REAL array name	Any valid variable. Should be REAL array	–	–
error variable name	Variable of TYPE INTEGER	–	–

Procedure Heading

```
PROCEDURE OUTPUT_ESC (          Opcode  : INTEGER;
                                Isize   : INTEGER;
                                Rsize   : INTEGER;
                       ANYVAR   Ilist   : Gint_list;
                       ANYVAR   Rlist   : Greal_list;
                          VAR   Ierr    : INTEGER     );
```

Semantics

The **operation selector** determines the device dependent output escape function to be performed. The codes supported for a given device are described in the device handlers section of this document.

The **INTEGER array size** is the number of INTEGER parameters contained in the INTEGER array. The thousand's digit of the operation selector is the number of INTEGER parameters that the graphics system expects.

The **REAL array size** is the number of REAL parameters contained in the REAL array by the escape function. The ten-thousand's digit of the operation selector is the number of REAL parameters that the graphics system expects.

The **INTEGER array** is the array in which zero or more INTEGER parameters are contained.

The **REAL array** is the array in which zero or more REAL parameters are contained.

The **error variable** will contain a value indicating whether the escape function was performed.

Value	Meaning
0	Output escape function successfully sent to the device.
1	Operation not supported by the graphics display device.
2	The INTEGER array size is not equal to the number of required INTEGER parameters.
3	The REAL array size is not equal to the number of required REAL parameters.
4	Illegal parameters specified.

If the error variable contains a non-zero value, the call has been ignored.

OUTPUT_ESC allows application programs to access special device features on a graphics display device. The desired escape function is specified by a unique value for opcode.

The type of information passed to the graphics display device is determined by the value of opcode. The graphics library does not check OUTPUT_ESC parameters which will be sent directly to the display device. This can lead to device dependent results if out of range values are sent.

Output escape functions only apply to the graphics display device.

The starting position may be altered by a call to OUTPUT_ESC.

Error Conditions

The graphics system must be initialized and a display device must be enabled or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

Raster Device Escape Operations

Operation Selector	Function
52	Dump graphics of the currently active display device if it is the console or a bit-mapped display. Graphics will be dumped to the graphics printer (PRINTER:); if color, all planes are ORed.

53	Await vertical blanking. This escape function will not exit until the CRT is performing vertical blanking.
	The following example shows how to use this function when changing the color table to reduce flicker.
	`OUTPUT_ESC ( 53, 0, 0, dummy, dummy, error );`
	`SET_COLOR_TABLE ( 0, r, g, b );`
	The color table is not changed until the crt is blank (during a refresh cycle). Otherwise changing the color map in the middle of a scan would create a screen that was half the old color, and half the new color for one frame (1/60 sec). To the eye this would look like a flicker.

250	Specify device limits. REAL Array [1] = Points (dots) per mm in X direction REAL Array [2] = Points (dots) per mm in Y direction
1050[1]	Turn on or off the graphics display. INTEGER array [1] = 0 → turn display off. INTEGER array [1] <> 0 → turn display on.
1051[1]	Turn on or off the alpha display. INTEGER array [1] = 0 → turn display off. INTEGER array [1] <> 0 → turn display on.
1052	Set special drawing modes. Using this escape function will redefine the meaning of the set color attribute. For details on how a given drawing mode affects a color see "Drawing Modes" in SET_COLOR. This drawing mode does not apply to device dependent polygons. Out of range values default to dominate drawing mode. INTEGER array[1] = 0 → Dominate drawing mode. = 1 → Non-dominate drawing mode. = 2 → Erase drawing mode. = 3 → Complement drawing mode.
1053	Dump graphics (from the specified color planes) to the graphics printer (PRINTER:). Also dumps graphics on a Model 237 if it is the currently active display. INTEGER array [1] = Color plane selection code. BIT 1 = 1 → Select plane 1. (Blue on HP 98627A) BIT 2 = 1 → Select plane 2. (Green on HP 98627A) BIT 3 = 1 → Select plane 3. (Red on HP 98627A) BIT 4 = 1 → Select plane 4.
1054	Clear selected graphics planes. INTEGER Array [1] = 0 - Clear all planes INTEGER Array [1] <> 0 - Color plane selection code. BIT 1 = 1 Clear plane 1 (Blue on HP 98627A) BIT 2 = 1 Clear plane 2 (Green on HP 98627A) BIT 3 = 1 Clear plane 3 (Red on HP 98627A) BIT 4 = 1 Clear plane 4

1 This operation is not available for the Model 237 computer.

Operation Selector	Function
10050	Set all HP 9836C color table locations. This escape function allows the user to change all locations in the hardware color map with one procedure. The software maintained color table will be updated by this call. This escape function is the same as calling SET_COLOR_TABLE with indexes 0 - 15. REAL Array [1] = Parm1 REAL Array [2] = Parm2 Index 0 REAL Array [3] = Parm3 REAL Array [4] = Parm1 REAL Array [5] = Parm2 Index 1 REAL Array [6] = Parm3 : : REAL Array [46] = Parm1 REAL Array [47] = Parm2 Index 15 REAL Array [48] = Parm3 Parm1, Parm2, and Parm3 are defined to be the same as used with SET_COLOR_TABLE. The size of the INTEGER array must equal 0 and the size of the REAL array 48.

The following table shows which escape codes are supported on which Series 200 raster displays:

Operation Selector	216	217	220	226	236	236 Color	237	98627A
52	yes	yes	yes	yes	yes	yes	yes	yes
53	no	no	no	no	no	yes	no	no
250	no	no	no	no	no	no	no	yes
1050	yes	yes	yes	yes	yes	yes	no	yes
1051	yes	yes	yes	yes	yes	yes	no	no
1052	yes	yes	yes	yes	yes	yes	yes	yes
1053	no	no	no	no	no	yes	yes	yes
1054	yes	no	no	yes	yes	yes	no	yes
10050	no	no	no	no	no	yes	no	no

HPGL Plotter Escape Operations

Operation Selector	Function
1052*	Enable cutter. Provides means to control the Plotter paper cutters. Paper is cut after it is advanced. INTEGER array [1] = 0 Cutter is disabled. INTEGER array [1] <> 0 Cutter is enabled.
1052	Set automatic pen. This instruction provides a means for utilizing the smart pen options of the plotter. Initially, all automatic pen options are enabled. INTEGER array [1]: BIT 1 = 1 Lift pen if it has been down for 60 seconds. BIT 2 = 1 Put pen away if it has been motionless for 20 seconds. BIT 3 = 1 Do not select a pen until a command which makes a mark. This causes the pen to remain in the turret for the longest possible time.
1053	Advance the paper either one half or a full page. INTEGER array [1] = 0 >> Advance page half INTEGER array [1] <> 0 >> Advance page full
2050	Select pen velocity. This instruction allows the user to modify the plotter's pen speed. Pen speed may be set from 1 to the maximum for the given device. INTEGER array [1] = Pen speed (INTEGER from 1 to device max). INTEGER array [2] = Pen number (INTEGER from 1 to 8; other integers select all pens)
2051	Select pen force. The force may be set from 10 to 66 gram-weights. INTEGER array [1] = Pen force (INTEGER from 1 to 8). 1: 10 gram-weights 2: 18 gram-weights 3: 26 gram-weights 4: 34 gram-weights 5: 42 gram-weights 6: 50 gram-weights 7: 58 gram-weights 8: 66 gram-weights INTEGER array [2] = Pen number (INTEGER 1 to 8; other integers select all pens)
2052	Select pen acceleration. The acceleration may be set from 1 to 4 G's. INTEGER array [1] = Pen acceleration (INTEGER from 1 to 4). INTEGER array [2] = Pen number (INTEGER 1 to 8; other integers select all pens)

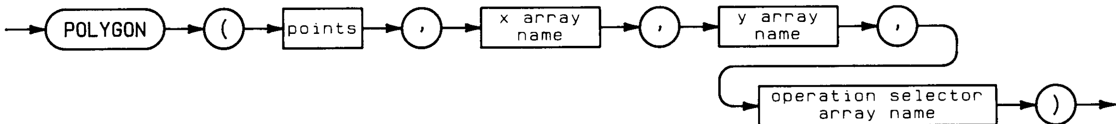
Operation Selector	9872	7470	7475	7550	7580	7585	7586
1052*	S/T	no	no	no	no	no	no
1052	no	no	yes	yes	yes	yes	yes
1053	S/T	no	no	yes	no	no	yes
2050	yes	yes	yes	yes	yes	yes	yes
2051	no	no	yes	yes	yes	yes	yes
2052	no	no	yes	yes	yes	yes	yes

POLYGON

IMPORT: dgl_types
dgl_lib
dgl_poly

This **procedure** displays a polygon-set starting and ending at the specified point adhering to the specified polygon style exactly as specified (i.e., device-independent results).

Syntax



Item	Description/Default	Range Restrictions
points	Expression of TYPE INTEGER	MININT thru MAXINT
x array name	Array of TYPE REAL.	–
y array name	Array of TYPE REAL.	–
operation selector array name	Array of TYPE *Gshortint*. *Gshortint* is a subrange of INTEGER.	–32 768 to 32 767

Procedure Heading

```
PROCEDURE POLYGON (        Npoint  : INTEGER;
                   ANYVAR  Xvec    : Greal_list;
                   ANYVAR  Yvec    : Greal_list;
                   ANYVAR  Opcodes : Gshortint_list);
```

Semantics

Points is the number of vertices in the polygon set.

The **x** and **y coordinate arrays** contain the world coordinate values for each vertex of the polygon-set. The vertices must be in order. The vertices for the first sub-polygon must be at the beginning of these arrays, followed by the vertices for the second sub-polygon, etc. So, the coordinate arrays must contain a total number of vertices that equals points.

The **operation selector array** contains a series of integer operation selectors defining which vertices start new polygons, and defining which edges should be displayed.

Value	Meaning
0	Don't display the line for the edge extending to this vertex from the previous vertex.
1	Display the line for the edge extending to this vertex from the previous vertex.
2	This vertex is the first vertex of a sub-polygon. Succeeding vertices are part of a sub-polygon until a new start-of-polygon operation selector (2) is encountered. (Or the end of the arrays is encountered.)

Note

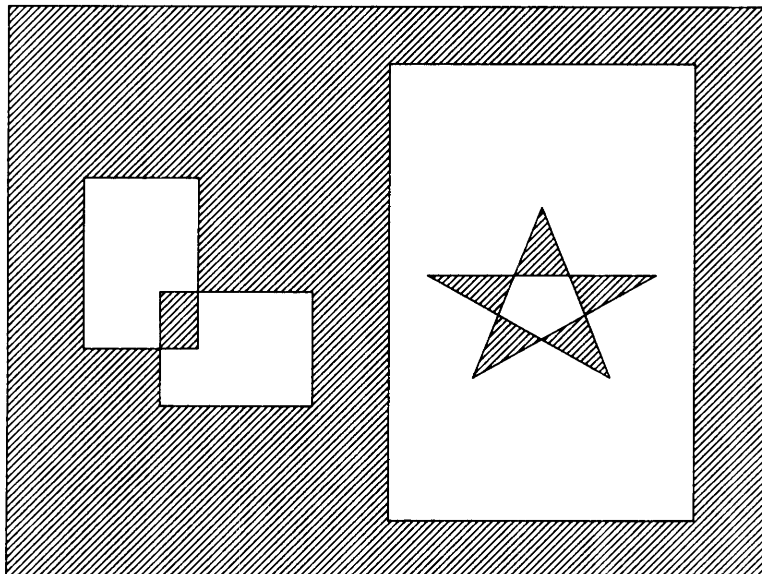
The first entry in the operation selector array **must** be 2, since it is the first vertex of a sub-polygon.

POLYGON is used to output a polygon-set, specified in world coordinates, adhering exactly to the polygon style attributes that are currently specified. A polygon-set is a set of polygons (called "sub-polygons") that are treated graphically as one polygon. This is accomplished by "stacking" the sub-polygons. The subpolygons in a polygon-set may intersect or overlap each other.

The edge of a sub-polygon is defined as the line sequence that connects its vertices in the order specified. If the last vertex specified for a sub- polygon is not the same as the first, they are automatically connected.

When a polygon-set is displayed, the primitive attributes for polygons and lines define its appearance. In particular, the interior of the polygon-set will be filled according to the attributes of polygon style, polygon interior color and polygon interior line-style. If the edges are to be displayed as specified in the polygon style, the edges will adhere to the current line attributes of color, line-style and line-width. A dot will disappear on an edged polygon if the edge is done with a complementing line.

The filling of polygons also depends on how the sub-polygons "nest" within each other. An "even-odd" rule is used for determining which areas will be filled. Moving across the screen, count the edges of the polygon. Odd-numbered edges will turn the fill on and even-numbered edges will turn the fill off. The picture below will help clear up how the fills work.



Polygon Filling

Refer to SET_PGN_TABLE, SET_PGN_STYLE, SET_PGN_COLOR, SET_PGN_LS for a more detailed description of how attributes affect polygons.

As stated above, the values in the operation selector array define how the edges of the sub-polygons are displayed. The edge from the (I-1)th vertex to the Ith vertex will only be displayed if the Ith entry in the operation selector array equals 1. To display the edge from the last vertex to the first vertex of a sub-polygon, the first vertex must be explicitly respecified after all the other vertices of the sub-polygon, with an operation selector equal to 1. Otherwise the edge from the last vertex to the first will not be drawn. It will, however, automatically be connected for polygon filling.

If it is within the capabilities of the device, filling of the sub-polygon will be done to the sub-polygon edges regardless of whether the edges are displayed. If an entry in the operation selector array does not equal 0, 1, or 2, it will be treated as if it were equal to 0 and the edge will not be drawn.

When POLYGON is used, the current position is updated to the end of the last sub-polygon specified in the polygon-set. The end of the last sub-polygon is defined to be the first (implicit last) vertex of the subpolygon. So, if there is only one vertex in a polygon-set this call degenerates to an update of the current position to the first coordinate set in the x and y point arrays (x coordinate array[1], y coordinate array[1]).

It is the application program's responsibility to ensure that the arrays are all dimensioned to at least the number of elements specified by points and that at least that many values are contained in each array.

Polygons are defined to be closed surfaces. When a sub-polygon extends beyond a clipping edge the closed nature of the sub-polygon is destroyed. As with other primitives, unpredictable results may occur if the sub-polygon extends beyond the clipping window.

Error Conditions

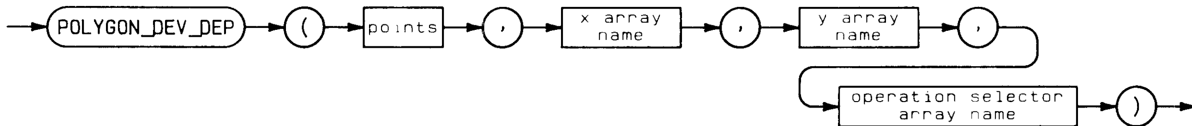
The graphics system must be initialized, a graphics display must be enabled, all parameters must be within specified limits and the number of points specified must be greater than 0 or the call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR will return a non-zero value.

POLYGON_DEV_DEP

IMPORT: dgl_types
dgl_lib
dgl_poly

This **procedure** displays a polygon-set starting and ending at the specified point adhering to the specified polygon style in a device- dependent fashion.

Syntax



Item	Description/Default	Range Restrictions
points	Expression of TYPE INTEGER	MININT thru MAXINT
x array name	Array of TYPE REAL.	–
y array name	Array of TYPE REAL.	–
operation selector array name	Array of TYPE *Gshortint*. *Gshortint* is a sub-range of INTEGER.	– 32 768 to 32 767

Procedure Heading

```
PROCEDURE POLYGON_DEV_DEP (        Npoint      : INTEGER;
                           ANYVAR  Xvec      : Greal_list;
                           ANYVAR  Yvec      : Greal_list;
                           ANYVAR  Opcodes   : Gshortint_list);
```

Semantics

Points is the number of vertices in the polygon set.

The **x** and **y coordinate arrays** contain the world coordinate values for each vertex of the polygon-set. The vertices must be in order. The vertices for the first sub-polygon must be at the beginning of these arrays, followed by the vertices for the second sub-polygon, etc. So, the coordinate arrays must contain a total number of vertices that equals points.

The **operation selector array** contains a series of integer operation selectors defining which vertices start new polygons, and defining which edges should be displayed.

Value	Meaning
0	Don't display the line for the edge extending to this vertex from the previous vertex.
1	Display the line for the edge extending to this vertex from the previous vertex.
2	This vertex is the first vertex of a sub-polygon. Succeeding vertices are part of a sub-polygon until a new start-of-polygon operation selector (2) is encountered. (Or the end of the arrays is encountered.)

Note

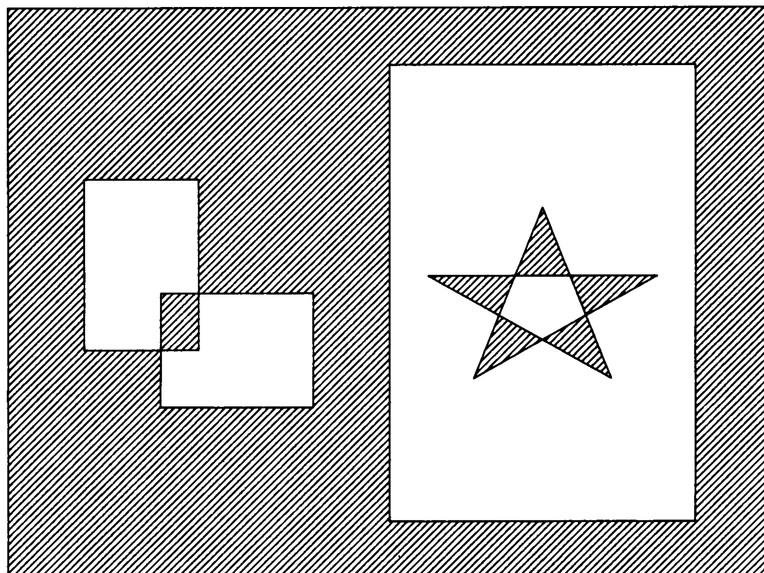
The first entry in the operation selector array **must** be 2, since it is the first vertex of a sub-polygon.

POLYGON_DEV_DEP is used to output a polygon-set, specified in world coordinates, adhering within the capabilities of the device to the polygon style attributes that are currently specified. A polygon-set is a set of polygons (called "sub-polygons") that are treated graphically as one polygon. The subpolygons in a polygon-set may intersect or overlap each other.

The edge of a sub-polygon is defined as the line sequence that connects its vertices in the order specified. If the last vertex specified for a sub- polygon is not the same as the first, they are automatically connected.

When a polygon-set is displayed, the primitive attributes for polygons and lines define its appearance. In particular, the interior of the polygon-set will be filled according to the attributes of polygon style, polygon interior color and polygon interior line-style. If the edges are to be displayed as specified in the polygon style, the edges will adhere to the current line attributes of color, line-style and line-width.

The filling of polygons also depends on how the sub-polygons "nest" within each other. An "even-odd" rule is used for determining which areas will be filled. Moving across the screen, count the edges of the polygon. Odd-numbered edges will turn the fill on and even-numbered edges will turn the fill off. The picture below will help clear up how the fills work.



Polygon Filling

Refer to SET_PGN_TABLE, SET_PGN_STYLE, SET_PGN_COLOR, SET_PGN_LS for a more detailed description of how attributes affect polygons.

As stated above, the values in the operation selector array define how the edges of the sub-polygons are displayed. The edge from the (I-1)th vertex to the Ith vertex will only be displayed if the Ith entry in the operation selector array equals 1. To display the edge from the last vertex to the first vertex of a sub-polygon, the first vertex must be explicitly respecified after all the other vertices of the sub-polygon, with an operation selector equal to 1. Otherwise the edge from the last vertex to the first will not be drawn. It will, however, automatically be connected for polygon filling.

If it is within the capabilities of the device, filling of the sub-polygon will be done to the sub-polygon edges regardless of whether the edges are displayed. If an entry in the operation selector array does not equal 0, 1, or 2, it will be treated as if it were equal to 0, i.e., the edge will not be drawn.

When POLYGON_DEV_DEP is used, the current position is updated to the end of the last sub-polygon specified in the polygon-set. The end of the last sub-polygon is defined to be the first (implicit last) vertex of the subpolygon. So, if there is only one vertex in a polygon-set this call degenerates to an update of the current position to the first coordinate set in the x and y point arrays (x coordinate array[1], y coordinate array[1]).

It is the application program's responsibility to ensure that the arrays are all dimensioned to at least the number of elements specified by points and that at least that many values are contained in each array.

Device capabilities vary widely. Not all devices are able to draw polygon edges as requested. If a device is not able to draw polygon edges as requested, they will be simulated in software. The simulation will always adhere to the edge value in SET_PGN_STYLE and the operation selector in POLYGON_DEV_DEP, but the line-style and color of the edge will depend on the capability of the device to produce lines with those attributes.

Polygon fill capabilities can vary widely between devices. A device may have no filling capabilities at all, may be able to perform only solid fill, or may be able to fill polygons with different fill densities and at different fill line orientations. POLYGON_DEV_DEP tries to match the device capabilities to the request. If the device cannot fill the request at all, then no simulation is done and the polygon will not be filled. For HPGL plotters, the fill is simulated. For raster devices, if the density is greater than 0.5, a solid fill is used, otherwise, the fill is simulated.

In the case where the polygon style specifies non-display of edged, this would result in no visible output although visible output had been specified. To provide some visible output in this case, POLYGON_DEV_DEP will outline the polygon using the color and line-style specified for the fill lines. However, only those edge segments specified as displayable by the operation selector array will be drawn. Therefore, if all edge segments are specified as non-displayed, there will still be no visible output.

Regardless of the capabilities of the device, POLYGON_DEV_DEP sets the starting position to the first vertex of the last member polygon specified in the call. If there is only one polygon specified, the starting position will therefore be set to the first vertex specified.

Polygons are defined to be closed surfaces. When a sub-polygon extends beyond a clipping edge the closed nature of the sub-polygon is destroyed. As with other primitives, unpredictable results may occur if the sub-polygon extends beyond the clipping window.

Error Conditions

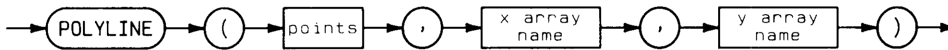
The graphics system must be initialized, a graphics display must be enabled, all parameters must be within specified limits and the number of points (Points) must be greater than 0 or the call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

POLYLINE

IMPORT: dgl_lib

This **procedure** draws a connected line sequence starting at the specified point.

Syntax



Item	Description/Default	Range Restrictions
points	Expression of TYPE INTEGER	MININT thru MAXINT
x array name	Array of TYPE REAL.	–
y array name	Array of TYPE REAL.	–

Procedure Heading

```
PROCEDURE POLYLINE (        Npts          : INTEGER;
                     ANYVAR Xvec, Yvec    : Greal_list )
```

Semantics

Points is the number of vertices in the polygon set.

The **x** and **y coordinate arrays** contain the world coordinate values for each vertex of the polyline-set. The vertices must be in order. The vertices for the first sub-polyline must be at the beginning of these arrays, followed by the vertices for the second sub-polyline, etc. So, the coordinate arrays must contain a total number of vertices that equals points.

The procedure POLYLINE provides the capability to draw a series of connected lines starting at the specified point. A complete object can be drawn by making one call to this procedure. This call first sets the starting position to be the first elements in the x and y coordinate arrays. The line sequence begins at this point and is drawn to the second element in each array, then to the third and continues until points-1 lines are drawn.

This procedure is equivalent to the following sequence of calls:

```
MOVE (X_coordinate_array[1],Y_coordinate_array[1]);
LINE (X_coordinate_array[2],Y_coordinate_array[2]);
LINE (X_coordinate_array[3],Y_coordinate_array[3]);
            :
            :
LINE (X_coordinate_array[Points],Y_coordinate_array[Points]);
```

The starting position is set to (X_coordinate_array[Points], Y_coordinate_array[Points]) at the completion of this call.

Specifying only one element, or Points equal to 1, causes a move to be made to the world coordinate point specified by the first entries in the two coordinate arrays.

It is the application program's responsibility to ensure that the arrays are all dimensioned to at least the number of elements specified by points and that at least that many values are contained in each array.

Depending on the nature of the current line-style nothing may appear on the graphics display. See SET_LINE_STYLE for a complete description of how line-style effects a particular point or vector.

The primitive attributes of color, line-style, and line-width apply to polylines.

Error Conditions

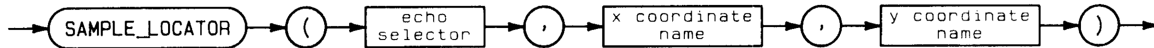
The graphics system must be initialized, a graphics display must be enabled, all parameters must be within specified limits and the number of points (points) must be greater than 0 or the call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR will return a non-zero value.

SAMPLE_LOCATOR

IMPORT: dgl_lib

This **procedure** samples the current locator device

Syntax



Item	Description/Default	Range Restrictions
echo selector	Expression of TYPE INTEGER	MININT to MAXINT
x coordinate name	Variable of TYPE REAL	–
y coordinate name	Variable of TYPE REAL	–

Procedure Heading

```
PROCEDURE SAMPLE_LOCATOR (      Echo    : INTEGER;
                            VAR Wx, Wy  : REAL      );
```

Semantics

The **echo selector** determines the level of input echoing. Possible values are:

0 - No echo.
≥1 - Echo on the locator device.

The **x** and **y coordinates** are the values of the coordinates, expressed in world coordinate units, returned from the enabled locator device.

SAMPLE_LOCATOR returns the current world coordinate value of the locator without waiting for any user intervention. Typically, the locator is sampled in applications involving the continuous input of data points that are very close together.

If the point sampled is outside of the current logical locator limits, the transformed point will still be returned .

The number of echoes supported by a locator device and the correlation between the echo value and the type of echoing performed is device dependent. Most locator devices support at least one form of echoing. Possible echoes are beeping, displaying the point sampled, etc. See the locator descriptions below to find the locators supported by the various devices. If the echo value is larger than the number of echoes supported by the enabled locator device, then echo 1 will be used.

Locator echoing can only be performed on the locator device. The locator echo position is not used in conjunction with any echoes performed while sampling a locator.

SAMPLE_LOCATOR implicitly makes the picture current before sampling the locator.

Relative Locators (Knob or Mouse)
The keyboard beeper is sounded when the locator is sampled if an echo is selected (echo selector$\geq$1). The sample locator function returns the last AWAIT_LOCATOR result or 0.0, 0.0 if AWAIT_LOCATOR has not been invoked since LOCATOR_INIT.

Absolute Locators (HPGL Plotter or Graphics Tablet)
The SAMPLE_LOCATOR function returns the current locator position without waiting for an operator response (pen position on plotters). On a 9111A graphics Tablet, the beeper is sounded when the stylus is depressed. For echo selectors greater than or equal to 9, the same echo as echo selector 1 is used.

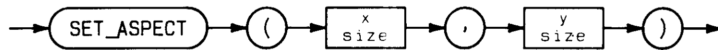
Error Conditions
The graphics system must be initialized and a locator device enabled or this call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_ASPECT

IMPORT: dgl_lib

This **procedure** redefines the aspect ratio of the virtual coordinate system.

Syntax



Item	Description/Default	Range Restrictions
x size	Expression of TYPE REAL	—
y size	Expression of TYPE REAL	—

Procedure Heading

```
PROCEDURE SET_ASPECT ( X_size, Y_size : REAL );
```

Semantics

The **x size** is the width of the virtual coordinate system in dimensionless units. The size must be greater than zero.

The **y size** is the height of the virtual coordinate system in dimensionless units. The size must be greater than zero.

SET_ASPECT sets the aspect ratio of the virtual coordinate system, and hence the view surface, to be y size divided by x size. A ratio of 1 defines a square virtual coordinate system, a ratio greater than 1 specifies it to be higher than it is wide; and a ratio less than 1 specifies it to be wider than it is high. Since x size and y size are used to form a ratio, they may be expressed in any units as long as they are the same units.

The range of coordinates for the virtual coordinate system is calculated based on the value of the aspect ratio. The coordinates of the longer axis are always set to range from 0.0 to 1.0 and those of the shorter axis from 0 to a value that achieves the specified aspect ratio. SET_ASPECT defines the limits of the virtual coordinate system.

ASPECT RATIO (AR)	X LIMITS	Y LIMITS
AR < 1	0.0, 1.0	0.0, 1.0 * AR
AR = 1	0.0, 1.0	0.0, 1.0
AR > 1	0.0, 1.0 / AR	0.0, 1.0

When a call to SET_ASPECT is made, the graphics system sets the viewport equal to the limits of the virtual coordinate system. This routine can therefore be used to access the entire logical display surface. A program could display an image on the entire HP 9826 graphics display, which has an aspect ratio of 399/299, in the following manner:

SET_ASPECT (399, 299);

To set the aspect ratio to the entire display in a device independent manor, INQ_WS may be used as follows:

```
PROCEDURE Set_max_aspect;

    CONST   Get_aspect=254;

VAR     Dummy      : INTEGER;
        Error      : INTEGER;
        Ratio_list: ARRAY[1..2] OF REAL;

    BEGIN {PROCEDURE Set_max_aspect}
       INQ_WS (Get_aspect,0,0,2,Dummy,Dummy, Ratio_list, Error);
       IF Error=0 THEN
          SET_ASPECT(1.0,Ratio_list[2]);
    END; {PROCEDURE Set_max_aspect}
```

The initial value of the aspect ratio is 1, setting the virtual coordinate system to be a square. This square is mapped to the largest inscribed square on any display surface, so that the viewable area is maximized. As a result, the initial virtual coordinate system limits range from 0.0 to 1.0 in both the X and Y directions. A program can access the largest inscribed rectangle on any display surface by modifying the value of the aspect ratio. The exact placement of the rectangle on the display surface is device dependent, but it is centered on CRT's and justified in the lower left hand corner of plotters.

The starting position is not altered by this call. Since this call redefines the viewing transformation, the starting position may no longer represent the last world coordinate position. A call to MOVE or INT_MOVE should therefore be made after this call to update the starting position.

If the logical locator is associated with the same physical device as the graphics display, then a call to SET_ASPECT will set the logical locator limits equal to the new limits of the virtual coordinate system.

Since the window is not affected by the SET_ASPECT procedure, distortion may result in the window to viewport mapping if the window does not have the same aspect ratio as the virtual coordinate system (see SET_WINDOW).

The locator echo position is set to the default value by this procedure.

Error Conditions

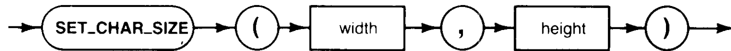
The graphics system must be initialized and both X and Y size must be greater than zero or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_CHAR_SIZE

IMPORT: dgl_lib

This **procedure** sets the character size attribute for graphical text.

Syntax



Item	Description/Default	Range Restrictions
width	Expression of TYPE REAL	–
height	Expression of TYPE REAL	–

Procedure Heading

```
PROCEDURE SET_CHAR_SIZE ( Width, Height : REAL );
```

Semantics

The **width** is the requested graphics character cell width in world coordinate units. (width <> 0.0)

The **height** is the requested graphics character cell height in world coordinate units. (height <> 0.0)

SET_CHAR_SIZE sets the character size for subsequently output graphics text. The absolute value of width and height are used to specify the world coordinate size of a character cell. Therefore, the actual physical size of a character output is determined by applying the current viewing transformations to the world coordinate units specification.

The default character size (set by GRAPHICS_INIT and DISPLAY_INIT) is dependent upon the physical device associated with the graphical display device. The size is determined as follows:

- Height := .05 x (height of the world coordinate system)
- Width := .035 x (width of the world coordinate system)

If a change is made to the viewing transformation (by SET_WINDOW, SET_VIEWPORT, SET_DISPLAY_LIM, or SET_ASPECT), the value of the character size attribute will not be changed, but the actual size of the characters generated may be modified.

Error Conditions

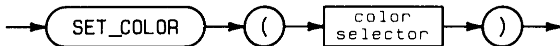
The graphics system must be initialized, a display must be enabled, and width and height must both be non-zero or this call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_COLOR

IMPORT: dgl_lib

This **procedure** sets the color attribute for output primitives except for polygon interior fill.

Syntax



Item	Description/Default	Range Restrictions
color selector	Expression of TYPE INTEGER	–

Procedure Heading

```
PROCEDURE SET_COLOR ( Color : INTEGER );
```

Semantics

SET_COLOR sets the color attribute for the following primitives:

Lines
Markers
Polylines
Polygon Edges
Text

At device initialization a default color table is created by the graphics system. The size and contents of the table are device dependent. At least one entry exists for all devices. A call to INQ_WS with OPCODE equal to 1053 will return the number of colors available on a given graphics device. Some devices allow the color table to be modified with SET_TABLE.

The **color selector** is an index into the color table. The contents of the color table are then used to specify the color when primitives are drawn. On some devices (HPGL plotters), the color selector maps directly to a pen number for the device. On the HP 9836C, the entries in the color table can be modified with SET_COLOR_TABLE.

The default value of the color attribute is 1. If the value of the color selector is not supported on the graphics display, the color attribute will be set to 1.

A color selector of 0 has special effects depending on the graphics display used. For raster devices, a color selector of 0 means to draw in the background color. For most plotters, it puts the pen away.

If the device is not capable of reproducing a color in the color table, the closest color which the device is capable of reproducing is used instead. On some devices, this may depend on the primitive being displayed. For example, the HP98627A color output interface card is capable of a large selection of polygon fill colors, but only 8 line colors. Thus, the fill color could match the selected color much more closely than the line color used to outline the polygon.

Default Raster Color Map

The following table shows the default (initial) color table for the black and white displays (HP 9816 / HP 9920 / HP 9826 / HP 9836):

Index #	Hue	Saturation	Luminosity
0	0	0	0
1	0	0	1.0000
2	0	0	0.9375
3	0	0	0.8750
4	0	0	0.8125
5	0	0	0.7500
6	0	0	0.6875
7	0	0	0.6250
8	0	0	0.5625
9	0	0	0.5000
10	0	0	0.4375
11	0	0	0.3750
12	0	0	0.3125
13	0	0	0.2500
14	0	0	0.1875
15	0	0	0.1250
16	0	0	0.0625

Colors 17 though 31 are set to white.

The following table shows the default (initial) color table for the color displays (HP 9836C and HP 98627A):

Index #	Color name	Red	Green	Blue
0	Black	0.000000	0.000000	0.000000
1	White	1.000000	1.000000	1.000000
2	Red	1.000000	0.000000	0.000000
3	Yellow	1.000000	1.000000	0.000000
4	Green	0.000000	1.000000	0.000000
5	Cyan	0.000000	1.000000	1.000000
6	Blue	0.000000	0.000000	1.000000
7	Magenta	1.000000	0.000000	1.000000
8	Black	0.000000	0.000000	0.000000
9	Olive green	0.800000	0.733333	0.200000
10	Aqua	0.200000	0.400000	0.466667
11	Royal blue	0.533333	0.400000	0.666667
12	Violet	0.800000	0.266667	0.400000
13	Brick red	1.000000	0.400000	0.200000
14	Burnt orange	1.000000	0.466667	0.000000
15	Grey brown	0.866667	0.533333	0.266667

Colors 9 though 15 are a graphic designers idea of colors for business graphics. Color table entries not shown above are set to white.

Raster Drawing Modes

For raster devices (e.g., HP 9836 display) the effect of the color selectors depends on the current drawing mode (drawing mode is set using the OUTPUT_ESC function). The color selectors and their effects are listed below:

Mode	Color Selector = 0	Color Selector >=1
DOMINATE (Default mode)	Background (erase, set bits to 0)	Draw (set bits to 1, overwrite current pattern)
NON-DOMINATE	Background (erase, set bits to 0)	Draw (set bits to 1 Inclusive OR with current pattern)
ERASE	Background (erase, set bits to 0)	Background (erase, set bits to 0)
COMPLEMENT	Background (erase, set bits to 0)	Complement (Invert bits in selected planes)

Plotters

A Color Selector of 0 selects no pens (the current pen is put away). The supported range of Color Selectors for each supported plotter is:

- 9872A - 0 thru 4
- 9872B - 0 thru 4
- 9872C/S/T - 0 thru 8
- 7580A/7585A - 0 thru 8
- 7470A - 0 thru 2

Error Conditions

The graphics system must be initialized and a display must be enabled or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_COLOR_MODEL

IMPORT: dgl_lib

This **procedure** chooses the color model for interpreting parameters in the color table.

Syntax

Item	Description/Default	Range Restrictions	Recommended Range
model selector	Expression of TYPE INTEGER	MININT thru MAXINT	1 or 2

Procedure Heading
PROCEDURE SET_COLOR_MODEL (MODEL : integer);

Semantics

The **model selector** determines the color model which will be used to interpret the values passed to the color table with SET_COLOR_TABLE or read from it with INQ_COLOR_TABLE.

Value	Meaning
1	RGB (Red-Green-Blue) color cube.
2	HSL (Hue-Saturation-Luminosity) color cylinder.

The RGB physical model is a color cube with the primary additive colors (red, green, and blue) as its axes. With this model, a call to SET_COLOR_TABLE specifies a point within the color cube that has a red intensity value (X-coordinate), a green intensity value (Y-coordinate) and a blue intensity value (Z-coordinate). Each value ranges from zero (no intensity) to one.

Effects of RGB color parameters

Parm 1 (RED)	Parm 2 (GREEN)	Parm 3 (BLUE)	Resultant color
1.0	1.0	1.0	White
1.0	0.0	0.0	Red
1.0	1.0	0.0	Yellow
0.0	1.0	0.0	Green
0.0	1.0	1.0	Cyan
0.0	0.0	1.0	Blue
1.0	0.0	1.0	Magenta
0.0	0.0	0.0	Black

The HSL perceptual model is a color cylinder in which:

- The angle about the axis of the cylinder, in fractions of a circle is the hue (red is at 0, green is at 1/3 and blue is at 2/3).
- The radius is the saturation. Along the center axis of the cylinder, (saturation equal zero) the colors range from white through grey to black. Along the outside of the cylinder (saturation equal one) the colors are saturated with no apparent whiteness.
- The height along the center axis is the luminosity (the intensity or brightness per unit area). Black is at the bottom of the cylinder (luminosity equal zero) and the brightest colors are at the top of the cylinder (luminosity equal one) with white at the center top.

Hue (angle), saturation (radius), and luminosity (height) all range from zero to one. Using this model, a call to SET_COLOR_TABLE specifies a point within the color cylinder that has a hue value, a saturation value, and a luminosity value.

Effects of HSL color parameters

Parm 1 (Hue)	Parm 2 (Sat)	Parm 3 (Lum)	Resultant color
Don't Care	0.0	1.0	White
0.0	1.0	1.0	Red
1/6	1.0	1.0	Yellow
2/6	1.0	1.0	Green
3/6	1.0	1.0	Cyan
4/6	1.0	1.0	Blue
5/6	1.0	1.0	Magenta
Don't Care	Don't Care	0.0	Black

When a call to SET_COLOR_MODEL switches color models, parameter values in subsequent calls to SET_COLOR_TABLE then refer to the new model. Switching models does not affect color definitions that were previously made using another model. Note that when the value of a color table entry is inquired (INQ_COLOR_TABLE), it is returned in the current model, which may not be the model in which it was originally specified.

Not all color specifications can be displayed on every graphics device, since the devices which the graphics library supports differ in their capabilities. If color specification is not available on a device, the graphics system will request the closest available color.

Error Conditions

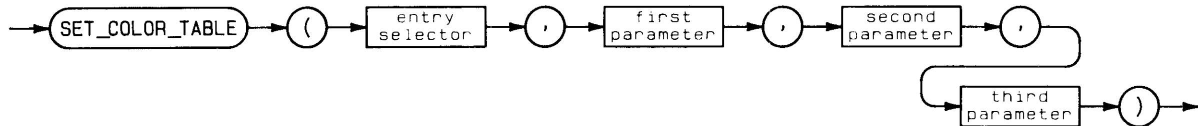
The graphics system must be initialized and the color selector must evaluate to 0 or 1 or this call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_COLOR_TABLE

IMPORT: dgl_lib

This **procedure** redefines the color description of the specified entry in the color table. This color definition is used when the color index is selected via SET_COLOR.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
entry selector	Expression of TYPE INTEGER	MININT to MAXINT	device dependent (see below)
first parameter	Expression of TYPE REAL	0 thru 1	–
second parameter	Expression of TYPE REAL	0 thru 1	–
third parameter	Expression of TYPE REAL	0 thru 1	–

Procedure Heading

```
PROCEDURE SET_COLOR_TABLE ( Index : INTEGER;
                            Colp1 : REAL;
                            Colp2 : REAL;
                            Colp3 : REAL     );
```

Semantics

SET_COLOR_TABLE is ignored by some devices (such as pen plotters) which do not allow their color table to be changed. The procedure INQ_WS (opcode 1073) tells whether the color table can be changed.

The **entry selector** specifies the location in the color capability table that is to be redefined. For raster displays in Series 200 computers, 32 entries are available.

The **first parameter** represents red intensity if the RGB model has been selected with the SET COLOR statement, or hue if the HSL model has been selected.

The **second parameter** represents green intensity if the RGB model has been selected with the SET COLOR statement, or saturation if the HSL model has been selected.

The **third parameter** represents blue intensity if the RGB model has been selected, or luminosity if the HSL model has been selected.

A more detailed description of the color models and the meaning of their parameters can be found under the procedure definition of SET_COLOR_MODEL.

The effect of redefinition of the color table on previously output primitives is device dependent. On most devices changing the color table will only affect future primitives; however, on the Model 36C changing a color table entry with a color selector from 0 through 15 will immediately change the color of primitives previously drawn with that entry. The procedure INQ_WS (opcode 1071) tells whether retroactive color change is supported.

Monochromatic Displays
All Series 200 computers except the Model 36C have a monochromatic internal CRT. Changing an entry in the table will not affect the current display; however, future changes to the display will use the new contents of the table. Device dependent polygons use the color table entry when performing dithering.

The color that lines are drawn with (black or white) is determined from the perceived intensity of the color table entry. This is calculated as follows:

```
if (red * 0.3 + green * 0.59 + blue * 0.11) > 0.1
    then
      color := white
    else
      color := black;
```

The HP 98627A Display
Changing an entry in the table will not affect the current display; however, future changes to the display will use the new contents of the table. Device dependent polygons use the color table entry when performing dithering.

The color that lines are drawn with (one of the 8 non-dithered colors) is determined from the closest HSL value to the requested value.

The Model 36C
The first 16 locations (0..15) of the color table map directly to the hardware color map. Changing one of these color table locations will immediately change the display (assuming the color has been used).

The next 16 locations (16..31) will not affect the current display; however, future changes to the display will use the new contents of the color table.

Device dependent polygons drawn with color table locations 0..15 will be drawn in a solid color without using dithering. When drawn with color table location above 15 dithering will be used.

Note

Since dithering on the HP 9836C uses the current color map values (i.e., color table locations 0..15) changing the first 16 color table locations will affect the dither pattern used. This leads to two major effects. First, changing the first 16 locations after a polygon was generated using dithering will change the dither pattern such that its averaged color no longer matches the color that it was generated with. Second, since the dither pattern is based on the first 16 colors, the first 16 colors can be set to produce a dither pattern with minimum color changes between pixels within the pattern. The following example produces a continuous shaded polygon across the crt:

```
$RANGE OFF$
PROGRAM T;

IMPORT dgl_types, dgl_lib, dgl_poly;

VAR I           : INTEGER;
    Xvec,Yvec   : ARRAY [1..2] OF REAL;
    Ovec        : ARRAY [1..2] OF Gshortint;
    C           : REAL;

BEGIN
  GRAPHICS_INIT;
  DISPLAY_INIT(3,0,i);
  SET_ASPECT(511,389);
  SET_WINDOW(0,511,0,389);

FOR I := 0 to 15 DO
SET_COLOR_TABLE(I,I/15,I/15,I/15);  { set up color map }

SET_PGN_COLOR ( 16 );
SET_PGN_STYLE ( 16 );

Yvec[1] := 100; Yvec[2] := 150; Ovec[1] := 2; Ovec[2] := 0;
FOR I := 0 to 511 DO
BEGIN
  Xvec[1] := I; Xvec[2] := I;
  C : 1-I/511;
  SET_COLOR_TABLE(16,C,C,C); { set polygon color }
  POLYGON_DEV_DEP(2,Xvec,Yvec,Ovec);
 END;
END.
```

The color that lines are drawn with (one of the first 16 non-dithered colors) is determined from the closest HSL value to the requested value.

Dithered Polygon Fills

All the raster displays use a technique called dithering for filling device dependent polygons. The polygon is divided into 4 pixel by 4 pixel 'dither cells'. The colors that are placed in each pixel location inside the dither cells average to the current polygon color. The eye will average the pixels, and see the intended color.

The 98627A has 3 memory planes thus, providing 8 non-dithered colors (white, red, green, blue, cyan, magenta, and black). Using dithering 4913 polygon colors may be generated. To obtain a polygon color of half-tone yellow (R = 0.5 G = 0.5 B = 0.0) the dither cell would contain 8 black pixels and 8 yellow pixels.

On black and white displays, the largest r,g,b value of the current_polygon color is used to determine the dither pattern.

On the HP 9836C the current values of the color map are used to determine the dither cell pixel colors. This leads to a very very large number of colors that the HP 9836C can produce when performing device dependent polygon fill.

The Background Color

Color index 0 represents the background color. The ability to redefine this index is device-dependent. Many devices do not allow the redefinition of their background color. Whether a display device has the ability to redefine the background color can be inquired via a call to INQ_WS with opcode = 1072. All raster displays in the 200 Series are capable of redefining the background color.

Error Conditions

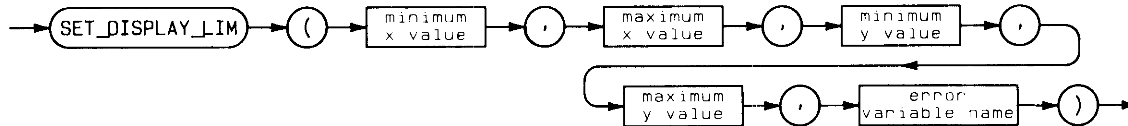
The graphics system must be initialized and a display device must be enabled or this call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_DISPLAY_LIM

IMPORT: dgl_lib

This **procedure** redefines the logical display limits of the graphics display.

Syntax



Item	Description/Default	Range Restrictions
minimum x value	Expression of TYPE REAL	–
maximum x value	Expression of TYPE REAL	–
minimum y value	Expression of TYPE REAL	–
maximum y value	Expression of TYPE REAL	–
error variable name	Variable of TYPE INTEGER	–

Procedure Heading

```
PROCEDURE SET_DISPLAY_LIM (      Xmin, Xmax,
                                 Ymin, Ymax : REAL,
                           VAR         Ierr : INTEGER );
```

Semantics

The **minimum x value** is the distance in millimetres that the left side of the logical display limits is offset from the left side of the physical display limits.

The **maximum x value** is the distance in millimetres that the right side of the logical display limits is offset from the left side of the physical display limits.

The **minimum y value** is the distance in millimetres that the bottom of the logical display limits is offset from the bottom of the physical display limits.

The **maximum y value** is the distance in millimetres that the top of the logical display limits is offset from the bottom of the physical display limits.

The **error variable** will contain an integer indicating whether the limits were successfully set.

Value	Meaning
0	The display limits were successfully set.
1	The minimum x value was greater than or equal to the maximum x value and/or the minimum y value was greater than the maximum y value.
2	The parameters specified were outside the physical display limits.

If the error variable is non-zero, the call was ignored.

SET_DISPLAY_LIM allows an application program to specify the region of the display surface where the image will be displayed. The limits of this region are defined as the logical display limits. Upon initialization, the graphics system sets these limits equal to some portion of the specified physical device. This routine allows a programmer to set the plotting surface of a very large plotter equal to the size of an 8 1/2 x 11 inch paper, for example.

The pairs (minimum x value, minimum y value) and (maximum x value, maximum y value) define the corner points of the new logical display limits in terms of millimetres offset from the origin of the physical display. The exact position of the physical display origin is device dependent. The specifics of various devices are covered later in this entry.

This procedure causes a new virtual coordinate system to be defined. SET_DISPLAY_LIM calculates the new limits of the virtual coordinate system as a function of the current aspect ratio and the new limits of the logical display. This does not affect the limits of the viewport. Since it changes the size of the area onto which the viewport is mapped, it may scale the size of the image displayed. It will not distort the image; it can only make it smaller or larger.

SET_DISPLAY_LIM should only be called while the graphics display is enabled.

Neither the value of the starting position nor the location of the physical pen or beam is altered by this routine. Since this routine may redefine the viewing transformation, the starting position may be mapped to a different coordinate on the display surface. A call to MOVE or INT_MOVE should therefore be made after this call to update the value of the starting position and in so doing, place the physical pen or beam at a known location.

If the logical display and logical locator are associated with the same physical device, a call to SET_DISPLAY_LIM will set the logical locator limits equal to the new limits of the virtual coordinate system. A call to SET_DISPLAY_LIM also sets the locator echo position to its default value, the center of the world coordinate system.

Display Limits of Raster Devices

The internal CRT's for Series 200 computers have the following limits:

Computer	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
Model 216	160	120	400	300	.75	2.5
Model 217	230	175	512	390	.7617	2.226
Model 220 (HP82913A)	210	158	400	300	.75	1.905
Model 220 (HP82912A)	152	114	400	300	.75	2.632
Model 226	120	88	400	300	.75	3.333
Model 236	210	160	512	390	.7617	2.438
Model 236 Color	217	163	512	390	.7617	2.39
Model 237	312	234	1024	768	.75	3.282

The physical size of the HP 98627A display (needed by the SET_DISPLAY_LIM procedure) may be given to the graphics system by an escape function (OPCODE = 250). The physical limits assumed until the escape function is given are:

CONTROL	=	256	153.3mm wide and 116.7mm high.
		512	153.3mm wide and 116.7mm high.
		768	153.3mm wide and 142.2mm high.
		1024	153.3mm wide and 153.3mm high.
		1280	153.3mm wide and 153.3mm high.

The default logical display surface of the graphics display device is the maximum physical limits of the screen. The physical origin is the lower left corner of the display.

The view surface is always centered within the current logical display surface. The origin of a raster display is the lower-left dot.

HPGL Plotter Display Limits

Plotter	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
9872	400	285	16000	11400	.7125	40.0
7580	809.5	524.25	32380	20970	.6476	40.0
7585	1100	891.75	44000	35670	.8107	40.0
7586	1182.8	898.1	47312	35924	.7593	40.0
7470	257.5	191.25	10300	7650	.7427	40.0
7550	411.25	254.25	16450	10170	.6182	40.0
7475	416	259.125	16640	10365	.6229	40.0

The maximum physical limits of the graphics display for a HPGL device not listed above are determined by the default settings of P1 and P2. The default settings of P1 and P2 are the values they have after an HPGL 'IN' command. Refer to the specific device manual for additional details.

The default logical display surface is set equal to the area defined by P1 and P2 at the time DISPLAY_INIT is invoked. The view-surface is always justified in the lower left corner of the current logical display surface (corner nearest the turret for the HP 7580 and HP 7585 plotters). The physical origin of the graphics display is at the lower left boundary of pen movement.

Note

If the paper is changed in an HP 7580, HP 7585 or HP 7586 plotter while the graphics display is initialized, it should be the same size of paper that was in the plotter when DISPLAY_INIT was called. If a different size of paper is required, the device should be terminated (DISPLAY_TERM) and re-initialized after the new paper has been placed in the plotter.

Error Conditions

The graphics system must be initialized and a display device enabled or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_ECHO_POS

IMPORT: dgl_lib

This **procedure** defines the locator echo position on the graphics display.

Syntax

Item	Description/Default	Range Restrictions
x coordinate	Expression of TYPE REAL	–
y coordinate	Expression of TYPE REAL	–

Procedure Heading

```
PROCEDURE SET_ECHO_POS ( Wx, Wy : REAL );
```

Semantics

The **x** and **y coordinate** pair is the new echo position in world coordinates.

When echoing on the display device, SET_ECHO_POS allows a programmer to define the position of the locator echo position. This is a point in the world coordinate system that represents the initial position of the locator. It is used with certain locator echoes on the graphics display. For example, it is used as the anchor point when a rubber band echo is performed. With this echo, the graphics cursor is initially turned on at the locator echo position. From that time on, the cursor reflects the position of the locator and a line extends from the locator echo position to the locator as it moves around the graphics display. To be used in echoing, the point must be displayable. Therefore, if the point specified is outside of the limits of the window the call is ignored.

The locator echo position will only be used when AWAIT_LOCATOR is called with echo types 2 through 8, e.g., type 4 is a rubber band line echo. The locator echo position is only used when the locator echo is being sent to the graphics display device, and is not used when sampling the locator.

SET_ECHO_POS should only be called while the graphics display and locator are initialized. If the point passed to SET_ECHO_POS is outside the current window limits, then the call to SET_ECHO_POS is ignored and no error is given.

The default locator echo position is the center of the limits of the window. When the locator is initialized, the locator echo position is set to the default value. When a call is made which affects the viewing transformations for the graphics display surface or the logical locator limits, the locator echo position is set to the default value. The calls which cause this are SET_ASPECT, DISPLAY_INIT, SET_DISPLAY_LIM, LOCATOR_INIT, SET_LOCATOR_LIM, SET_WINDOW, and SET_VIEWPORT.

Once the locator echo position is set, it retains this value until the next call to SET_ECHO_POS or until a call is made which resets it to the default value.

Error Conditions

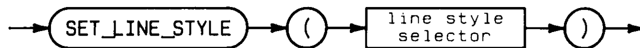
The graphics system must be initialized, and a display device and a locator device must be enabled, or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_LINE_STYLE

IMPORT: dgl_lib

This **procedure** sets the line style attribute.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
line style selector	Expression of TYPE INTEGER	MININT thru MAXINT	Device Dependent

Procedure Heading

```
PROCEDURE SET_LINE_STYLE (Line_Style : INTEGER);
```

Semantics

The **line style selector** is the line style to be used for lines, polylines, polygon edges, and text.

Markers are not affected by line-style. Polygon interior line-style is selected with SET_PGN_LS.

SET_LINE_STYLE sets the line style attribute for lines and text. The mapping between the value of the line style attribute and the line style selected is device dependent. If a line style attribute is requested that the device cannot perform exactly as requested, line style 1 will be performed.

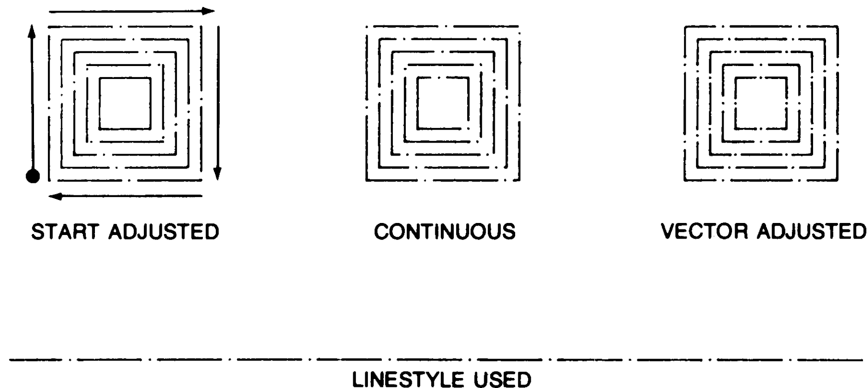
There are three types of line-styles: start adjusted, continuous, and vector adjusted:

Start adjusted line-styles always start the cycle at the beginning of the vector. Thus if the current line-style starts with a pattern, each vector drawn will start with that pattern. Likewise, if the current line-style starts with a space and then a dot, each vector will be drawn starting with a space and then a dot. In this case if the vectors are short, they might not appear at all.

Continuous line styles are generated such that the pattern will be started with the first vector drawn. Subsequent vectors will be continuations of the pattern. Thus, it may take several vectors to complete one cycle of the pattern. This type of line-style is useful for drawing smooth curves, but does not necessarily designate either endpoint of a vector. A side effect of this type of line-style is if a vector is small enough it might be composed only of the space between points or dashes in the line-style. In that case, the vector may not appear on the graphics display at all.

Vector adjusted line-styles treat each vector individually. Individual treatment guarantees that a solid component of the dash pattern will be generated at both ends of the vector. Thus, the endpoints of each vector will be clearly identifiable. This type of line-style is good for drawing rectangles. The integrity of the line-style will degenerate with very small vectors. Since some component of the dash pattern must appear at both ends of the vector, the entire vector for a short vector will often be drawn as solid.

The following figure illustrates how one pattern would be displayed using each one of the different line-style types:

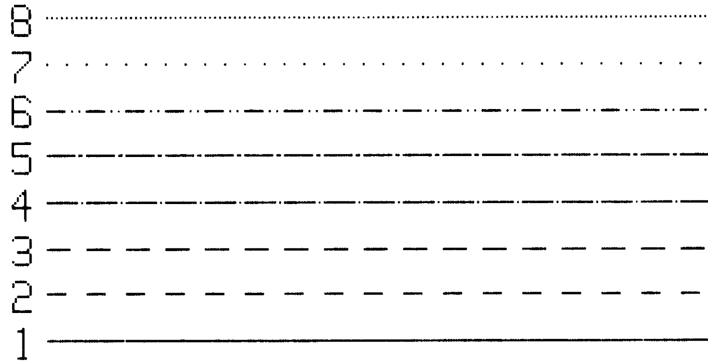


It should be apparent from the above discussion that drawing to the starting position will generate a point (the shortest possible line) only if the line-style is such that the pen is down (or the beam is on) at the start of that vector. Likewise, whole vectors may not appear on the graphics display surface if the line-style is such that the vector is smaller than the blank space in the line-style. The device handlers section of this document details the line-styles available for each device.

Note

When using continuous line styles, complement and erase drawing modes (available on some raster displays e.g., HP 9826) may not completely remove lines previously drawn. This happens since the line style pattern may not be in sync with the first line when the second line is drawn. By setting the line-style to solid when using complement and erase drawing modes the application program can insure that the line is completely removed.

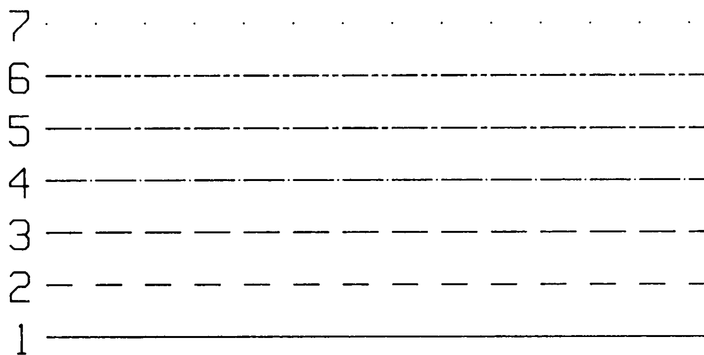
Raster Line Styles
Eight pre-defined line-styles are supported on the graphics display. All of the line-styles may be classified as being "continuous":

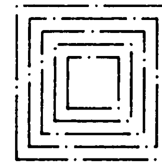


Raster Line Styles

Plotter Line Styles
The following table describes the line styles available on the supported plotters.

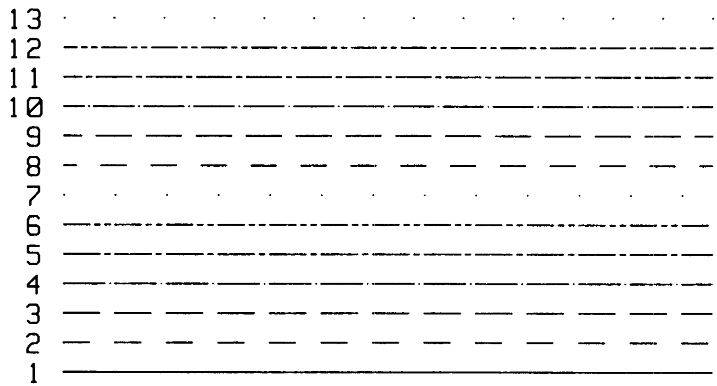
Device	Number of continuous line-styles	Number of vector adjusted line-styles
9872	7	0
7580	7	6
7585	7	6
7470	7	0
Other	7	0

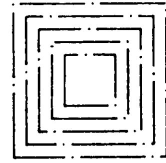




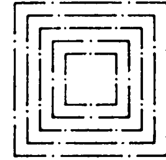
CONTINUOUS

HP 9872 and 7470 Line Styles
(all are continuous)





CONTINUOUS



VECTOR ADJUSTED

HP 7580, 7585 and 7586 Line Styles

If the line style specified is not supported by the graphics display, the call is completed with LINE_STYLE = 1 and no error is reported.

Error Conditions

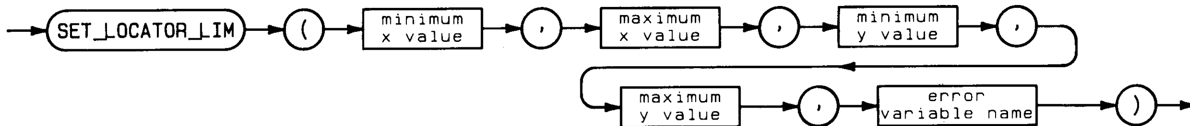
The graphics system must be enabled and a display device must be enabled or this call will be ignored and GRAPHICSERROR will return a non-zero value.

SET_LOCATOR_LIM

IMPORT: dgl_lib

This **procedure** redefines the logical locator limits of the graphics locator.

Syntax



Item	Description/Default	Range Restrictions
minimum x value	Expression of TYPE REAL	–
maximum x value	Expression of TYPE REAL	–
maximum y value	Expression of TYPE REAL	–
minimum y value	Expression of TYPE REAL	–
error variable name	Variable of TYPE INTEGER	–

Procedure Heading

```
PROCEDURE SET_LOCATOR_LIM (      Xmin, Xmax,
                                 Ymin, Ymax  : REAL,
                             VAR Ierr        : INTEGER );
```

Semantics

The **minimum x value** is the distance in millimetres that the left side of the logical locator limits is offset from the left side of the physical locator limits.

The **maximum x value** is the distance in millimetres that the right side of the logical locator limits is offset from the left side of the physical locator limits.

The **minimum y value** is the distance in millimetres that the bottom of the logical locator limits is offset from the bottom of the physical locator limits.

The **maximum y value** is the distance in millimetres that the top of the logical locator limits is offset from the bottom of the physical locator limits.

The **error variable** will contain an integer indicating whether the limits were successfully set.

Value	Meaning
0	The display limits were successfully set.
1	The minimum x value was greater than or equal to the maximum x value and/or the minimum y value was greater than the maximum y value.
2	The parameters specified were outside the physical display limits.
3	Attempt to explicitly define locator limits on a device which is both the logical locator and the logical display. The logical display limits are used when a device is shared for both purposes, and they cannot be redefined with this call.

If the error variable is non-zero, the call was ignored.

SET_LOCATOR_LIM allows an application program to specify the portion of the physical locator device that should be used to perform locator functions. When the logical locator device is enabled (via LOCATOR_INIT) the logical device limits are set to a device dependent portion of the physical locator device. With a call to this routine the user can set the logical locator limits by specifying a new area within the physical locator limits.

The pairs (minimum x value, minimum y value) and (maximum x value, maximum y value) define the corner points of the new logical locator limits in terms of millimetres offset from the origin of the physical locator. The exact position of the physical locator origin is device dependent. Specific origins are covered later in this entry.

If a logical locator and a logical display are associated with the same physical device, then the logical locator limits must be the same as the logical view surface limits. Specifically, the effects of the association with the same physical device are as follows:

- The logical locator limits are initialized to the same values as the virtual coordinate system.
- Any call which redefines the virtual coordinate system limits will also redefine the logical locator limits.
- The logical locator limits can not be defined by a call to SET_LOCATOR_LIM.

By changing the logical locator limits any portion of the graphics locator can be addressed, with the restrictions stated above.

The logical locator limits always map directly to the view surface, therefore, distortion may result in the mapping between the logical locator and the display when the logical locator limits and the view surface have different aspect ratios. If the distortion is not desired it can be avoided by assuring that the logical locator limits maintain the same aspect ratio as that of the view surface.

SET_LOCATOR_LIM should only be called while the graphics locator is enabled. SET_LOCATOR_LIM sets the locator echo position to the default value (see SET_ECHO_POS).

Relative Locator Limits (Knob or Mouse)
The knob may be used as a locator on Series 200 computers. The default characteristics of the knob on various Series 200 computers is listed in the table below.

Computer	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
Model 216	160	120	400	300	.75	2.5
Model 217	230	175	512	390	.7617	2.226
Model 220 (HP82913A)	210	158	400	300	.75	1.905
Model 220 (HP82912A)	152	114	400	300	.75	2.632
Model 226	120	88	400	300	.75	3.333
Model 236	210	160	512	390	.7617	2.438
Model 236 Color	217	163	512	390	.7617	2.39
Model 237	312	234	1024	768	.75	3.282

The knob uses the current display limits as its locator limits for locator echoes 2 though 8. For all other echoes the above limits are used. An example of when the two limits may differ follows:

> The knob locator is initialized on an HP 9826. The graphics display is an HP 98627A color output card. The resolution of the locator is 0 through 399 in x dimension, and 0 through 299 in y dimension. The resolution of the display is 0 through 511 in x dimension, and 0 through 389 in y dimension. When await_locator is used with echo 4, the locator will effectively have the HP 98627A resolution for the duration of the await_locator call. However if echo 1 is used with await_locator, the cursor will appear on the HP 9826 and the locator has a resolution of 0×399 and 0×299. Note that all conversion routines, and inquiries will use the HP 9826 limits.

The physical origin of the locator device is the lower left corner of the display.

Absolute Locator Limits (HPGL Plotter or Graphics Tablet)
HPGL plotter and graphics tablets can be used as locators. The default characteristics of some HPGL devices are listed below.

Plotter/ Tablet	Wide mm	High mm	Wide points	High points	Aspect	Resolution points/mm
9872	400	285	16000	11400	.7125	40.0
7580	809.5	524.25	32380	20970	.6476	40.0
7585	1100	891.75	44000	35670	.8107	40.0
7586	1182.8	898.1	47312	35924	.7593	40.0
7470	257.5	191.25	10300	7650	.7427	40.0
7550	411.25	254.25	16450	10170	.6182	40.0
7475	416	259.125	16640	10365	.6229	40.0
9111	300.8	217.6	12032	8704	.7234	40.0

The maximum physical limits of the locator for a HPGL device not listed above are determined by the default settings of P1 and P2. The default settings of P1 and P2 are the values they have after an HPGL 'IN' command. Refer to the specific device manual for additional details.

The default logical display surface is set equal to the area defined by P1 and P2 at the time LOCATOR_INIT is invoked.

Note

If the paper is changed in an HP 7580, HP 7585 or HP 7586 plotter while the graphics display is initialized, it should be the same size of paper that was in the plotter when DISPLAY_INIT was called. If a different size of paper is required, the device should be terminated (DISPLAY_TERM) and re-initialized after the new paper has been placed in the plotter.

Error Conditions

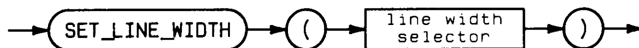
The graphics system must be initialized and a display device enabled or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_LINE_WIDTH

IMPORT: dgl_lib

This **procedure** sets the line-width attribute. The number of line-widths possible is device dependent.

Syntax



Item	Description/Default	Range Restrictions
line-width selector	Expression of TYPE INTEGER	MININT thru MAXINT

Procedure Headings

```
PROCEDURE SET_LINE_WIDTH ( Linewidth : INTEGER );
```

Semantics

SET_LINE_WIDTH sets the line-width attribute for lines, polylines and text. The line-width attribute does not affect markers which are defined to be always output with the thinnest line-width supported on the device. All devices support at least one line-width. The range of line-widths is device dependent but line-width 1 is always the thinnest line-width supported. For devices that support multiple line-widths, the line-width increases as line-width does until the device supported maximum is reached. For example, line-width = 1 specifies the thinnest, line-width = 2 specifies the next wider line-width, etc.

If line-width is greater than the number of line-widths supported by the graphics display or line-width is less than 1, then the line-width will be set to the thinnest available width (line-width = 1). All subsequent lines and text will then be drawn with the thinnest available line-width. A call to INQ_WS with OPCODE equal to 1063 to inquire the value of the line-width will then return a 1.

The initial line-width is the thinnest width supported by the device (line-width = 1).

Note

All current devices support a single line-width.

Error Conditions

The graphics system must be initialized and a display device must be enabled or this call is ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_PGN_COLOR

IMPORT: dgl_lib
dgl_poly

This **procedure** selects the polygon interior color attribute for subsequently generated polygons by providing a selector for the color table.

Syntax

Item	Description/Default	Range Restrictions	Recommended Range
color selector	Expression of TYPE INTEGER	MININT thru MAXINT	Device dependent.

Procedure Heading

```
PROCEDURE SET_PGN_COLOR ( Cindex : INTEGER );
```

Semantics

The **color selector** is an index into the color table. The contents of the color table are then used to specify the color when primitives are drawn. On some devices (HPGL plotters), the color selector maps directly to a pen number for the device. On the HP 9836C, the entries in the color table can be modified with SET_COLOR_TABLE. The color actually used depends on the value in a device dependent color table.

At device initialization a default color table is created by the graphics system. The size and contents of the table are device dependent. At least one entry exists for all devices. A call to INQ_WS with OPCODE equal to 1053 will return the number of colors available on a given graphics device. Some devices allow the color table to be modified with SET_TABLE.

The default value of the color attribute is 1. If the value of the color selector is not supported on the graphics display, the color attribute will be set to 1.

A color selector of 0 has special effects depending on the graphics display used. For raster devices, a color selector of 0 means to draw in the background color. For most plotters, it puts the pen away.

Dithering

If the device is not capable of reproducing a color in the color table, the closest color which the device is capable of reproducing is used instead. For polygon fill (in a device dependent mode) this may involve dithering. For example, the HP 98627A color output interface card is capable of a large selection of polygon fill colors, but only 8 line colors. Thus, the fill color could match the selected color much more closely than the line color used to outline the polygon. See SET_COLOR_TABLE for details on how colors are matched to the devices.

Default Raster Color Map

The following table shows the default (initial) color table for the black and white displays (HP 9816 / HP 9920 / HP 9826 / HP 9836):

Index #	Hue	Saturation	Luminosity
0	0	0	0
1	0	0	1.0000
2	0	0	0.9375
3	0	0	0.8750
4	0	0	0.8125
5	0	0	0.7500
6	0	0	0.6875
7	0	0	0.6250
8	0	0	0.5625
9	0	0	0.5000
10	0	0	0.4375
11	0	0	0.3750
12	0	0	0.3125
13	0	0	0.2500
14	0	0	0.1875
15	0	0	0.1250
16	0	0	0.0625

Colors 17 though 31 are set to white.

The following table shows the default (initial) color table for the color displays (HP 9836C and HP 98627A):

Index #	Color name	Red	Green	Blue
0	Black	0.000000	0.000000	0.000000
1	White	1.000000	1.000000	1.000000
2	Red	1.000000	0.000000	0.000000
3	Yellow	1.000000	1.000000	0.000000
4	Green	0.000000	1.000000	0.000000
5	Cyan	0.000000	1.000000	1.000000
6	Blue	0.000000	0.000000	1.000000
7	Magenta	1.000000	0.000000	1.000000
8	Black	0.000000	0.000000	0.000000
9	Olive green	0.800000	0.733333	0.200000
10	Aqua	0.200000	0.400000	0.466667
11	Royal blue	0.533333	0.400000	0.666667
12	Violet	0.800000	0.266667	0.400000
13	Brick red	1.000000	0.400000	0.200000
14	Burnt orange	1.000000	0.466667	0.000000
15	Grey brown	0.866667	0.533333	0.266667

Colors 9 though 15 are a graphic designers idea of colors for business graphics. Color table entries not shown above are set to white.

Raster Drawing Modes

Raster drawing modes have no effect on polygon fill color.

Plotters

A Color Selector of 0 selects no pens (the current pen is put away). The supported range of Color Selectors for each supported plotter is:

- 9872A - 0 thru 4
- 9872B - 0 thru 4
- 9872C/S/T - 0 thru 8
- 7550A/7580A/7585A/7586B - 0 thru 8
- 7470A - 0 thru 2
- 7475 - 0 thru 6

Error Conditions

The graphics system must be initialized and a display must be enabled or this call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR returns a non-zero value.

SET_PGN_LS

IMPORT: dgl_lib
dgl_poly

This **procedure** selects the polygon interior line-style attribute for subsequently generated polygons by providing a selector for the device dependent line-style table.

Syntax

Item	Description/Default	Range Restrictions	Recommended Range
line-style selector	Expression of TYPE INTEGER	MININT thru MAXINT	Device dependent

Procedure Heading

```
PROCEDURE SET_PGN_LS ( Lindex : INTEGER );
```

Semantics

The **line style selector** is the line style to be used for polygon interiors.

Line-styles for other primitives are selected using SET_LINE_STYLE.

The mapping between the value of the line style attribute and the line style selected is device dependent. If a line style attribute is requested that the device cannot perform exactly as requested, line style 1 will be performed.

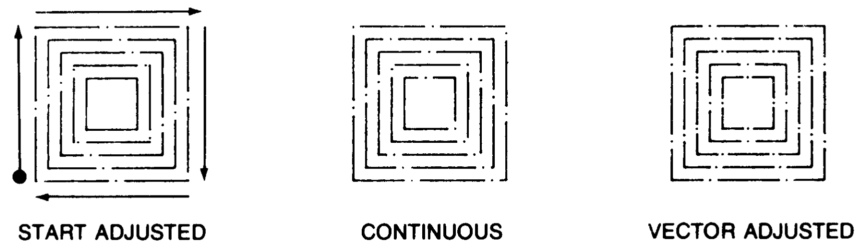
There are three types of line-styles - start adjusted, continuous, and vector adjusted:

Start adjusted line-styles always start the cycle at the beginning of the vector. Thus if the current line-style starts with a pattern, each vector drawn will start with that pattern. Likewise, if the current line-style starts with a space and then a dot, each vector will be drawn starting with a space and then a dot. In this case if the vectors are short, they might not appear at all.

Continuous line styles are generated such that the pattern will be started with the first vector drawn. Subsequent vectors will be continuations of the pattern. Thus, it may take several vectors to complete one cycle of the pattern. This type of line-style is useful for drawing smooth curves, but does not necessarily designate either endpoint of a vector. A side effect of this type of line-style is if a vector is small enough it might be composed only of the space between points or dashes in the line-style. In that case, the vector may not appear on the graphics display at all.

Vector adjusted line-styles treat each vector individually. Individual treatment guarantees that a solid component of the dash pattern will be generated at both ends of the vector. Thus, the endpoints of each vector will be clearly identifiable. This type of line-style is good for drawing rectangles. The integrity of the line-style will degenerate with very small vectors. Since some component of the dash pattern must appear at both ends of the vector, the entire vector for a short vector will often be drawn as solid.

The following figure illustrates how one pattern would be displayed using each one of the different line-style types:

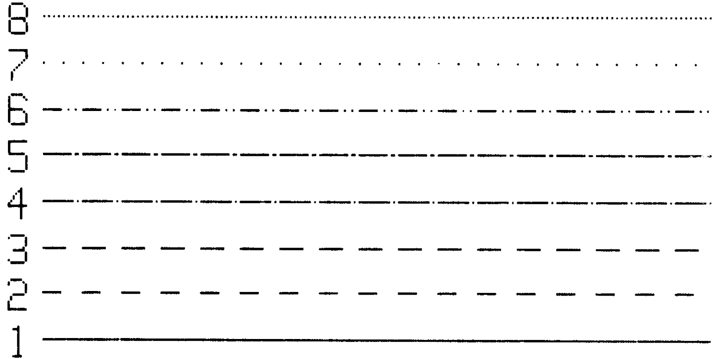


It should be apparent from the above discussion that drawing to the starting position will generate a point (the shortest possible line) only if the line-style is such that the pen is down (or the beam is on) at the start of that vector. Likewise, whole vectors may not appear on the graphics display surface if the line-style is such that the vector is smaller than the blank space in the line-style. The device handlers section of this document details the line-styles available for each device.

Note

When using continuous line styles, complement and erase drawing modes (available on some raster displays e.g., HP 9826) may not completely remove lines previously drawn. This happens since the line style pattern may not be in sync with the first line when the second line is drawn. By setting the line style to solid when using complement and erase drawing modes the application program can insure that the line is completely removed.

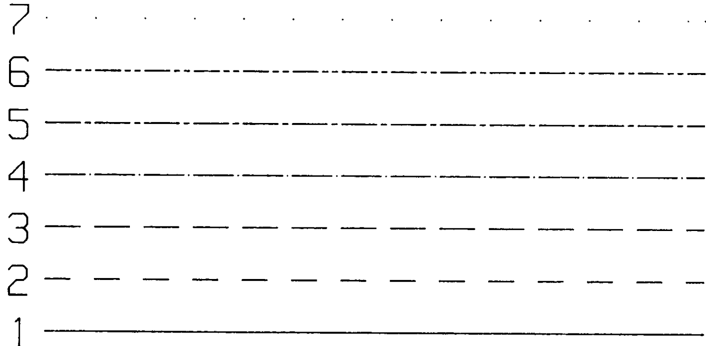
Raster Line Styles
Eight pre-defined line-styles are supported on the graphics display. All of the line-styles may be classified as being "continuous":

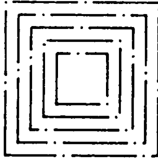


Raster Line Styles

Plotter Line Styles
The following table describes the line styles available on the supported plotters.

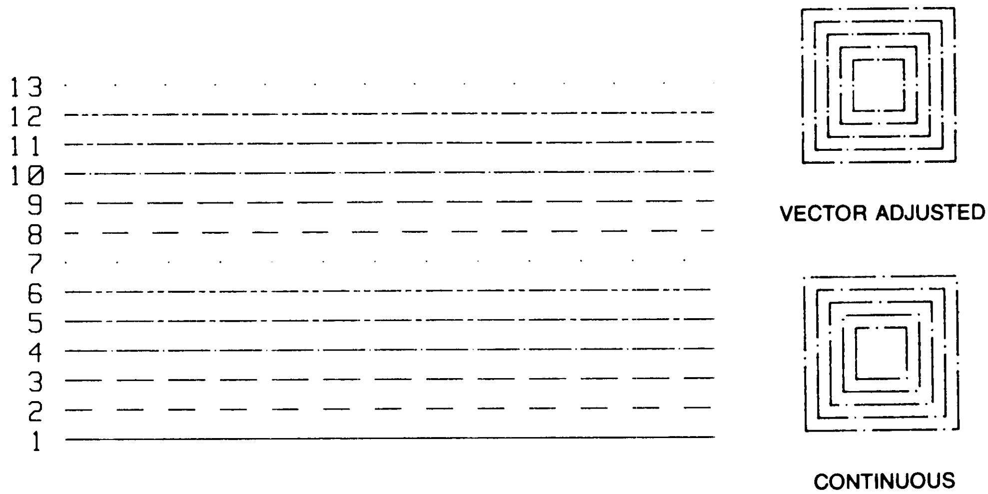
Device	Number of continuous line-styles	Number of vector adjusted line-styles
9872	7	0
7470	7	0
7475	7	0
7550	7	6
7580	7	6
7585	7	6
7586	7	6
Other	7	0





CONTINUOUS

HP 9872, 7470 and 7475 Line Styles
(all are continuous)



HP 7550, 7580, 7585 and 7586 Line Styles

If the line style specified is not supported by the graphics display, the call is completed with LINE_STYLE = 1 and no error is reported.

The graphics system must be enabled and a display device must be enabled or this call will be ignored and GRAPHICSERROR will return a non-zero value.

Error conditions:
The graphics system must be initialized and a display device must be enabled or this call will be ignored, an ESCAPE (– 27) will be generated, and GRAPHICSERROR will return an non-zero value.

SET_PGN_STYLE

IMPORT: dgl_lib
dgl_poly

This **procedure** selects the polygon style attribute for subsequently generated polygons by providing a selector for the polygon style table.

Syntax

Item	Description/Default	Range Restrictions	Recommended Range
polygon style selector	Expression of TYPE INTEGER	MININT thru MAXINT	Device dependent

Procedure Heading

```
PROCEDURE SET_PGN_STYLE ( Pindex : INTEGER );
```

Semantics

Polygon styles can vary in polygon interior density, polygon interior orientation and polygon edge display. See SET_PGN_TABLE for details on default styles, and how the polygon style table may be changed.

Error Conditions

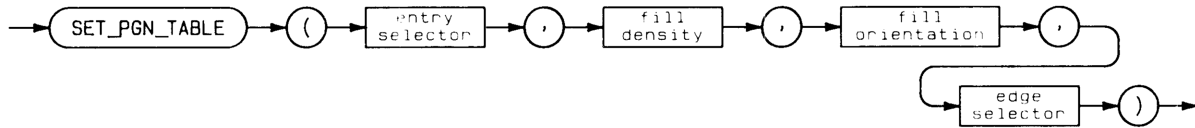
The graphics system must be initialized and a display device must be enabled or this call will be ignored and GRAPHICSERROR will return an non-zero value.

SET_PGN_TABLE

IMPORT: dgl_lib
dgl_poly

This **procedure** defines a polygon style attribute, i.e. an entry in a polygon style table.

Syntax



Item	Description/Default	Range Restrictions	Recommended Range
entry selector	Expression of TYPE INTEGER	MININT thru MAXINT	Device dependent
fill density	Expression of TYPE REAL	MININT thru MAXINT	-1 thru 1
fill orientation	Expression of TYPE REAL	MININT thru MAXINT	-90 thru 90
edge selector	Expression of TYPE INTEGER	MININT thru MAXINT	—

Procedure Heading

```
PROCEDURE SET_PGN_TABLE ( Index   : INTEGER;
                          Densty  : REAL;
                          Orient  : REAL;
                          Edge    : INTEGER );
```

Semantics

This routine defines the attribute of polygon style, i.e. it specifies an entry in a polygon style table. This entry contains information that specifies polygon interior density, polygon interior orientation, polygon edge display, and device-independence of polygon display.

The **entry selector** specifies the entry in the polygon style table that is to be redefined.

The **fill density** determines the density of the polygon interior fill. The magnitude of this value is the ratio of filled area to non-filled area. Zero means the polygon interior is not filled. One represents a fully filled polygon interior. All non-zero values specify the density of continuous lines used to fill the interior.

Positive density values request parallel fill lines in one direction only. Negative values are used to specify crosshatching. For a given density, the distance between two adjacent parallel lines is greater with cross hatching than in the case of pure parallel filling. Calculations for fill density are based on the thinnest line possible on the device and on continuous line-style.

The distance between fill lines – hence density – does not change with a change of scale caused by a viewing transformation. If the interior line-style is not continuous, the actual fill density may not match that found in the polygon style table.

The **fill orientation** represents the angle (in degrees) between the lines used for filling the polygon and the horizontal axis of the display device. The interpretation of fill orientation is device-dependent. On devices that require software emulation of polygon styles, the angle specified will be adhered to as closely as possible, within the line-drawing capabilities of the device. For hardware generated polygon styles, the angle specified will be adhered to as closely as is possible given the hardware simulation of the requested density. If crosshatching is specified, the fill orientation specifies the angle of orientation of the first set of lines in the crosshatching, and the second set of lines is always perpendicular to this.

The value of the **edge selector** determines whether the edge of the polygon is displayed. If the edge selector is 0, the edges will not be displayed. If the edge selector is 1, display of individual edge segments depends on the operation selector in the call that draws the polygon set, POLYGON, INT_POLYGON, POLYGON_DEV_DEP, or INT_POLYGON_DD.

If polygon edges are displayed, they adhere to the current line attributes of color, line-style, and line-width, in effect at the time of polygon display.

A device-dependent number of polygon styles are available. All devices support at least 16 entries in the polygon table. The polygon styles defined in the default tables are defined to exploit the hardware capabilities of the devices they are defined for.

Polygon interiors can be generated in either a device-dependent or device-independent fashion, by calling POLYGON_DEV_DEP or POLYGON respectively.

Polygons generated in a device-dependent fashion will utilize the available hardware polygon generation capabilities of the device to increase the speed and efficiency of polygon generation. The output may vary depending on the device. Devices that have no hardware polygon generation capabilities will only do a minimal representation of the polygon if a device-dependent representation of the polygon is requested. If an edge is not requested, an outline of the non-clipped boundaries of the polygon interior will be drawn in the current polygon interior color and polygon interior line-style if the density of the polygon interior was not zero.

Polygons generated in a device-independent fashion will adhere strictly to the polygon style specification. The polygon interior generated would look similar when generated on different devices for a given polygon style specification. However, on raster devices rasterization of the fill lines may leave empty pixels when solid fill is requested with an orientation that is not 0 or 90 degrees. Available hardware would only be used where the polygon style could be generated exactly as specified.

The number of entries in the polygon style table and the default contents of the table are device dependent. However, all devices support the following polygon style table:

Entry	Density	Angle	Edge
1	0.0	0.0	1
2	0.125	90.0	1
3	0.125	0.0	1
4	−0.125	0.0	1
5	0.125	45.0	1
6	0.125	−45.0	1
7	−0.125	45.0	1
8	0.25	90.0	1
9	0.25	0.0	1
10	−0.25	0.0	1
11	0.25	45.0	1
12	0.25	−45.0	1
13	−0.25	45.0	1
14	−0.5	0.0	1
15	1.0	0.0	0
16	1.0	0.0	1

Error Conditions

The graphics system must be initialized, a display must be enabled, and the parameters must be within the specified limits or this call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_TEXT_ROT

IMPORT: dgl_lib

This **procedure** specifies the text direction.

Syntax

Item	Description/Default	Range Restrictions
x-axis offset	Expression of TYPE REAL	–
y-axis offset	Expression of TYPE REAL	–

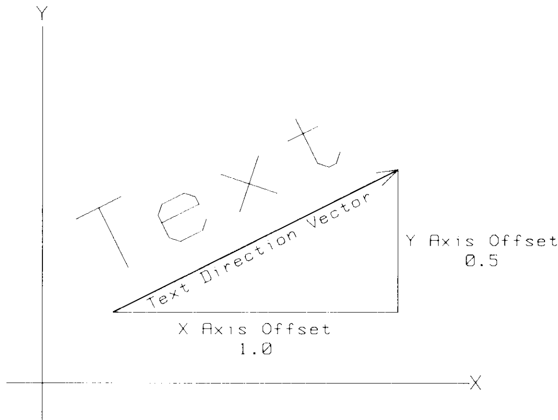
Procedure Heading

```
PROCEDURE SET_TEXT_ROT ( Dx, Dy : REAL );
```

Semantics

The **x axis offset** and the **y axis offset** specify the world coordinate components of the text direction vector relative to the world coordinate origin. These components cannot both be zero.

This procedure specifies the direction in which graphics text characters are output. The default value (X-axis offset = 1.0; Y-axis offset = 0.0) for the text direction vector is such that characters are drawn in a horizontal direction left to right. The default value is set during GRAPHICS_INIT and DISPLAY_INIT. With X-axis offset = - 1.0 and Y-axis offset = 1.0 a 135 degree rotation from the horizontal (in a counter clockwise direction) may be obtained.



Text Rotation Angle

Error Conditions

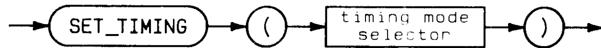
The graphics system must be initialized, a display must be enabled, and the parameters must be within the specified limits or this call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_TIMING

IMPORT: dgl_lib

This **procedure** selects the timing mode for graphics output.

Syntax



Item	Description/Default	Range Restrictions
timing mode selector	Expression of TYPE INTEGER	0 or 1

Procedure Heading

```
PROCEDURE SET_TIMING ( Opcode : INTEGER );
```

Semantics

The **timing mode selector** determines the timing mode used.

Value	Meaning
0	Immediate visibility mode
1	System buffering mode

Graphics library timing modes are provided to control graphics throughput and picture update timing. Picture update timing refers to the immediacy of visual changes to the graphics display surface. Regardless of the timing mode used, the same final picture is sent to the graphics display. SET_TIMING only controls when a picture appears on the graphics display, not what appears.

The graphics system supports two timing modes:

- *Immediate visibility* Requested picture changes will be sent to the graphics display device before control is returned to the calling program. Due to operating system delays there may be a delay before the picture changes are visible on the graphics display device.
- *System buffering* Requested picture changes will be buffered by the graphics system. This means that the graphics output will not be immediately sent to the display device. This allows the graphics library to send several graphics commands to the graphics display device in one data transfer, therefore, reducing the number of transfers. System buffering is the initial timing mode.

The following routines implicitly make the picture current:

AWAIT_LOCATOR	DISPLAY_TERM	INPUT_ESC
LOCATOR_INIT	SAMPLE_LOCATOR	

The immediate visibility mode is less efficient than the system buffering mode. It should only be used in those applications that require picture changes to take place as soon as they are defined, even if the finished picture takes longer to create. When changing the timing mode to immediate visibility the picture is made current.

An alternative to immediate visibility that will solve many application needs is the use of system buffering together with the MAKE_PIC_CURRENT procedure. With this method, an application program places graphics commands into the output buffer and flushes the buffer (see MAKE_PIC_CURRENT) only at times when the picture must be fully displayed.

A call to MAKE_PIC_CURRENT can be made at any time within an application program to insure that the image is fully defined. MAKE_PIC_CURRENT flushes the output buffer but does not modify the timing mode.

Before performing any non-graphics system input or output (to a graphics system device) such as a PASCAL read or write, the output buffer must be empty. If the buffer is not flushed (via immediate visibility of MAKE_PIC_CURRENT) prior to non-graphics system I/O, the resulting image may contain some 'garbage' such as escape functions or invalid graphics data.

Note

Although SET_TIMING can be used with all display devices, only HPGL plotters buffer commands.

Error Conditions

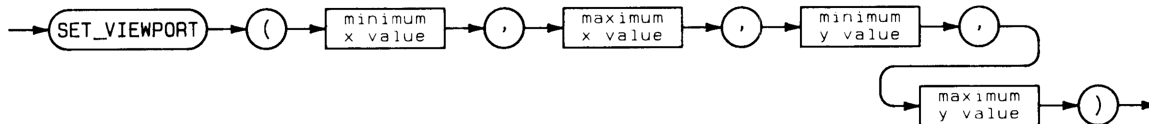
The graphics system must be initialized and all parameters must be in range or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

SET_VIEWPORT

IMPORT: dgl_lib

This **procedure** sets the boundaries of the viewport in the virtual coordinate system.

Syntax



Item	Description/Default	Range Restrictions
minimum x value	Expression of TYPE REAL	0.0-1.0
maximum x value	Expression of TYPE REAL	0.0-1.0
minimum y value	Expression of TYPE REAL	0.0-1.0
maximum y value	Expression of TYPE REAL	0.0-1.0

Procedure Heading

```
PROCEDURE SET_VIEWPORT ( Vxmin, Vxmax,
                         Vymin, Vymax : REAL );
```

Semantics

The **minimum x value** is the minimum boundary in the X-direction expressed in virtual coordinates.

The **maximum x value** is the maximum boundary in the X-direction expressed in virtual coordinates.

The **minimum y value** is the minimum boundary in the Y-direction expressed in virtual coordinates.

The **maximum y value** is the maximum boundary in the Y-direction expressed in virtual coordinates.

SET_VIEWPORT sets the limits of the viewport in the virtual coordinate system. The viewport must be within the limits of the virtual coordinate system; otherwise the call will be ignored.

The initial viewport is set up with the minimum x and y values set to 0.0 and the maximum X and Y values set to 1.0.

The initial viewport is set by GRAPHICS_INIT and SET_ASPECT. This initial viewport is mapped onto the maximum visible square within the logical display limits. This area is called the view surface. The placement of the view surface within the logical display limits is dependent upon the device being used. It is generally centered on CRT displays and is placed in the lower left-hand corner of plotters.

By changing the limits of the viewport, an application program can display an image in several different positions on the same graphics display device. A program can make a call to SET_VIEWPORT anytime while the graphics system is initialized.

The starting position is not altered by this call. Since this call redefines the viewing transformation, the starting position may no longer represent a known world coordinate position. A call to MOVE or INT_MOVE should be made after this call to update the starting position.

Error Conditions

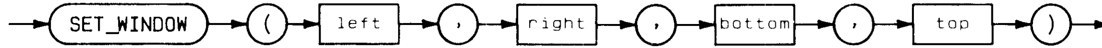
The graphics system must be initialized, all parameters must be within the specified range, the minimum X value must be less than the maximum X value and the minimum Y value must be less than the maximum Y value and all parameters must be within the current virtual coordinate system boundary, or this call will be ignored, an ESCAPE (−27) will be generated, and GRAPHICSERROR will return a non-zero value..

SET_WINDOW

IMPORT: dgl_lib

This **procedure** defines the boundaries of the window.

Syntax



Item	Description/Default	Range Restrictions
left	Expression of TYPE REAL	See below
right	Expression of TYPE REAL	See below
bottom	Expression of TYPE REAL	See below
top	Expression of TYPE REAL	See below

Procedure Heading

```
PROCEDURE SET_WINDOW ( Wxmin, Wxmax,
                       Wymin, Wymax : REAL );
```

Semantics

The **left** is the minimum boundary in the X-direction expressed in world coordinates. (i.e., the left window border). Must not equal maximum x value.

The **right** is the maximum boundary in the X-direction expressed in world coordinates. (i.e. the right window border). Must not equal minimum x value.

The **bottom** is the minimum boundary in the Y-direction expressed in world coordinates. (i.e. the bottom window border). Must not equal maximum y value.

The **top** is the maximum boundary in the Y-direction expressed in world coordinates. (i.e. the top window border). Must not equal minimum y value.

SET_WINDOW defines the limits of the window. All positional information sent to and received from the graphics system is specified in world coordinate units. This allows the application program to specify coordinates in units related to the application.

If the top value is less than the bottom value, the Y-axis will be inverted. If the right value is less than the left boundary, the X-axis will be inverted.

The window is linearly mapped onto the viewport specified by SET_VIEWPORT. This is done by mapping the left boundary to the minimum X-viewport boundary, the right boundary to the maximum X-viewport boundary, the bottom boundary to the minimum Y-viewport boundary, and the top boundary to the maximum Y-viewport boundary. If distortion of the graphics image is not desired, the aspect ratio of the window boundaries should be equal to the aspect ratio of the viewport.

The default window limits range from − 1.0 to 1.0 on both the X and Y axis. GRAPHICS_INIT is the only procedure which sets the window to its default limits.

The starting position is not altered by this call. Since this call redefines the viewing transformation, the starting position may no longer represent a known world coordinate position. A call to MOVE or INT_MOVE should therefore be made after this call to update the starting position.

SET_WINDOW can be called at anytime while the graphics system is initialized.

Error Conditions
The graphics system must be initialized, the minimum value for either axis must not equal the maximum value for that axis or this call will be ignored, an ESCAPE (− 27) will be generated, and GRAPHICSERROR will return a non-zero value.

Module Dependency Table

The Module Dependency Table shows which modules are imported by the standard LIBRARY, IO, GRAPHICS, and SEGMENTER modules.

Module to Be Imported	Module(s) Upon Which It Depends
LIBRARY Modules:	
RND	SYSGLOBALS
HPM	–
UIO	–
LOCKMODULE	SYSGLOBALS
IO Modules:	
IODECLARATIONS	SYSGLOBALS
IOCOMASM	SYSGLOBALS, IODECLARATIONS
GENERAL_0	SYSGLOBALS, IODECLARATIONS
GENERAL_1	SYSGLOBALS, IODECLARATIONS
GENERAL_2	SYSGLOBALS, IODECLARATIONS, GENERAL_1, HPIB_1
GENERAL_3	SYSGLOBALS, IODECLARATIONS
GENERAL_4	SYSGLOBALS, IODECLARATIONS, HPIB_1
HPIB_0	SYSGLOBALS, IODECLARATIONS
HPIB_1	SYSGLOBALS, IODECLARATIONS
HPIB_2	SYSGLOBALS, IODECLARATIONS, HPIB_0, HPIB_1
HPIB_3	SYSGLOBALS, IODECLARATIONS, GENERAL_1, HPIB_0, HPIB_1
SERIAL_0	SYSGLOBALS, IODECLARATIONS
SERIAL_3	SYSGLOBALS, IODECLARATIONS
GRAPHICS (and FGRAPHICS) Modules:	
DGL_LIB	ASM, IODECLARATIONS, SYSGLOBALS, MINI, ISR, MISC, FS, SYSDEVS, and all GRAPHICS modules *except* DGL_INQ and DGL_POLY
DGL_POLY	SYSGLOBALS, SYSDEVS, and all GRAPHICS modules *except* DGL_INQ
DGL_INQ	ASM, SYSGLOBALS, A804XDVR, DGL_TYPES, DGL_VARS, DGL_GEN, GLE_TYPES, GLE_GEN
SEGMENTER Modules:	
SEGMENTER	LOADER, LDR, SYSGLOBALS, MISC

Subject Index

a

Acceleration, pen........ 86
Anisotropic scaling........ 10
Aspect ratio........ 11,34,60,98
Attributes, color........ 99
AWAIT_LOCATOR procedure........ 94,220
Axes:
  Description of........ 20,62
  Labelling........ 25
  Logarithmic........ 64
AxesGrid program........ 62,142

b

Background value........ 121
BAR_KNOB program........ 87,89,149
BAR_KNOB2 program........ 88,152
Bold labels........ 19
Booting the Pascal system........ 7

c

Cartesian coordinates........ 6
Cell, character........ 43
Centering labels........ 17
Character cell........ 43
Character size, setting........ 16,45
CharCell program........ 43,157
CHARSIZE procedure........ 46,50
Choosing the graphics display device........ 6
CLEAR_DISPLAY procedure........ 55,225
CLIPDRAW procedure........ 25,32,62
Clipping lines........ 23
Closed loop system........ 87
Color displays, external........ 84
COLOR program........ 107,158
Color:
  Additional colors........ 116,121,123
  Business........ 100
  CMY Color Cube........ 111
  Dithered colors........ 118,121
  Effective use of........ 133
  Gamuts........ 137
  Graphics........ 99
  Hardcopy........ 137
  HSL Color Cylinder........ 112
  HSL model........ 103,107
  Hue........ 103
  Luminosity........ 103
  Map........ 122
  Mixing........ 134
  Model resolution........ 126
  Models........ 102,107
  Objective use of........ 135
  Primary........ 100
  References........ 139
  RGB Color Cube........ 110
  RGB model........ 102,107
  Saturation........ 103
  Seeing........ 133
  Spaces........ 109
  Subjective use of........ 135
  Table........ 100
  Vector........ 118
Compiling demonstration programs........ 4
Complementing lines........ 55,128
Continuous degrees of freedom........ 92
Control value (DISPLAY_INIT)........ 7
ControlWord variable........ 7
Conversion between coordinate systems........ 40
CONVERT_WTODMM procedure........ 24,226
CONVERT_WTOLMM procedure........ 227
ConvertVirtualToWorld procedure........ 41
ConvertWorldToVirtual procedure........ 42
Coordinate systems, conversion between........ 40
Coordinates:
  Cartesian........ 6
  Rectangular........ 6
  Virtual........ 13
  World........ 13,226,227
CRT drawing modes........ 55,128
CRT, graphics........ 6
CrtAddr variable........ 7,81
CsizeProg program........ 165
Cube, Color........ 110
Customizing demo programs for your system 6
Cylinder, Color........ 112

d

Data-driven plotting 71
DataPoint program 6,166
Defining a viewport 13
Degrees of freedom:
  Continuous 92
  Non-separable 92
  Number of 88
  Quality of 88
  Quantizable. 93
  Separability of. 88,92
Demonstration programs. 4,6
Device selector (DISPLAY_INIT) 6
DGLPRG disc 1
Direction, label. 17,48
Display design 134
Display limits, setting. 33
DISPLAY_FINIT procedure 228
DISPLAY_INIT procedure 6,81,84,232
DISPLAY_TERM procedure. 237
Displays:
  External color 84
  Turning on and off. 39
Dithered colors 121
Dithering. 75,117,123
Dominant lines, drawing 55,128
Drawing lines 7
Drawing modes, CRT 55,128
DrawMdPrg program. 56,166
Dumping raster images 82

e

Echoes. 94,97,221
Erasing lines 55,128
External color displays 84
External plotter control 85

f

Fast drawing procedures. 56
FillGraph program 78,170
Filling, polygon 74
FillProg program 76,169
Force, pen 86
Frame buffer 115,121,129
Frame, window 37
Freedom, degrees of 88

g

Gamuts, color 137
GLOAD procedure 68
Graphics display device, selecting 6
Graphics dump 82
Graphics, interactive 87
GRAPHICS key. 39
GRAPHICS Library, using 4
Graphics memory address 69
Graphics memory size. 69
Graphics tablet. 98
GRAPHICSERROR procedure 238
GRAPHICS_INIT procedure. 9,240
GRAPHICS_TERM procedure 9,241
GRID procedure 64
Grids. 62
GSTORE procedure 68,83
GstorProg program. 171
GTEXT procedure 15,25,32,50,242

h

Halftoning 117
Hardcopy, color 137
Highlighting data curves 79
HP 98627A RGB interface. 84
HSL color model. 103,107,126
Hue 103

i

Images:
  Dumping. 82
  Storing and retrieving 68
INCLUDE files 5,7
Input device selection 89
INPUT_ESC procedure. 244
INQ_COLOR_TABLE procedure. 247
INQ_PGN_TABLE procedure. 249
INQ_WS procedure. 13,34,37,51,251
Interactive graphics 87
INT_LINE procedure 56,258
INT_MOVE procedure. 56,260
INT_POLYGON procedure 262
INT_POLYGON_DD procedure. 265
INT_POLYLINE procedure 269
IsoProg program 60,180
Isotropic scaling 10,59

j

Justifying labels 50
JustProg program 53,186

k

KEYBOARD file 89

l

LABELJUSTIFY procedure 50
Labelling a plot 43
Labelling Axes 25
Labels:
  Bold 19
  Centering 17
  Direction of 17,48
  Justifying 50
LdirProg program 49,190
LEM programs 73,76
Limits, display 33
Line drawing 7
LINE procedure 8,32
Line Styles, selecting 57
Line value 121
Lines, clipping 23
Loading the Pascal system 7
LOCATOR program 94,98,191
LOCATOR_INIT procedure 271
LOCATOR_TERM procedure 274
Locators 220
Logarithmic plotting 64
LogPlot program 66,194
LT instruction 57
Luminosity 103

m

MAKE_PIC_CURRENT procedure 275
Map, color 122
MARKER procedure 79,276
MarkrProg program 80,196
Memory address, graphics 69
Memory size, graphics 69
Models, color 102,107
Modes, drawing 55
Module Dependency Table 339
Monochromatic defaults in color table 101
MOVE procedure 8,32,277
Multi-line objects 72

n

Non-separable degrees of freedom 92

o

OUTPUT_ESC procedure . . . 39,55,82,85,278

p

Pascal system, loading 7
Pen:
  Acceleration 86
  Force 86
  Speed 85
Permanent command 4
Photographing the CRT 138
Pixel 11,75
PLineProg program 71,197
Plot labelling 43
Plotter control 85
Plotter, selecting a 81
Plotters 114
Plotting and the CRT 138
Polygon filling 74
Polygon interiors 121
POLYGON procedure 72,74,283
POLYGON_DEV_DEP procedure 74,286
Polygons 132
POLYLINE procedure 71,290
PolyProg program 73,198

q

Quantizable degrees of freedom 93

r

Raster images, dumping . . . 82
Ratio, aspect . . . 34,60,98
Rectangular coordinates . . . 6
References, color . . . 139
Resolution of color models . . . 126
Retrieving and storing images . . . 68
RGB color model . . . 102,107,126
RGB interface . . . 84
Rotation, label . . . 17,48
Rubber echoes . . . 97
Running demonstration programs . . . 4

s

SAMPLE_LOCATOR procedure . . . 292
Saturation . . . 103
Scaling . . . 9
Scaling,isotropic . . . 59
Screen dump . . . 82
Selecting the graphics display device . . . 6
Separable degrees of freedom . . . 88,92
SET_ASPECT procedure . . . 11,14,33,294
SET_CHAR_SIZE procedure 16,44,46,50,296
SET_COLOR procedure . . . 99,114,297
SET_COLOR_MODEL procedure . . . 101,300
SET_COLOR_TABLE procedure . . . 55,101,114,302
SET_DISPLAY_LIM procedure . . . 24,33,306
SET_ECHO procedure . . . 98
SET_ECHO_POS procedure . . . 309
SET_LINE_STYLE procedure . . . 57,311
SET_LINE_WIDTH procedure . . . 319
SET_LOCATOR_LIM procedure . . . 98,315
SET_PGN_COLOR procedure . . . 99,114,320
SET_PGN_LS procedure . . . 323
SET_PGN_STYLE procedure . . . 75,327
SET_PGN_TABLE procedure . . . 328
SET_TEXT_ROT procedure . . . 17,48,331
SET_TIMING procedure . . . 332
SET_VIEWPORT procedure . . . 14,34,37,334
SET_WINDOW procedure . . . 9,14,34,35,47,59,336
Shading graphs . . . 78
SinAspect program . . . 12,199
SinAxes1 program . . . 22,200
SinAxes2 program . . . 26,204
SinClip program . . . 24,209
SinLabel1 program . . . 15,213
SinLabel2 program . . . 16,18,214
SinLabel3 program . . . 19,215
SinLine program . . . 8,216
SinViewpt program . . . 216
SinWindow program . . . 10,217
Solution vector . . . 118
Speed, pen . . . 86
Storing and retrieving images . . . 68
STRLEN procedure . . . 17
STRWRITE procedure . . . 25
System Library . . . 4

t

Target vector . . . 118
Test program . . . 89,94
Text, writing to the graphics screen . . . 15,132
Tick marks . . . 20,62

v

Vector, color . . . 118
Viewport, defining . . . 13,34
Virtual coordinates . . . 13
Vision, color . . . 133

w

What command . . . 4
Window frame, drawing . . . 37
Window limits, calculating . . . 35
World coordinates . . . 13,226,227
WRITELN procedure . . . 25
Writing modes . . . 127
Writing text to the graphics screen . . . 15

Part No. 98615-90035
E 1184
Microfiche No. 98615-99035

Printed in U.S.A.
First Edition, with update
November 1984