GINO

user guide

version 6.0

BRADLY ASSOCIATES LTD
Manhattan House
140 High Street
Crowthorne
Berkshire RG45 7AY
England
Tel: +44 (1344) 779381
Fax: +44 (1344) 773168
support@gino-graphics.com
www.gino-graphics.com

Information in this manual is subject to change without notice.

While Bradly Associates Ltd. makes every endeavour to ensure the accuracy of this document, it does not accept liability for any errors or omissions, or for any consequences arising from the use of the program or documentation.

GINO user guide version 6.0

Contents

INTRODUCTION 23

General Description 23

Facilities 24

Initializing GINO 25

GINO States 26

Use of External Files 28

Diagnostic Facilities 29
- Output of Error and Warning Messages 29
- Error Limit 30
- Trapping of Errors and Warnings 30
- Enquiry of Errors and Warnings 30
- Routine Trace Facility 31
- Output File for Error and Tracer Messages 31
- DEBUG Utility 31

Workspaces 33
- Management of Workspace Area 33
- Allocation of Workspace Area 34

GINO Coordinate System 36

GRAPHICS DEVICES 39

Graphics Devices Introduction 39
- Device Drivers 39
- Device Class 40
- 2D and 3D Devices 42

Device Nomination 42
- Device Defaults 43
- Device Attributes 43

Device Qualification · · · · · 43
Device Output Filename · · · · · 44
Drawing Units · · · · · 44
Drawing Limits · · · · · 45
Colour Capabilities · · · · · 46

Device Initialization · · · · · 48

New Drawing · · · · · 48

Device Dependent Routines · · · · · 49
Emptying the Graphics Buffer · · · · · 49
Auxiliary Drawing Areas · · · · · 49
Batch Modifications to Display · · · · · 50
Alphanumeric Mode · · · · · 51
Window visibility · · · · · 51
Device Titles · · · · · 52

Device Release and Suspension · · · · · 52

Using Multiple Devices · · · · · 52
Mapping to the Second Device · · · · · 53
Saving and Restoring GINO State · · · · · 54
Duplicating Output · · · · · 55

IMPORTING AND EXPORTING 57

Importing and Exporting Introduction · · · · · 57

Overview · · · · · 57

Metafile Formats · · · · · 58
Summary · · · · · 61

Exporting Metafiles from GINO · · · · · 62

Metafiles into External Packages · · · · · 62

Importing Metafiles into GINO · · · · · 64
SAVDRA Metafile · · · · · 64
CGM Metafiles · · · · · 69
Image Metafiles · · · · · 73

2D DRAWING 77

2D Drawing Introduction . . . 77
- Pen . . . 77
- Axes . . . 78
- 2D Start and End Pen Position . . . 79
- 2D Naming Conventions . . . 79

Positioning . . . 80

Straight Lines . . . 80

Polylines . . . 82

Polyline Sets . . . 84
- Polyline Set Definition . . . 84
- Polyline Usage . . . 85

Circular Arcs . . . 86
- Two-Dimensional Arcs . . . 86
- Drawing Circles . . . 88
- Hardware and Software Arcs . . . 89
- Arc Control Routines . . . 89
- Arc Settings . . . 91
- Use of Arc Routines . . . 91

Parametric Curves . . . 93
- Curve End Conditions . . . 94

Spline Curves . . . 98
- Spline Curve End Conditions . . . 99
- Spline Curve Tension Control . . . 100

Bezier Curves . . . 101
- End Conditions . . . 102
- Elevation and Reduction . . . 103

Point Storage . . . 103

2D Interpolation . . . 107

LINE ATTRIBUTES 111

Line Attributes Introduction . . . 111
- Routines Described in this Chapter . . . 111
- Current Line Definition and Enquiry . . . 112
- Drawing Attribute Tables . . . 113

Individual Attributes 114
Changing Individual Attributes of the Current Line 114
Line Visibility 116
Broken Line Type 116
Line Colour 117
Line Width 119
Drawing Mode 120
Line Ends 120
Use of Current Attribute Enquiry Routines 122

Attribute Tables 124
Attribute Definition Tables 124
Broken Line Types Table 124
Continuous v Discontinuous 128
Line Definition Table 129
Changing the Current Line Attributes 130
Retrieving and Storing Current Line Attributes 131

CHARACTERS 135

Character Introduction 135

Character Mode - Hardware v Software 136

Output of Characters 137
Single ASCII Characters 137
Character Strings 138

Output of Numbers 138
Field Width 139
Conversion of Numbers to Character Strings 140

Character Fonts 141
Font styles 141
Font Fill Style 144
Font Weight 145
Fixed Pitch Control 146
Software Font Representation 146
Font Enquiry 147

Character Attributes 147
Default Character Settings 147
Character Size 147
Character Orientation 149
Italic Characters 150
Current Character Settings Enquiry 151
Underlining of Characters 152
Representation of Zero Character 152
Line Attributes affecting Characters 152

Character String Attributes 153
Justification 153
Text Blocks 154
Exponents and Indices 155
Escape Characters 156
Changing the Escape Character 158
Escape Character Enquiry 158
Character Strings Adjusted to Fit a Specified Width 158
Character Strings Drawn Along a Curve 159
Returning Information about a String 159
Country Specific Characters 160

Symbols 161
Positioning Symbols 162
Multiple Symbols 163

AREA FILLING 165

Area Filling Introduction 165

Filling a Rectangle 165

Filling Single Polygons 167

Filling Polygon Sets 169
Polygon Set Definition 169
Polygon Usage 170

Filling Modes 171

Hatch Style Definition 172
Example 1 178
Example 2 180
Hatch Style Enquiry 182

Multiple Hatch Styles · · · · · 183
Box Hatch style · · · · · 183
Brick Hatch Style · · · · · 184
Honeycomb Hatch Style · · · · · 185
Trellis Hatch Style · · · · · 187

Complex Polygonal Definition, Drawing and Filling · · · · · 188

IMAGE HANDLING 189

Image Handling Introduction · · · · · 189
Pixel Coordinate System · · · · · 190

Reading and Writing Single Pixels · · · · · 191

Image Display · · · · · 191
Image Data · · · · · 192
Sub Images · · · · · 192
Pixel Packing · · · · · 194
Image Display Mode · · · · · 197

Pixel Coordinate Conversion · · · · · 198

Pixel Transformations · · · · · 198
Pixel Rotation and Scaling · · · · · 198
Pixel Replication · · · · · 201

Pixel Enquiry Routines · · · · · 202

Reading Pixel Data · · · · · 203

Copying Pixel Images · · · · · 203

COLOUR DEFINITION 205

Colour Definition Introduction · · · · · 205

Colour Table · · · · · 205
Display Types · · · · · 206
Colour Resolution · · · · · 207

Colour Coordinate Systems · · · · · 207
Conversion Between Coordinate Systems · · · · · 210

RGB Colour Coordinate System · · · · · 211
Using the RGB System · · · · · 212

HSV Colour Coordinate System · · · · · 212
Using the HSV System · · · · · 214

HLS Colour Coordinate System · · · · · 215
Using the HLS System · · · · · 216

Direct Colour Control · · · · · 217

MAPPING, WINDOWING AND MASKING 219

Mapping, Windowing and Masking Introduction · · · · · 219

Viewport Mapping · · · · · 219
Viewport Enquiry · · · · · 221
Clearing the Viewport · · · · · 221

Clipping · · · · · 222
Window Mode · · · · · 222
Rectangular Window · · · · · 223
Enquiring Window Limits · · · · · 224

Rectangular Masks · · · · · 224
Mask Enquiry · · · · · 225

2D TRANSFORMATIONS 227

2D Transformations Introduction · · · · · 227

Simple 2D Transformations · · · · · 227
2D Shifting · · · · · 227
2D Rotation · · · · · 228
2D Scaling · · · · · 229
Mirror Images · · · · · 230
2D Shearing · · · · · 231

Combining Transformations · · · · · 232
Using the Same Transformation Type · · · · · 232
Using Different Transformation Types · · · · · 233

2D Transformation Enquiry 238
Current Drawing Position 238
2D Untransforming 238
Point Testing of Current 2D Transformation 238

2D Transformation Control 239

Transforming Characters and Symbols 239

BASIC INTERACTION 241

Basic Interaction Introduction 241

Cursor Input 241

Defining Cursor Shapes 242

Defining Cursor Action Types 243

Application 243

ADVANCED USE OF 2D POLYGONS 245

Advanced Use of 2D Polygons Introduction 245

Allocating Workspace for the Storage of Polygons 245

Polygon Definition 247
Polygon Identity 249

Clearing Polygon Workspace 250

Status of Polygon Workspace 250

Drawing Polygon Boundaries 251

Polygon Filling Workspace Requirements 252
Hardware Fill Workspace Requirements 252
Software Fill Workspace Requirements 253
Example Calculations of Workspace Requirements 253

Polygon Selection 254
Polygon Selection Enquiry 257

Filling a Polygon · · · · · 257

Interaction with Polygons · · · · · 261

Polygon Windowing and Masking · · · · · 261
Polygons Suitable for Windowing and Masking · · · · · 261
Workspace Requirements for Windowing and Masking of Filled Areas · · · · · 262
Windowing Requirements · · · · · 262
Masking Requirements · · · · · 262
Requirements for Simultaneous Windowing and Masking · · · · · 263
Example - Calculation of Fill Workspace Requirements · · · · · 263
Polygonal Windowing · · · · · 264
Polygonal Masking · · · · · 265
Windowing and Masking Polygon List Enquiry · · · · · 266
Windowing and Masking Control · · · · · 267

3D GRAPHICS 269

3D Graphics Introduction · · · · · 269
Shaded Objects · · · · · 270
The Scene · · · · · 270
3D Device Drivers · · · · · 271

The 3D World · · · · · 272

3D Viewport Mapping · · · · · 273

3D Clipping · · · · · 275
Enquiring 3D Window Limits · · · · · 275

3D DRAWING 277

3D Drawing Introduction · · · · · 277
3D Axes · · · · · 278
3D Start and End Pen Position · · · · · 278
3D Naming Conventions · · · · · 278

3D Positioning · · · · · 279

3D Straight Lines · · · · · 280

3D Polylines · · · · · 280
Shaded Polylines · · · · · 281

3D Polyline Sets 282
3D Polyline Set Definition 282

3D Arcs 284
Direction Vector 288

3D Spline Curves 288
3D Spline Curve Control 289

3D Bezier Curves 290
3D Elevation and Reduction 290

3D Polygons 290
Overlapping Polygons 291

3D Point Storage 293

3D Interpolation 294

FACETS 295

Facets Introduction 295

Facet Definition 296
Facet Faces 296
Normals 297
Textured Facet 299
Coloured Facet 299

Facet Attributes 301
Facet Fill Style 301
Facet Offset 302

3D OBJECTS 305

3D Objects Introduction 305
Local Axes System 306
Object Complexity 306
Object Shading 306
Object Texture Mapping 306

Shaded Polyline 307

3D Primitives 307
Boxes 307
Wedges 309
Cylinders and Cones 309
Spheres 310
Volumes of Rotation 311

Surface Primitives 312
Spline surface 313
Bezier surface 317
Tabulated Bezier surface 320
Swept Bezier surface 320
Ruled Bezier surface 321
Bezier sphere 322
Bezier volume 323

LIGHTING AND SHADING 325

Lighting and Shading Introduction 325
Shading 325
Culling 326
Blending 326
Winding Rule 327
Shading Enquiry 327

Depth Buffering 327

Lighting 329
Light Sources 329
Light Switch 332
Default Lights 332
Light Source Enquiry 333
Light Usage 333

Fog 337
Fog Enquiry 338

MATERIAL PROPERTIES 339

Material Properties Introduction 339

Material Property Definition 339
Colour Matching 340
Material Table 340
Facet Material Properties 342

Translucence · · · · · 342

Shadows · · · · · 342

TEXTURE MAPPING 345

Texture Mapping Introduction · · · · · 345

Texture Mapping Modes · · · · · 345

Texture Mapping Data · · · · · 346
Multiple Texture Maps · · · · · 348
Tiling Images · · · · · 349

Texture Mapping Coordinates · · · · · 349
Direct Assignment · · · · · 350
Automatic Generation · · · · · 351
Environment Mapping · · · · · 353
3D Objects · · · · · 353

Texture Mapping Attributes · · · · · 354
Blending Textures · · · · · 354
Repeating and Clamping Images · · · · · 354
Filtering Textures · · · · · 355
Texture Border Colour · · · · · 357

Texture Mapping Enquiry · · · · · 358

3D TRANSFORMATIONS 359

3D Transformations Introduction · · · · · 359
Current Transformation · · · · · 360

Simple 3D Transformations · · · · · 360
3D Shifting · · · · · 360
3D Rotation · · · · · 360
Permutating the Axes · · · · · 362
3D Scaling · · · · · 363
3D Shearing · · · · · 364

Combining 3D Transformations · · · · · 364
Using the Same 3D Transformation Type · · · · · 364
Combining 3-D Rotations · · · · · 365
Using Different 3D Transformation Types · · · · · 367

3D Transformation Enquiry 368
Finding the Current Drawing Position 368
3D Untransforming 368
Point Testing of Current 3D Transformation 369

TRANSFORMATION CONTROL 371

Transformation Control Introduction 371

View Transform Mode 371

Transformation State 372
Reinitializing 372

Transformations Matrix Control 375
Push and Pop Transformation Matrix 376
Saving and Restoring Transformation Matrix 376
Getting and Setting Transformation Matrix 377
Modify Transformation Matrix 377

Transformation Matrix Building 378
Example showing Building and Combining Transformation matrices 379

Transformation Enquiry 381

Transformation Mode 382

VIEWING 385

Viewing Introduction 385

Useful Concepts 386

From View Plane to Paper 387

The Basic Viewing Routines 388

Perspective Views of a Volume 388

Perspective View from a Point 393

Parallel Projection 396

Setting Viewing Transformations 398
Use of Superseded Routine 399

Modifying the Drawing 400
Re-specifying the View 400
Positioning the Image 402
Orientation of the Image 402

Moving Eye, View Plane or both 404
Zooming 404
Moving Eye and View Plane 405
Moving the Eye Alone 407

Changing the Line of Sight 410

Projections onto an Oblique Plane 415

Saving and Restoring View Parameters 416

Modifying the View Matrix 417

Listings of the Routines used in this Chapter 417

PICTURE SEGMENTS 423

Picture Segments Introduction 423
Software Display File Storage 426

Segment Building 426
Segment Anchor 427
Picture Segment Body 430

Segment Manipulation 430
Picture Segment Transformations 431

Segment Enquiry 432

Segment Redrawing and Repairing 433

Segment Structures 434
Copying 434
Hierarchical Segment Structures 434
Use of Modelling Transformations within Segments 435
Segment Groups 437
Implicit Segment Groups 439

Light Pen Simulation · · · · · 439
Dragging · · · · · 440
Software Display Files Across Devices · · · · · 440
Archiving and Restoring Software Display File · · · · · 442

ADVANCED INTERACTION 447

Advanced Interaction Introduction · · · · · 447
Programming in a windowing environment · · · · · 449
Event Types · · · · · 449
Requesting Event Types · · · · · 450
Deleting Event Types · · · · · 450
Getting Next Event · · · · · 451
Reading Event Data · · · · · 451
Keys · · · · · 452
Event Generating Implements · · · · · 454
Event Programming · · · · · 454
Queues · · · · · 456
Mouse Position · · · · · 456
Keyboard State · · · · · 457

SYSTEM UTILITIES 459

System Utilities Introduction · · · · · 459
File and Directory Handling · · · · · 460
Time and Date Utilities · · · · · 463

Other System Utilities 463
Command-line arguments 463
Enquire User Name 464
Environment Variable Settings 464
System Command Execution 465
Task Priority 466
Sound System Speaker 466
Random Number Generation 466
String Handling 467

ROUTINE SPECIFICATIONS 469

MACHINE IMPLEMENTATIONS 735

GENERAL 735

UNIX 736

OpenVMS 737

Microsoft Windows 738

DEVICE DRIVERS 739

Device Drivers Introduction 739

Configuration File 741

Dummy Device 741

SCREENS AND WORKSTATIONS 742
Output Filenames and Unit Numbers (Fortran only) 742
Screen Driver Configuration Settings 743

GLX OpenGL Extension to X 743

Regis Series Devices 752

VGA and SVGA PC Screens (LF90 only) 755

Windows (Microsoft) System 759
Using Windows Driver in Windows Programming Environment 769

Windows OpenGL (Microsoft) System · · · · · 778
Using Windows OpenGL Driver in Windows Programming Environment · · · · · 788

X Windows System · · · · · 790

PRINTERS AND PLOTTERS · · · · · 797
Output Filename and Unit Numbers (Fortran only) · · · · · 797
Printer and Plotter Configuration Settings · · · · · 798
Intermediate Vector File · · · · · 799
8-bit data · · · · · 799

Calcomp 907 Series Plotters · · · · · 799

Hewlett-Packard Series Plotters (HPGL) · · · · · 802

Hewlett-Packard Series Plotters (HPGL-2) · · · · · 806

Hewlett-Packard Laserjet Series Printers (HPLJ) · · · · · 809

Hewlett-Packard Paintjet and Deskjet Printers (HPPJ) · · · · · 810

DEC LA100 andLN03 Series Printers · · · · · 812

Postscript Series Printers · · · · · 813

METAFILES · · · · · 818
Output Filename and Unit Numbers for Metafiles(Fortran only) · · · · · 818
File Format · · · · · 819
Metafile Configuration Settings · · · · · 819

Computer Graphics Metafile (CGM) · · · · · 819

Drawing Exchange Format (DXF) Metafile · · · · · 824

Image File Formats (BMP, XWD, SUNRAS) · · · · · 825

JPEG File Interchange Format (JPG) · · · · · 827

PNG Portable Network Graphics (PNG) · · · · · 829

SAVDRA and SAVPIC Metafile · · · · · 830

Windows Metafile (WMF) · · · · · 833

FONT TABLES 835

Font Tables Introduction 835

The Font Tables 836

DEFAULTS 859

Defaults Introduction 859

ERROR AND WARNING MESSAGES 867

Error and Warning Introduction 867

GINO Errors and Warnings 867

CGM Errors 881

System Input and Output Errors 888

Configuration File Errors 889

GINO STRUCTURES 891

Structures Introduction 891

CROSS REFERENCES 901

Cross References Introduction 901

F77-F90 Cross-Reference 901

F90-F77 Cross-Reference 913

DEPRECATED ROUTINES 927

Deprecated Routines Introduction 927

TECHNICAL INFORMATION 931

Homogeneous Coordinate Transformations 931

2-D Transformations 931
Null transformation 931
Shifting 932
Rotating 932
Permutating 932
Scaling 932
Shearing 932

2-D Matrices 932

3-D Homogeneous Transformations 933

Combining Multiple Transformations 934

2-D Summary 935

Extending 2-D Operations 935

Perspective Transformations 937

Chapter 1

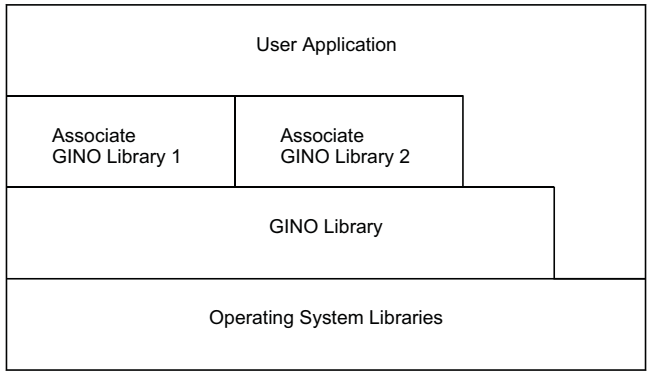
INTRODUCTION

General Description

GINO stands for Graphical INput/Output. It is a graphics package that takes the form of a library of 2D and 3D drawing and administrative routines and is designed to offer implementation and device independent graphics facilities on a wide range of machine platforms. GINO provides a common graphics interface to all the widely available graphics hardware through its many device drivers, to such an extent that in many cases changing one line of a GINO application is all that is needed to enable the application to operate on a different graphics device.

GINO provides a base level of graphics functionality and associated with GINO are a number of additional graphics libraries with specialist functionality in the areas of graph drawing (GINOGRAF), surface and contour drawing (GINOSURF) and the design of graphical user interfaces (GINOMENU). This document details the use of the facilities available within GINO, and a full specification of each routine available.

The association between the user application, GINO and its associated libraries is shown in the diagram below:



User Application and GINO Libraries

GINO has been written to be able to control a wide range of graphics devices from a single off-line pen plotter to a sophisticated 24 bit, 3D multi-window networked display through a common routine interface.

This enables large and complex graphical applications to be written in a device independent manner, so that the only step required when switching to another graphics device (possibly in a completely different environment) is to notify GINO of the new device name.

Facilities

GINO provides facilities for producing graphical output that can range from two-dimensional graphs to complex three-dimensional interactive systems with shaded models. Facilities are provided for:

- Nomination and device specific options
- Basic drawing (2D and 3D) both to position and to draw straight lines, circular arcs, curve drawing, and multiple drawing routines
- Character output: software fonts, hardware font access, 1000+ symbols, attribute control (angle, justification, italic, fill style, underline, weight, etc)
- Attribute control: colour definition models, visibility, line and fill styles
- Area fill with default or user-defined hatch styles, and solid fill styles
- Transforming and viewing objects, polygonal windowing and masking
- Picture segments: hierarchical structure and manipulation
- Cursor or mouse interaction and event handling
- CGM and proprietary metafiles
- Pixel rectangle read, write, and transformation control
- Surface Primitives and 3D objects
- Lighting, Shading and Texture mapping

Initializing GINO

The first call in a graphics application using the GINO library should always be to initialize the library. This is performed by the routine:

gOpenGino()

This may occur anywhere in a user program but should precede all calls to other GINO routines. If the call to gOpenGino() is omitted, what happens depends largely on the computer system running the program but in most implementations, the initialization will occur when the first GINO routine is called, however GINO error 5 may be output, ‘GINO not initialized’.

The routine gOpenGino() should not be called after any other GINO routines with the exception of gCloseGino(). When gOpenGino() is called, GINO is re-initialized and all data previously defined is discarded.

When GINO is initialized, one of the first steps carried out by the library is to check for the existence of a legal Configuration File and if this file does not exist or does not have the correct licence information encoded within it, GINO will immediately stop with an appropriate error message. In order to provided a cleaner check on the existence of the Configuration file, an alternative initialization routine is provided, returning a status flag through its single argument. The routine is gEnqConfigStatus():

status=gEnqConfigStatus([cfgdir])

where **status** returns a value of 0 if a legal Configuration File has been located and 1 otherwise. This can be useful in providing a user controlled abort mechanism if the Configuration File does not exist. The optional argument **cfgdir** can be used to set the location of the GINO configuration file if it is known to be in a non-standard location. The function gEnqConfigStatus() must only be used in place of a call to gOpenGino as it will re-initialize GINO if called anywhere else.

Closing Down GINO

At the end of an application, GINO should be properly closed down through the routine gCloseGino(). This contains an implicit call to gCloseDevice() to close down the currently opened device.

gCloseGino()

The routines gOpenGino() and gCloseGino() should be used to open and close the GINO part of a user program.

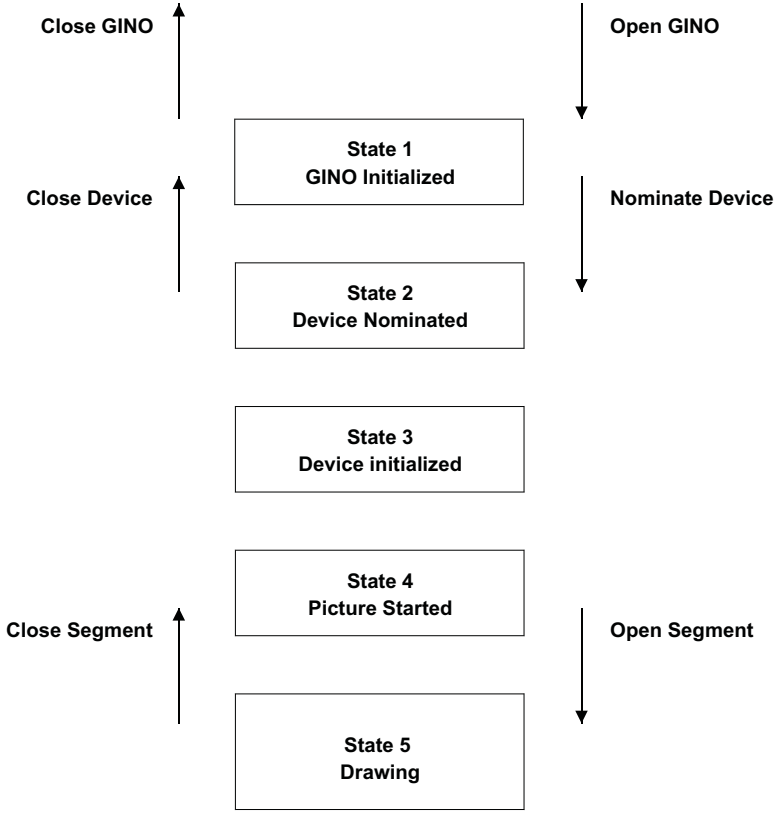
GINO States

When GINO is first initialized, it enters a state where a small number of operations can take place. These include setting error and tracer operation, initializing gDebug() and nominating a graphics device. GINO can exist in fact in one of five states as shown in the diagram below. An application can enquire which state GINO is in through the routine gEnqGinoState():

gEnqGinoState(gstate)

where **gstate** returns the state of GINO and all its associated libraries. The structure member .gino contains the GINO state, in the range 1 to 5, and other members of the structure indicate whether associated libraries are active or not.

GINO will automatically move from one state to another through the calling of certain routines, (eg. gCloseSeg() will move GINO from state 5 to state 4), while some routines are only permitted while GINO is in a particular state (eg. Device Qualifying routines are only permitted while GINO is in state 2). In some cases GINO may move through a number of states by a single GINO routine (eg. a call to gDrawLineTo2D() after device nomination will move GINO from state 2 to 5, and a call to gCloseDevice() will move GINO to state 1 which ever state it was in at the time).



GINO States

Use of External Files

Several GINO routines need to reference external files in order to either direct graphics commands to a file, reading or writing metafiles or directing diagnostics information. For historical reasons many of these routines require an identifier of an already opened file.

Two routines are provided to open and close such files in a language independent way:

file=gFopen(name,mode)

gFclose()

In the C interface, the function gFopen() returns a pointer to a special file structure called ‘GFILE’, which is required by all the file handling routines in GINO. The routine has the same arguments as the standard C routine fopen. The file can be closed with the routine gFclose(). If a NULL pointer is used, standard input or standard output is used as appropriate to the GINO routine.

When using the Fortran 90 interface, the function gFopen() returns an integer file unit number of a file opened by the GINO library. This prevents the possible problem of a file being opened by a GINO application (using the OPEN statement) being unknown to the GINO library which can occur if the GINO library is being used as a DLL.

The following example shows how to direct error/tracer output and PostScript output to two named files within a GINO program:

C code

```
#include <gino-c.h>
GFILE *erfile
main()
{
   gOpenGino();
/* open file for error and tracer output */
   erfile = gFopen("error.out","w");
   gSetErrorFile(erfile);
/* nominate postscript driver */
   gEps(1,0.0,0.0,297.0,210.0,297.0,210.0);
/* direct formatted output to file */
   gSetDeviceFilename("triangle.eps",-1);
/* draw triangle */
   triangle(20.0,20.0,25.0);
/* close device and file */
   gCloseDevice();
   gFclose(erfile);
}
```

F90 Code

```
use gino_f90
integer erunit
   call gOpenGino
! open file for error and tracer output
   erunit=gFopen('error.out',GWRITE)
   call gSetErrorFile(erunit)
! nominate postscript driver
   call gEps(1,0.0,0.0,297.0,210.0,297.0,210.0)
! direct formatted output to file
   call gSetDeviceFilename('triangle.eps',-1)
! draw triangle
   call triangle(20.0,20.0,25.0)
! close device and file
   call gCloseDevice
   call gFclose(erunit)
   stop
   end
```

Diagnostic Facilities

Output of Error and Warning Messages

GINO generates errors and warnings when it detects that something is wrong. A fault is classed as an error if GINO is unable to perform some task or has to take drastic action to remedy the situation, e.g. GINO error 1 - an attempt to output something when no device is nominated, so GINO nominates DUMMY as the default device-driver.

Warnings are displayed after less serious faults. GINO is able to carry on by assuming a straightforward default and the warning is generated simply to inform the user that this has happened, e.g.

```
GINO warning 46 - Negative colour index
Detected in call to gSetLineColour (LINCOL)
```

is generated when the colour index is set less than zero in a call to gSetLineColour(), so GINO uses the absolute value. The message will specify the GINO library within which the error was detected, the error/warning number, some text to describe the fault, and an indication of the GINO routine that detected the fault. The routine name listed is the F90 long name with the F77 short name in brackets. A complete list of GINO error and warning messages is given in Appendix E.

While error and warning messages are useful to indicate a possible problem in an application, there may be reasons to switch off warnings and/or error messages throughout, or at a particular place within an application. This can be achieved through the following routine:

gSetErrorMode(switch)

where its argument sets the desired output state of either GON (the default) or GOFF.

Error Limit

GINO keeps a count of errors and keeps a log of errors and warnings. When this count exceeds a specified limit, GINO stops the program. The limit is set to 10 and the count is set to zero when GINO is initialized. A call to the following routine allows the user to change the limit:

gSetMaxErrorLimit(n)

The limit can be disabled by setting **n** to -1. The count is reset to zero each time gSetMaxErrorLimit() is called.

Trapping of Errors and Warnings

There is a straightforward way to trap the occurrence of any errors or warnings. A call to gSetErrorTrap() enables or disables the trapping mechanism:

gSetErrorTrap(switch)

gEnqNumberOfErrors() may then be called to return the number of errors and warnings counted since enabling error trapping:

gEnqNumberOfErrors(n)

gEnqNumberOfErrors() returns **n** = -1, if trapping is disabled. More information about the errors and warnings that were trapped can be obtained by calling gEnqLastErrors() (see below).

Enquiry of Errors and Warnings

GINO can store up to 12 error and warning numbers. If more than 12 have been generated, GINO discards the oldest numbers. A call to gEnqLastErrors() will return up to **n** numbers along with the total **count** of errors only:

gEnqLastErrors(list, n, count)

The numbers are returned in integer array **list**, with the first element containing the most recent one. Error numbers are positive and warning numbers are negative. Any element of **list** that does not return a valid number is set to zero.

Routine Trace Facility

The routine trace facility outputs a message to identify each call made to a GINO routine. It is switched on by calling the routine:

gSetTracerMode(switch)

with a non-zero argument. The argument allows different trace reporting, listing the routine names of GINO and GINO's application packages . The routine trace facility is very useful for determining the exact sequence of calls to GINO routines, or to any routines in the GINO application packages.

Output File for Error and Tracer Messages

By default, all error and trace messages are output on the systems default standard output unit or standard error output unit for UNIX installations. Users can direct error and tracer messages to a different external file using the routine:

gSetErrorFile(file)

where **file** is a pointer to a file opened through the gFopen() routine or a Fortran 90 file unit number.

DEBUG Utility

GINO provides a debug facility designed to assist the user in tracing bugs in programs. It does not replace the actual device driver (MWIN, EPS, etc.) but sits between the front-end and the device driver keeping track of the graphical input/output generated by the user program. Nor does gDebug() affect the user program in any way, it simply mirrors the calls to GINO routines in the user's program and outputs these to an external file.

gDebug(file, level)

where **file** is a pointer to a file opened through the gFopen() routine or a Fortran file unit number and **level** controls the amount of information to output, i.e. the level of trace.

Debug output may be switched on and off during a program execution in order to generate output at the desired section of code using:

gSetDebugSwitch(switch)

where **switch** = GOFF to switch Debug output off and GON (the default state) to switch Debug output on.

The routine gDebug() must be called just before any device driver nomination routine since a call to gCloseDevice() terminates the action of gDebug(). Note that gDebug() itself also makes an implicit call to gCloseDevice().

This is demonstrated in the following example program:

C code

```
#include <gino-c.h>
int main()
{
  GFILE *file;
  GLIMIT rect={10.0.20.0,10.0,20.0};

  file=gFopen("debug","w");
  gOpenGino();
  gDebug(file,GEXTRA);
  gMwin();
  gMoveTo2D(0.0,0.0);
  gDrawLineTo2D(50.0,50.0);
  gSetLineColour(GRED);
  gSetFillMode(GSOFT);
  gFillRect(GCURRENT,GSOLID,&rect);
  gCloseGino();
  gFclose(file);
}
```

F90 Code

```
program debug
use gino_f90
integer :: file
type (GLIMIT) :: rect = GLIMIT(10.0,20.0,10.0,20.0)

   file=gFopen('debug',GWRITE)
   call gOpenGino
   call gDebug(file,GEXTRA)
   call gMwin
   call gMoveTo2D(0.0,0.0)
   call gDrawLineTo2D(50.0,50.0)
   call gSetLineColour(GRED)
   call gSetFillMode(GSOFT)
   call gFillRect(GCURRENT,GSOLID,rect)
   call gCloseGino
   call gFclose(file)
stop
end
```

Note that the output produced from the DEBUG utility lists routine names as per the F77 short-name convention. Use the F77-F90 cross reference table in Appendix G for converting to the appropriate long names.

Workspaces

At various times GINO needs to store information in memory. Normally these are of a known size and can therefore be declared internally as arrays.

However, there are facilities in GINO which possibly require large amounts of memory depending on the complexity of the users application. In such cases it is not sensible for GINO to pre-assign space as it is impossible to predict the requirement. These areas include polygon storage, area filling, internal point storage and the software display file.

GINO therefore provides a mechanism for the user to allocate a single block of memory for such purposes of a size defined by the user as required by the application. In fact such an area MUST be allocated by the user if these facilities are being used by an application program. GINO then provides a handler to manage this area, allocating a smaller amount of memory as and when required.

However, it should be noted that the mechanism for allocating this workspace differs in each of the C/C++ and Fortran 90 versions of GINO as described below.

Management of Workspace Area

While the same routine gSetWorkspaceLimit() is used to allocate the size of the workspace area in both the C/C++ and Fortran 90 versions of the GINO library, the location of the storage area and the number of arguments to this routine differ.

gSetWorkspaceLimit(n1[,n2])

In the C/C++ library the workspace area is allocated via the standard library routine malloc when gSetWorkspaceLimit() is called. The routine therefore only requires a single argument giving the total memory requirement for the workspace area:

```
gSetWorkspaceLimit(n1);
```

where **n1** is the required size of the total workspace area.

In the Fortran 90 library the workspace area is allocated in an allocatable real array. For historical reasons, two arguments are required, but only the second is used:

```
call gSetWorkspaceLimit(n1, n2)
```

where **n1** is ignored and **n2** is the number of real words required in the workspace area.

It is important to free (deallocate) this memory at the end of an application but this is automatically achieved through the GINO routine gCloseGino().

Allocation of Workspace Area

For example:

C code

```
#include <gino-c.h>
main ()
{
/*  Initialize GINO */
     gOpenGino();
     .
     .
/*  Define workspace area */
     gSetWorkspaceLimit(6000);
     .
     .
/*  Free allocated workspace */
     gCloseGino();
}
```

F90 code

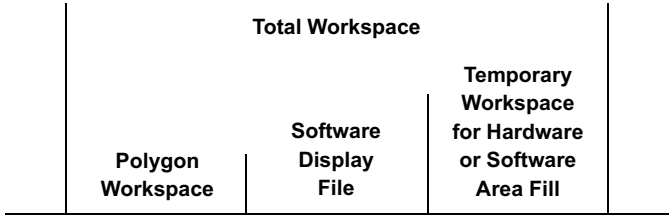
```
program work
use gino_f90

!  Initialize GINO
     call gOpenGino
     .
     .
!  Define workspace area
     call gSetWorkspaceLimit(1,6000)
     .
     .
!  Free allocated workspace
     call gCloseGino
stop
```

would define a workspace area of 6000 words.

As only a single block of memory can be allocated for the workspace area, its total estimated size must be calculated before allocating the space. This can be achieved through studying the following sections.

The space within gSetWorkspaceLimit() is used for a number of different purposes and is allocated in continuous blocks as required and for as long as required. Three routines are provided to the user for specific allocation of space for polygon storage (gDefinePolygonWorkspace()), for internal point storage (gDefinePointWorkspace()) and for the software display file (gDefineSegWorkspace()) if this is required in memory. Other areas are allocated and returned internally by GINO for temporary workspace for area filling.

The workspace handler does its addressing and word counting in real words. The workspace area must be large enough for all the concurrent workspace needs of a given program. For example, an application where polygons will be stored, a software display file used and area filled, might have a workspace area which would appear as shown below.



Example Organisation of the Workspace Area

The total size of the workspace area depends upon the individual workspace needs found by consulting the sections given in the table below.

Space	Reference Section
Polygon Workspace	ADVANCED USE OF POLYGONS
Polygon List	ADVANCED USE OF POLYGONS
Polygonal Area Filling	ADVANCED USE OF POLYGONS
Software/Hardware Area Fill	ADVANCED USE OF POLYGONS
Polygonal Windowing / Masking	ADVANCED USE OF POLYGONS
Segment Workspace	SEGMENTS

Areas allocated within gSetWorkspaceLimit() may be returned to the workspace by calling the appropriate routine gDefinePolygonWorkspace() or gDefineSegWorkspace() with an argument of 0.

The workspace area may be enlarged during an application program by calling gSetWorkspaceLimit() with a larger size. (i.e. no information is ever moved within the workspace). The workspace area cannot be reduced except by freeing the total area (in which case all information stored within it is lost) and allocating a smaller area.

Any workspace that has been allocated will be freed when GINO is closed through gCloseGino().

Enquiring about the Workspace Area

The routine gEnqWorkspaceLimit() may be called to enquire about the workspace area:

gEnqWorkspaceLimit(n [,n2])

Where **n** (and **n2** in Fortran 90) returns the size of the workspace area as defined by the last call to gSetWorkspaceLimit(). If gSetWorkspaceLimit() has not been called, this routine returns zero in its arguments(s).

GINO Coordinate System

When a graphics device is first nominated, GINO defines a default paper coordinate system which consists of the default drawing area measured in millimetres with its origin at the bottom left corner of the drawing area. At this point, the user may alter the paper (or drawing) units and drawing limits before drawing commences (see page 39). The transformation between paper coordinates and device coordinates is defined within the current device driver and cannot be changed by the GINO programmer.

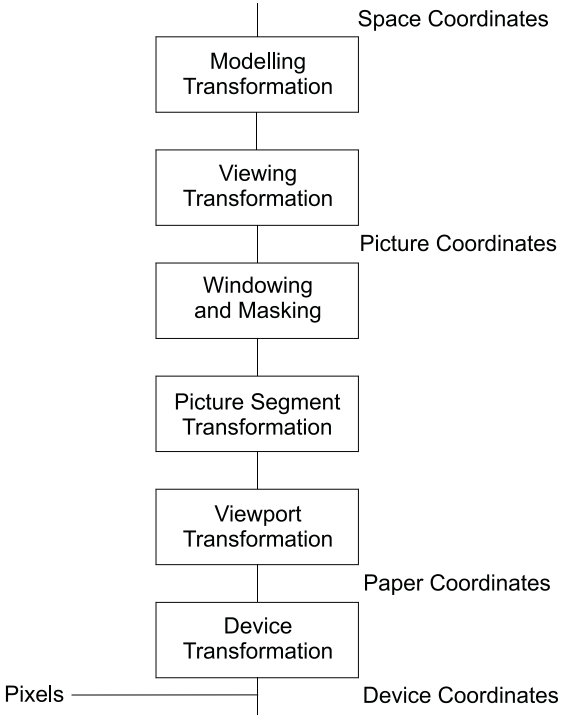
Once a device has been initialized, the user may define a viewport, which will set up a mapping between picture coordinates and the devices' paper coordinates (see page 219). All drawing operations will then operate in picture coordinates. The mapping may define any range of picture coordinates which are to be displayed over a portion of the drawing area defined in paper coordinates. All windows and masks are defined in picture coordinates. The viewport mapping may be redefined at any point within an application. At this point the user is still operating in a 2D coordinate system (although a notional Z coordinate can exist, but is effectively ignored) but the origin may exist on or off the drawing area.

In order to operate in a fully 3D coordinate system, a modelling and/or viewing transformation may be defined, setting up a new space (or world) coordinate system (see page 359). All output primitive coordinates are then mapped to picture coordinates (2D) through the current modelling and viewing transformation matrices and then clipped to the current window, mask or viewport boundary before being transformed again according to the current viewport mapping.

Picture segment transformations, where appropriate take place between clipping and viewport transformation (see page 423).

The pixel input and output routines (see page 189) operate in device coordinates and are therefore completely independent of the GINO pipeline.

The figure below shows the GINO pipeline in a diagrammatic form.



GINO Pipeline

Chapter 2

GRAPHICS DEVICES

Graphics Devices Introduction

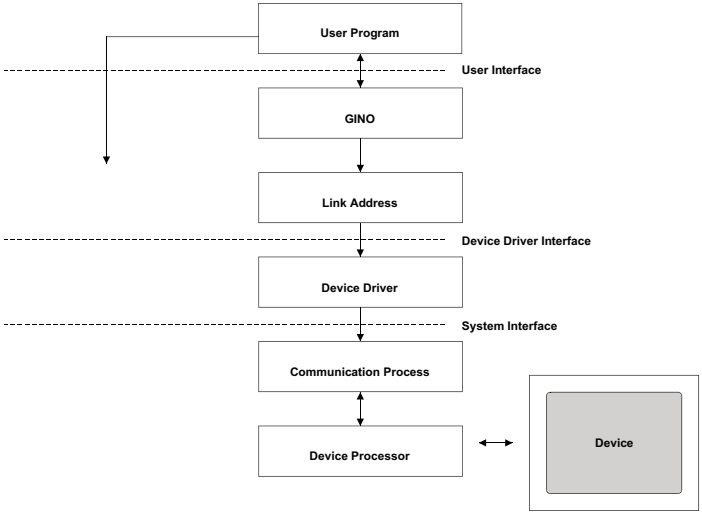
Before any input or output to a graphics device can take place the required graphics device must be initialized. This chapter describes the means by which this step is achieved and the various states in which a device can exist.

Device Drivers

GINO controls each graphics device by means of a common software interface between the GINO library and a piece of software called a Device Driver. Information is passed to and from the driver, which in turn carries out all requested operations that it is able to achieve. Operations that the driver is unable to perform are, in the majority of cases, carried out by the GINO library itself through some form of emulation using a lower level graphics function. For example, if a plotter driver is unable to hatch fill a complex polygon, GINO will generate the required hatch lines and ask the driver to simply draw the lines in 2D. Equally if a 2D screen driver is asked to draw an object in 3D, GINO will perform the transformations and pass the appropriate 2D coordinates to the driver.

Each device driver is written as a separate piece of code which is either supplied as a separate library from the GINO library itself or may be included within the GINO library. They are also generally written to drive a number of devices within a particular family of hardware devices (eg HPGL or POSTSCRIPT).

While a large range of devices can be controlled with GINO, only one device can be addressed at any one time and may only do so once the selected device is nominated. The operation of calling the device nomination routine establishes a link between the GINO library and the required device driver through a routine address plugging mechanism as shown below.



Device Driver Interface

Device Class

There are four classes of output device that GINO is able to drive, each offering a different level of sophistication and range of facilities:

Notional

One notional device driver is provided in the GINO library.

The DUMMY device driver provides the facility to run a GINO program without generating any graphical output. It is useful for testing programs, for example, when a graphics device is not available or when the non-graphical part of a program is being developed.

Screens/Workstations

Screen and workstation device drivers provide the most sophisticated level of facilities, all of which have input and output capabilities. Such devices include:

- OpenGL
- Windows API
- X-Windows

Plotters/Printers

Printer/Plotter device drivers only provide for output facilities which can be directed to an on-line device or to a file for spooling later. Such device families include:

- HPGL
- HPGL/2
- HPLJ
- HPPJ
- PostScript

Metafiles

Metafile device drivers also provide output only facilities but are stored in a device independent format. Metafiles are further sub-divided into vector and bitmap classes. Output from metafile formats can be read into many Word Processing, DTP and Image Processing packages and many can also be read back into GINO (see page). Metafile formats that can be produced by GINO include:

- BMP (Bitmap)
- CGM (Vector)
- DXF (Vector)
- JPEG (Bitmap)
- PNG (Vector)
- SAVDRA (Vector) [GINO proprietary]
- WMF (Vector)

2D and 3D Devices

GINO device drivers are further categorized into either 2D or 3D, depending on their capability to be able to handle 3D facilities directly or not.

At present the vast majority of device drivers are 2D drivers, which means that they operate in device coordinates. In many cases these are known as pixels (addressable screen units), but they may be any other units as appropriate to the device or device interface software. All 3D work (transformations, viewing etc.) is therefore emulated by the GINO library and passed to the device in its local coordinate system.

A 3D device is able to handle 3D coordinates directly (as well as lighting and shading facilities) using their own firmware/hardware to correctly place objects on the screen. In such cases the GINO routines used to define the required mapping and transformation state, pass information directly to the device driver bypassing much of GINO's internal processing. A 3D device is also able to handle 2D coordinates and pixel information by by-passing the modelling and viewing stages, in the same manner that GINO does.

Device Nomination

The nomination of a graphics device is carried out by calling a device nomination routine. Within each device driver library there are usually a number of nomination routines (often known by the name of the device being nominated - eg HP7475 within the HPGL family). A list of nomination routines is given in Appendix B at the beginning of each device driver family.

Examples of nomination calls are:

```
gXwin();
gHplj3();
```

```
call gXwin
call gHplj3
```

which select the X Windows driver and the HPLaserJet III printer respectively.

It is essential that the correct nomination routine is called from within a family device driver as each physical hardware device will require slightly different control (ie. calling gHp7550() will not work correctly on an HP7475 etc.).

Some nomination routines require arguments which control various aspects of the device which are outside the usual GINO control (resolution, position etc.).

A device must be nominated before any input or output is carried out, otherwise GINO error 1 will be output and no drawing takes place. With the absence of a device nomination, a notional device called gDummy() is used by GINO.

Throughout this document, many example programs have 'xxxxx()' in place of the device nomination routine. This should be replaced by a suitable nomination routine as documented in Appendix B.

Device Defaults

Once a device has been nominated, GINO initializes two sets of attributes to their default values. One set consists of GINO's input and output attributes, such as linestyle, hatch table, font style etc. and the second set consists of device specific attributes, such as default paper size, drawing line thickness, character size etc. Both sets are documented in Appendix D, with the device specific attributes marked [DD]. A table at the start of each Appendix B document gives these settings associated with the device driver or nomination routine as appropriate. All of these settings can be enquired through the appropriate enquiry routine.

Device Attributes

Every device has a range of settings for many attributes or specific capabilities in its feasible operation. These include maximum paper limits or number of colours or whether it can do hardware thick lines, or polygon filling or dialogue area control etc. It is often useful to be able to enquire these device capabilities within an application so that different operations can be performed as appropriate. This is achieved through the routine gEnqDeviceState():

gEnqDeviceState(devstate)

This routine returns the complete device state in a structure of type GDEVSTATE. Details of the information returned by this routine can be found in the Reference Section. The routine may be called at any time within a GINO application to enquire device capabilities as some of the settings may be changed through device qualification or other GINO routine calls.

Device Qualification

Immediately after device nomination, but before any drawing has started, the currently selected device is in a special state whereby certain device defaults can be changed. The settings are those that relate to the actual initialization of the device, such as the opening of the window or defining of file output formats and therefore cannot be changed after drawing has commenced. They include:

Destination of graphical output	gSetDeviceFilename()
Setting drawing units	gDefinePictureUnits()
Setting drawing limits	gSetDrawingLimits()
Setting device colour capabilities	gSetColourInfo()

This set of routines are known as device qualification routines and in all cases (except gSetDrawingLimits()) cannot be called once drawing has started. A GINO error 9 will be generated if these routines are called at any other time.

Device Output Filename

In many cases, graphical output is sent from the program to a file which has a default file name or unit number associated with it. The file name may be changed within a program by using:

gSetDeviceFilename(filename, ntype)

where filename is a system dependent filename or pathname to which the device output will be directed using an internally generated file unit. The argument **ntype** provides additional information but which is machine dependent. Users are referred to the Reference Manual for more information on the specific use of these variables.

Drawing Units

When a device is nominated, the device driver defines a default physical drawing area as appropriate to the nomination routine and GINO sets up a default coordinate system with the drawing units set to millimetres. This means that the default drawing limits represent an area measured in millimetres with its origin in the bottom left corner of the paper, screen or window. This is known as the default paper coordinate system.

Specification of Physical Units

It is possible, at the device qualification state, to select different drawing units by using the routine:

gDefinePictureUnits(umils)

Its argument, **umils**, specifies the number of millimetres in the new drawing units. For example:

- To change units to metres - set **umils** to 1000.0.
- To change units to inches - set **umils** to 25.4.

The new drawing units, and its associated paper coordinate system, then remain current for all drawing operations until a viewport is defined or a modelling transformation takes place, after which a new picture or space coordinate system is defined. The current paper units are always used, however, for the defining of drawing limits (see below) and viewport mappings. As with other device qualification routines, gDefinePictureUnits() may not be called after drawing has commenced.

Different drawing units, may be selected for different graphics devices within the same application, but the routine gSetViewport2D() is the preferred method of defining the required mapping between the user coordinate system and the device (see page 219).

Drawing Limits

Where the device has the possibility of variable window sizes or multiple paper sizes, it is possible to alter these limits at the device qualification state, before drawing commences. The routine to alter the drawing limits is:

gSetDrawingLimits(dim, type)

where **dim** is a structure with two elements (xpap and ypap) which define the dimensions of the required drawing area and **type** defines a paper or drawing type as relevant to the currently nominated device. The actual drawing limits can never be greater than the maximum limits for the device.

The specification for each device driver in Appendix B gives details of the default and maximum drawing limits for each nomination routine as well as details of drawing/paper types that are available. It is possible at any stage in a program, to request the current and maximum drawing attributes through the following routines:

gEnqDrawingLimits(dim, type)

and

gEnqMaxDrawingLimits(dim)

The default drawing orientation on printers and plotters is usually landscape however this can be changed to portrait using the routines gEnqDrawingLimits() and gSetDrawingLimits() setting the vertical dimension greater than the horizontal:

C code

```
GDIM land_dim,port_dim;
   gHpjlr();
   gEnqDrawingLimits(land_dim,&type);
   port_dim.xpap=land_dim.ypap;
   port_dim.ypap=land_dim.xpap;
   gSetDrawingLimits(port_dim,type);
```

F90 code

```
type (GDIM) land_dim,port_dim
   call gHpjlr
   call gEnqDrawingLimits(land_dim,itype)
   port_dim%xpap=land_dim%ypap
   port_dim%ypap=land_dim%xpap
   call gSetDrawingLimits(port_dim,itype)
```

Obviously, immediately after device nomination, gEnqDrawingLimits() will return the default drawing limits, whereas after a call to gSetDrawingLimits(), gEnqDrawingLimits() will return the new current drawing limits and drawing/paper type.

Unlike the other device qualification routines, gSetDrawingLimits() can be called at any point within a GINO program. However, it will only take effect after the next call to clear the drawing area at which point the new drawing area will take on the size requested by gSetDrawingLimits() (see below).

Colour Capabilities

GINO is able to output to a wide range of plotters, printers and screen devices. Colour variations can only be achieved within the limitations of the device and to examine a device's capability, use the following routine:

gEnqColourInfo(ndc, ndt)

will return the number of independently selectable colours/greyscales (**ndc**), and the display type (**ndt**) of the currently nominated device.

If it is a monochrome device, **ndc** and **ndt** will return zero. Positive values of **ndt** indicate a colour display and negative values of **ndc** indicate a greyscale display. For a fixed colour/greyscale display (**ndt** = ± 1), **ndc** returns the number of colours that can be selected with gSetLineColour(), but these cannot be changed with gDefineRGB() etc.

For a static colour/greyscale display (**ndt** = ± 2) or a dynamic colour/greyscale display (**ndt** = ± 3), the device has its own colour look-up table with **ndc** entries. Colours are selected by index with gSetLineColour(), the entries of which can be modified by index with gDefineRGB() etc. When defining colours, the result may not be exactly what the user requested but it will be the closest approximation possible given the limitations of the device. Because of the nature of the colour look-up table on a dynamic colour/greyscale display, when a change is made to a particular colour definition it will result in a change to all output drawn previously (and still displayed) in that colour. This gives the facility for instant colour switching.

Some devices will operate in direct colour (24 bit) mode (**ndt** = ± 4), although this is never the default mode of operation for GINO devices if more than one mode is available. Under these circumstances there is no hardware colour look-up table and colour is defined using specified red-green-blue components.. However, an application may continue to use colour indices as GINO maintains its own colour look-up table which can be modified through gDefineRGB() etc. (see page 205). Specific red-green-blue colour settings may also be defined using the gTrueCol() function in conjunction with gSetLineColour() (see page 217). An application can determine whether a device can operate in true colour mode by enquiring the highest colour mode type through a call to gEnqDeviceState().

Note that all colour enquiries will return 24bit RGB triplets when operating in direct colour mode.

GINO provides a facility to change the default device colour capabilities through the routine gSetColourInfo():

gSetColourInfo(ndc, ndt)

This routine allows an application to increase or decrease the number of available colour indices (**ndc**) within the devices maximum capabilities and/or increase or decrease the colour mode (**ndt**) again within the devices maximum capabilities. The limits being found through the routine gEnqDeviceState(). For example, an application may limit the number of colour table entries required, leaving more system resources for other applications, or it may set the device into true colour mode (**ndt**=4) for full 24bit colour work.

Device Initialization

Once the device qualification steps have been performed (if necessary), the next input or output operation requested by the GINO application will initiate an initialization of the graphics device that has been nominated. This will involve the opening the output stream to which the device is connected or opening the output file to which the graphics commands are being sent and sending the necessary commands to initialize the graphics device and set it to a known initialized state. (Enquiry routines do not force device initialization).

It is not until this point that any communication takes place with the device or file, or the graphics screen or window is activated. After the initialization of the graphics device has taken place, the actual input or output operation is then carried out, followed by the rest of the graphics operations of the application. The current graphics device remains active until a request is made to close or suspend it.

New Drawing

During the activation of a graphics device, output may be divided into a number of ‘drawings’. Each one can be seen as a set of output primitives separated by a request to start a new drawing. The routine to start a new drawing is:

gNewDrawing()

The effect of this routine depends on the type of device being used. On graphical displays the screen is erased while on plotters or printers, fresh paper is loaded or wound on.

The request to start a new picture is not actually carried out at the exact time the routine is called within an application. In fact, on printing devices, if no drawing has taken place prior to the call, no action is taken at all by this routine. The delay also allows an application to redefine the drawing limits for the new picture using the routine gSetDrawingLimits(). However, on window devices the action at gNewDrawing() also takes into account any changes made by the user of the application in resizing the active window. Thus, if the application user has resized the window, this takes priority over a programming call to gSetDrawingLimits(). It is therefore essential to always enquire the actual drawing limits after a call to gNewDrawing() using the routine gEnqDrawingLimits() when using a window device.

In addition to possibly redefining the drawing limits, the routine gNewDrawing() also resets the current drawing position to 0.0,0.0,0.0. All other drawing attributes and states are maintained.

Device Dependent Routines

In addition to the standard device independent input and output facilities of GINO, there are a number of device specific facilities available to the user of GINO, such as batch modifications or access to auxiliary drawing areas. These are provided through standard library routines, but the routines may have different effects on different devices. Routines that are not applicable to the currently selected device have no effect (e.g. the selection of alpha-numeric mode is ignored on window devices).

Emptying the Graphics Buffer

GINO buffers the graphical information before transmission to the output device. This has the effect that some output may not appear at the same time as the generating routine is called. It may be desirable to force the emptying of this buffer prior to some numerical processing to ensure the graphics display is completely up to date. The routine gFlushGraphics() can be used for this purpose.

gFlushGraphics()

Auxiliary Drawing Areas

Where a device has the facility of multiple drawing areas in the form of memory planes or display windows as in a windowing system, GINO provides device independent routines for the opening, closing and selection of these as part of a multi-windowing application. In most windowing systems, auxiliary drawing areas are grouped in pairs as each visible drawing area or window usually has an associated invisible memory copy or backing store of the same size. This area is used to update the visible part of the drawing area or window if any portion becomes exposed as a result of the re-sizing or re-positioning of other windows. Therefore, a windowing device driver such as X Windows or Microsoft Windows will automatically create two drawing areas which are identified as area zero (the display) and one (the backing store).

Additional drawing areas may be opened using the following routine:

gOpenAuxDrawingArea(idn, title, xp, yp, width, height)

and depending on the value of the identifier **idn**, are deemed visible or invisible.

All new even numbered areas (2+) are deemed visible and will automatically generate an invisible area of the same size with an identifier of **idn**+1. These will operate in the same manner as the default drawing areas 0 and 1. All new odd numbered areas (3+) are deemed invisible and will not have an associated visible area. The title and origin arguments are only used for even numbered (visible) drawing areas. You cannot open an invisible area if a visible one of identifier (**idn**-1) already exists or vice-versa.

Auxiliary drawing areas may be closed and therefore removed using:

gCloseAuxDrawingArea(idn)

The closure of a visible drawing area (even numbered) will automatically remove its associated backing store (**idn**+1).

The default drawing area for all devices is identifier 1, but GINO's output may be directed to any opened drawing areas through the routine:

gSelectDrawingArea(idn)

As with the default drawing area, it is recommended that all drawing is directed to the backing store (invisible part) of a visible drawing area. This will ensure the visible part of the display is kept up-to-date by the Window Manager. It is possible, however, to direct output to the visible part only, which may offer faster display speeds but suffer display loss if a portion of the drawing area becomes covered and then exposed. The selection of the invisible part (backing store) only is offered through the routines gStartBatchUpdate()/gEndBatchUpdate().

Where output is directed to an invisible drawing area (ie. with no visible part), this can only become visible by copying rectangular regions from this area to another drawing area using the routine gCopyPixelArea() (see page 189). This facility may be useful for animation sequences where picture segments are stored on an invisible drawing area, to be copied at speed to a visible drawing area.

Batch Modifications to Display

The routines gStartBatchUpdate() and gEndBatchUpdate() may be used on some devices to begin and end a batch of modifications to the display surface or hardware display file. Users should refer to Appendix B to see whether or not these routines can be used on a particular device.

gStartBatchUpdate()

gEndBatchUpdate()

When gStartBatchUpdate() is called no further drawing primitives are sent to the display surface, but are batched up in the display file or backing store. When gEndBatchUpdate() is called the display surface is updated with the contents of the display file or backing store. gStartBatchUpdate() contains an implicit call to gEndBatchUpdate().

The process of storing and releasing drawing primitives from a backing store is called double buffering. On certain devices the time to update the drawing surface using gEndBatchUpdate() is noticeably shorter than the time it takes for all the drawing primitives to be processed individually. The double buffering can, therefore, improve real-time display.

Alphanumeric Mode

Many graphics devices have separate windows or planes for graphics and alpha modes where GINO graphics is directed to the graphics area and standard C/C++ or Fortran I/O and GINO error and tracer messages are directed to the alpha area or text window. While standard I/O is not recommended for a complete working application, the user should switch to alphanumeric mode before using such facilities. The routine to switch to alphanumeric mode is:

gSetAlphaMode()

GINO will switch to graphics automatically when required.

Window visibility

The visibility of the dialogue and graphics areas can be changed with the routines:

gSetDialogueVis(diavis)

gSetGraphicsVis(gravis)

where **diavis** or **gravis** may be set to either GINVISIBLE or GVISIBLE. The visibility state of a dialogue or graphics area may be used on some systems to place the appropriate window to the front (GVISIBLE) or back(GINVISIBLE) of the stack of visible windows.

Device Titles

The routine gSetDeviceTitle() provides a means to define a device title or banner as appropriate to the currently nominated device.

gSetDeviceTitle(title)

This routine can for example be used to set a window title or banner.

Device Release and Suspension

When graphical output to a device is complete it should be terminated by releasing the device. This is done by calling the routine:

gCloseDevice()

If a new device is nominated, any previously nominated device is first released. A program should include at least one call to gCloseDevice() since its omission may leave the device in an undesirable state. After a device has been released, further pictures cannot be defined until another device is nominated.

Graphics output to a device can be suspended by the routine:

gSuspendDevice()

This allows an alternative device to be nominated without completely closing or resetting the original device. If such a device is then re-nominated, the device driver may omit part of the initialization phase (eg resetting hardware colour tables etc) allowing output to continue to the same display area without affecting visualization (see page 52).

Using Multiple Devices

Whilst it is not possible to draw to more than one device simultaneously, it is often the case that an application will require to put graphical output onto more than one device in a sequence. The most usual example of this is when a copy of the graphics is required on a printer, plotter or metafile.

The process of exporting graphical output onto a secondary device involves the following steps:

1) Nominate screen device

2) Draw objects

3) Close or suspend screen device

4) Nominate plotter, printer, metafile device

5) Set up appropriate mapping/scaling

6) Redraw required objects

7) Close printer, plotter, metafile device

8) Re-nominate screen device (if required)

Where a simple copy of the graphics is required on a printer, plotter or metafile, it would be usual to close the screen device (using gCloseDevice()), nominate the required hard copy device and then close this device at the end of the application (again using gCloseDevice()).

In a more sophisticated application, the process of drawing to the screen, selecting a print option and going back to the screen may be carried out a number of times. In this case it would be usual to suspend the screen device (using gSuspendDevice()) before nominating the plotter, closing the plotter device on each cycle and closing the screen device at the end of the application.

Mapping to the Second Device

An important consideration when outputting to more than one device is the mapping of your graphics to fit what may be different sized drawing areas on the plotter, printer or metafile. This is especially true if the printer output may be directed to A4 and/or A3 paper on different occasions. The situation may be further complicated if the orientation of the plotter device does not match that of the screen.

There are a number of different methods to cater for this situation:

1) Modify the drawing units (gDefinePictureUnits())

2) Set up an appropriate viewport (gSetViewport2D())

3) Manually scale/rotate the output using GINO transformations (gScale2D()/gRotate2D())

Each method has its advantages and disadvantages.

Saving and Restoring GINO State

It should be noted that the process of nominating a new device has the effect of resetting all current GINO and device attributes to their default settings. Therefore if an application has set up various broken line and/or hatch styles etc., these are all lost when the new device is nominated. For the simple application these can perhaps be re-defined for the secondary device, and again when re-nominating the screen device. However, GINO, provides a means to save and restore all these attributes through the following routines:

gSaveGinoState()

gRestoreGinoState(map)

When gSaveGinoState() is called, all the current attributes, and output tables are saved in a temporary scratch file to be restored when gRestoreGinoState() is called. The argument **map** in gRestoreGinoState() determines whether the saved device limits are mapped to the current device limits. Multiple calls to gSaveGinoState() will store the attributes on a stack to be restored in the reverse order in which they were saved.

The following example shows the use of these routines and also shows the steps required to output to a plotter, printer or metafile device.

C code

```
#include gino-c.h>

      gOpenGino();
      Screen();
/* Define line styles, hatch tables etc. */
      gDefineLineStyle(...
/* Save gino state twice */
      gSaveGinoState();
      gSaveGinoState();
/* Output graphics */
      Output();
/* Suspend screen device */
      gSuspendDevice();
/* Nominate plotter */
      Plotter();
/* Restore GINO state (mapped to plotter limits) */
      gRestoreGinoState(GMAPPED);
/* Output graphics */
      Output();
/* Re-nominate screen */
      Screen();
/* Restore gino state */
      gRestoreGinoState(GABSOLUTE);
/*
```

F90 Code

```
use gino_f90

      call gOpenGino
      call Screen
! Define line styles, hatch tables etc.
      call gDefineLineStyle(...
! Save gino state twice
      call gSaveGinoState
      call gSaveGinoState
! Output graphics
      call Output
! Suspend screen device
      call gSuspendDevice
! Nominate plotter
      call Plotter
! Restore GINO state (mapped to plotter limits)
      call gRestoreGinoState(GMAPPED)
! Output graphics
      call Output
! Re-nominate screen
      call Screen
! Restore gino state
      call gRestoreGinoState(GABSOLUTE)
!
```

Note that gSaveGinoState() is called twice before nominating the plotter, in order to save two copies of the current GINO state. The first restoration (carried out after the plotter is nominated), specifies that the saved drawing limits are to be mapped to the new plotter limits. This sets up an appropriate viewport which ensures that the output on the plotter contains all the output on the screen. The second restoration is carried out once the screen device is re-nominated to ensure it is restored to the same state as before the plotter nomination.

Duplicating Output

There are no automatic procedures in GINO to duplicate the output generated on a screen device to be sent to the desired printer/plotter. This is because it is necessary, by some means or other, to re-issue the drawing commands on the selected output device, and for the reasons stated above these may not be the same dimension or orientation as that on the screen. It may also be desirable to highlight parts of a drawing differently on monochrome output than on a colour screen.

It is therefore the responsibility of the application writer to re-draw (with or without modification) the required output onto the printer/plotter using GINO drawing routines.

The simplest method of easing this task is to put all the drawing commands into a separate routine, so that it may be called when the screen or plotter device is nominated (as shown in the example above).

A more sophisticated method is to store the output into one or more picture segments which can then be redrawn on the new device. This facility is described later in the document (see page 423).

Chapter 3

IMPORTING AND EXPORTING

Importing and Exporting Introduction

An extension to the process of simply generating graphics using the GINO library, is the issue of generating the right graphics for importing into Word Processing, DTP or image processing packages and importing graphics into GINO applications.

The wide range of metafile formats that are available often makes the selection of the appropriate one even more difficult. This section outlines the different formats that GINO can handle, their advantages and disadvantages, hopefully making the choice easier.

This section also covers the necessary steps that need to be taken to duplicate graphics that has been displayed on some interactive device or screen, out to an appropriate metafile or printer device.

Overview

The process of importing and exporting graphics is achieved through metafiles. These are files that store graphical information in a published format that are generated and/or interpreted by graphics software such as Word Processing, Image Processing packages or WEB browsers. The GINO library can generate a number of different metafile formats and can interpret a smaller set. GINO does not claim to handle large numbers of metafile formats as there are a number of widely available packages that can read, convert and export a wide range of formats that GINO users can use.

Metafiles are divided into two classes, those that store the graphical information as a bitmap (i.e. A finite number of 'pixels' of information representing a rectangular drawing area), or as a vector format (containing the lines and characters etc. that make up the picture). Bitmap formats are often compressed to save space but vector formats are usually the more complex and can store additional information including data and images.

In addition to the external metafile formats that GINO can handle, GINO has its own proprietary format called SAVDRA which is included in the summary below.

Metafile Formats

There are many advantages and disadvantages for using a particular type of file format when exporting GINO output to a word-processor or DTP package. Choosing one format over another may result in smaller file sizes, better capabilities such as a greater number of colours or fonts, or provide the ability to 'edit' the graphics once it has been imported.

BMP

BMP files are image files and are therefore more suited to image handling packages. GINO only handles the uncompressed BMP format resulting in fairly large files on creation, but GINO does generate and interpret both colour indexed and true colour formats of BMP file. The size of the imported image is determined by the call to gSetDrawingLimits(), so ensure that gSetDrawingLimits() is called matching the size of the GINO picture. Keeping gSetDrawingLimits() to the correct size, also keeps the file size to the minimum possible.

Note that BMP files are created using the gImage() driver as detailed in Appendix B, however shaded 3D images using OpenGL are exported to BMP via the gWogl() driver.

CGM

The CGM standard provides for three encodings - character, binary and clear text, the first two of which can be generated and interpreted within GINO.

Users of CGM are referred to the functional specification and encodings of the standard as published by BSI or ISO. Four documents are available:

Functional Specification	BSI 6945(1)	ISO 8632-1
Character Encoding	BSI 6945(2)	ISO 8632-2

Binary Encoding	BSI 6945(3)	ISO 8632-3
Clear Text Encoding	BSI 6945(4)	ISO 8632-4

CGM metafiles are acceptable to many other software systems and so pictures may be transferred between different vendors' packages, but as there are many different encodings not many systems generate or interpret all of them. The Binary encoding is more efficient for time and file space requirements on all systems that use it.

CGM files are vector-format files. Up to 255 colours are supported but only one font is available.

File size is quite small and most packages can only read Binary Integer format (gCgmbi()). The size of the picture is automatically detected by the importing program, so gSetDrawingLimits() accuracy is not important.

DXF

DXF files are vector files originally developed for AutoCAD. File sizes are very big and the format does not include many hardware features. DXF files include an image size header calculated from the default paper size or a call to gSetDrawingLimits(). Ensure that this matches the size of the GINO picture as this is used to scale the image in the importing program.

EPS (Encapsulated Postscript)

EPS files generated from GINO are vector-format files. GINO does not include an image header in the file but most packages do not now require one. (If one is required, the image will not appear on the screen and the graphics will only print on a Postscript-compatible printer. The recommended EPS routine to use is gEpsexp() which provides a default orientation of portrait. (The other Postscript nomination routines assume a default of landscape).

The GINO Postscript file contains the size of the image in a BoundingBox comment and this is used in the importing program to display the correct size of image. By default, the BoundingBox value is calculated automatically by the GINO driver and is placed at the end of the file. Some filters however, require the BoundingBox value to be at the start of the file and this can be achieved by setting the iprop(3) parameter in gEpsexp() accordingly (See Page 816).

Postscript's advantage is its universal acceptance and the availability of many fonts, giving the most professional appearance in most cases.

ICO

Windows Icon files are a particular type of image file that is usually restricted to 16x16 or 32x32 pixels. GINO can import these files but not create them.

JPEG

JPEG is a standardized compression method for full-colour and grey-scale images. JPEG is intended for compressing “real-world” scenes. Line drawing, cartoons and other ‘non-realistic’ images are not its strong point. JPEG is lossy, meaning that the output image is not exactly identical to the input image. The amount of lossyness can be controlled by a quality setting, with low quality giving very high levels of compression. JPEG files are accepted by all common Web browsers.

PNG

PNG (Portable Network Graphics) is an extensible file format for the lossless, portable, well-compressed storage of raster images. PNG provides a patent-free replacement for GIF and can also replace many common uses of TIFF. Indexed-colour, grayscale, and truecolor images are supported, plus an optional alpha channel.

PNG is designed to work well in online viewing applications, such as the World Wide Web, so it is fully streamable with a progressive display option. PNG is robust, providing both full file integrity checking and fast, simple detection of common transmission errors.

SAVDRA

SAVDRA is GINO’s proprietary metafile format and is designed to match the functionality of GINO catering for all the features of the package. The format consists of fixed length records of ASCII printable characters permitting easy file transfer from one GINO system to another.

Two different forms of SAVDRA metafile exist, one for storing complete drawings (gSavdra()) and one for storing graphical elements or components in the form of a library (gSavpic()).

In general, gSavdra() is used for storing complete drawings that might be subsequently exported to a number of different hard copy devices. Alternatively, it can be used to enable drawings to be produced a number of times on different devices. In addition, it may be used to generate a drawing on one machine and to produce a graphic output on another.

The alternative form, gSavpic() is designed to store picture segments for subsequent use as library objects (see page 423). In this form, global information such as line and hatch styles are not stored in the file and the segments consist of graphical primitives that can be adapted to the application that uses them.

WMF

WMF files are vector files and are amongst the smallest. Some fonts are available but appearance may be a problem if fonts are substituted by the import program. The size of the imported image is determined by the parameters in the nomination routine therefore ensure that these match the size of the GINO graphics as close as possible.

XWD

These are image dumps used on Unix systems supporting X windows. Normally the files are created using the Unix command xwd and read in using the corresponding command xwud (X Windows UnDump). GINO's XWD metafiles can be read into third party applications or displayed on the X terminal or screen.

Summary

Below is a summary of image file types:

Metafile	Class	GINO Export/Import
BMP	Bitmap	Export & Import
CGM	Vector	Export & Import
DXF	Vector	Export
EPS	Vector	Export
ICO	Bitmap	Import
JPEG	Bitmap	Export & Import
PNG	Bitmap	Export & Import
SAVDRA	Vector	Export & Import
WMF	Vector	Export
XWD	Bitmap	Export & Import

Exporting Metafiles from GINO

Metafiles are created in GINO in the same way that graphics is drawn to any graphics device. In other words, the appropriate metafile driver is nominated through the device nomination routine, an external file name is assigned, the drawing is done and the device is closed (see page 39).

For example, to create a BMP file the following code would be used:

```
gBmp();
gSetDeviceFilename
         (”example.bmp”,0);
/*
   Do Drawing
*/
gCloseDevice();
```

```
call gBmp
call gSetDeviceFilename &
                 (’example.bmp’,0)
!
! Do Drawing
!
call gCloseDevice
```

If a file name is not assigned, a default name of BMP.OUT is created, where the prefix of the file name is the same as the nomination routine name. It is better practice to assign a name with the appropriate suffix (e.g. example.bmp) as most window systems will give the resulting file an appropriate icon and enable the file to be viewed by any installed metafile viewing packages.

Details on the specifications of the GINO drivers for all the metafile formats can be found in Appendix B.

Metafiles into External Packages

The following table indicates which packages can read metafiles generated from GINO and gives hints on either how to read the file in, what to expect visually and what can be done to the picture once read in. The packages that support editing of the graphics handle editing in different ways; either the picture may need to be read in as a drawing, the picture may need ‘ungrouping’ after being read in or the picture may simply require double-clicking to go into edit mode. Consult the package documentation for more details on this.

The table also lists the recommended GINO nomination routine together with a rough guide on file size comparisons. The example filesizes were all generated from exactly the same GINOGRAF program containing a mixture of polygon fills, hardware text, software text, lines and arcs.

	BMP	CGM	DXF	EPS	WMF	JPG	PNG
Recommended routine:	gBmp()	gCgmbi()	gDxf()	gEpsexp(...)	gWmfp(...)	gJpeg()	gPng()
Example filesize:	692K	40K	745K	36K	30K	61K	14K
Package							
Adobe PageMaker V5.0	✓	No	✓	✓[1,6]	✓	✓	✓
Adobe Photoshop V5.0	✓	No	No	✓	No	✓	✓
Borland Delphi v6.0	✓	No	No	No	✓	✓	No
CorelDraw V9.0	✓	✓[10]	✓	✓[2]	✓	✓	✓
Corel Ventura V8.0	✓	✓[10]	✓	✓[2]	✓	✓	✓
Corel WordPerfect V9.0	✓	✓	✓	✓	✓[7]	✓	✓
Lotus Freelance v9.6	✓	✓[3]	✓	No	✓	✓	No
Lotus WordPro v9.6	✓	No	No	✓[1,6]	✓	✓	✓
Lotus 1-2-3 v9.6	✓	✓[10]	No	No	✓	✓	No
Microsoft Word V6.0	✓	✓[5][10]	No	✓[1,6]	✓	No	No
Microsoft Word 2000	✓	✓[5][9]	No	✓[1,6]	✓	✓	✓
Microsoft Excel 2000	✓	✓[5][9]	No	✓[1,6]	✓	✓	✓
Microsoft Internet Explorer 6.0	No	No	No	No	✓	✓	✓
Microsoft Powerpoint 2000	✓	✓[5,9]	No	✓[1,6]	✓	✓	✓
Netscape Navigator V3.03 (OpenVMS)	No	No	No	No	No	✓	No
Netscape Communicator V4.5 (UNIX)	No	No	No	No	No	✓	✓
Netscape Communicator V6.2 (Windows)	No	No	No	No	No	✓	✓
Paint Shop Pro V6.0	✓	✓[4][10]	✓	✓	✓	✓	✓
Quark Xpress V4.0	✓	No	No	✓[1,6]	✓	✓	No
Visual Basic V6.0	✓	No	No	No	✓	✓	No
Visual Studio .NET 2002	✓	No	No	No	✓	✓	✓

[1] Picture doesn't show up on screen but will print to a Postscript printer

[2] Loads using the interpreted EPS filter only

[3] Multi-polygons are not drawn (This shows up with polygonally filled characters as well)

[4] Colours may be displayed wrongly due to a limited fixed palette

[5] The image does not appear if a gNewDrawing() is at the beginning of the GINO program

[6] Use **iprop(3)** in gEpsexp() to move the Bounding Box comment from the bottom to the top

[7] Hardware fonts may not show correctly

[8] Must check box "Make Postscript Object" or "Create a Postscript Object" when importing

[9] Cell arrays (pixel data) only displayed if the cell has < 2048 pixels and when interpreting char. encoded files (gCgmchi)

[10] Cell array and pixel images are not displayed

Importing Metafiles into GINO

GINO can handle the import of several types of metafile; its own proprietary SAVDRA metafile, CGM character and binary encodings and a number of image metafile formats.

In all enquiries and interpretations of SAVDRA and CGM metafiles, the appropriate routines require a file unit or pointer which identifies the external file itself. This is obtained through use of the gFopen() function which opens the desired metafile (see page 28).

For example, where an enquiry is to be made on the SAVDRA metafile, ‘picture.sav’, the following code is required to open the file:

```
GFILE *file;

file=gFopen("picture.sav","r");
```

```
integer file

file=gFopen('picture.sav',GREAD)
```

SAVDRA Metafile

As described in the above section, the SAVDRA metafile can be used to store complete drawings (using gSavdra()) or a library of picture elements (using gSavpic()).

Being proprietary to the GINO library a number of facilities are available in the GINO library to enquire information about a SAVDRA file and to interpret its contents.

SAVDRA Enquiries

In order to find the drawing limits of an existing SAVDRA metafile, the routine gEnqSavdraDimension() is provided:

gEnqSavdraDimension(file, type, dim)

where **file** is a pointer to a GINO file unit opened by gFopen() (see above) from which the file is read, **type** returns the type of metafile (1=SAVDRA, 2=SAVPIC) and **dim** contains the limits of the positive quadrant of that file.

In order to enquire information about segments within a SAVDRA metafile, two routines are provided:

gEnqSavdraSegList(file, list, n, icount)

gEnqSavdraSegAttribs(file, nseg, att)

where **file** is the opened file unit. The routine gEnqSavdraSegList() returns a **list** of segment numbers that are contained in the metafile and the routine gEnqSavdraSegAttribs() returns the segment attributes of an individual segment **nseg** in the metafile. The information is returned in a structure of type GPICATT which includes whether the segment exists, its anchor position and other attributes.

In all cases, the file is then rewound ready for interpretation as required.

SAVDRA Interpretation

SAVDRA code is interpreted using the routine:

gGetDrawing(file, nseg, mode, paper)

where **file** is a pointer to a GINO file opened by gFopen() or a Fortran 90 file unit (see page 28) from which the code is to be read and **nseg** specifies the picture to be drawn. If **nseg** is -1, all the pictures in the file will be drawn. The values of **mode** and **paper** enable the code to be interpreted in different ways to satisfy the individual requirements.

For the 'quick look' facility, generally **mode** would be GMAPPED and **paper** =GPROGRAM. The output would then be drawn to utilize as much of the drawing area as possible. For the final production of the output, **mode** would be GABSOLUTE and **paper**=GPROGRAM.

Similarly for exactly reproducing drawings on a number of devices and/or machines, the combination of **mode** = GABSOLUTE and **paper** = GMETAFILE would be used. If the code were to be used as part of a layer drawing and combined with output from other programs **mode** = GTRANSFORMED and **paper** = GPROGRAM would probably be used.

Modes GABSOLUTE, GMAPPED and GTRANSFORMED reproduce only those parts of the drawing that were drawn on the positive quadrant of the SAVDRA device. Mode GWHOLE reproduces the entire SAVDRA drawing (-ve and +ve quadrants) uniformly scaled to fit within the current clipping limits.

Consider for example the following program for producing a file containing the code for defining a 10mm square on a drawing area of 200mm x 200mm.

```
static GDIM paper =
               {200.0,200.0};

gSavdra();
gSetDrawingLimits(&paper,0);
gOpenSeg(4);
gMoveTo2D(0.0,0.0);
gDrawLineTo2D(10.0,0.0);
gDrawLineTo2D(10.0,10.0);
gDrawLineTo2D(0.0,10.0);
gDrawLineTo2D(0.0,0.0);
gCloseDevice();
```

```
type (GDIM) :: paper = &
              GDIM{200.0,200.0}

call gSavdra
call gSetDrawingLimits(paper,0)
call gOpenSeg(4)
call gMoveTo2D(0.0,0.0)
call gDrawLineTo2D(10.0,0.0)
call gDrawLineTo2D(10.0,10.0)
call gDrawLineTo2D(0.0,10.0)
call gDrawLineTo2D(0.0,0.0)
call gCloseDevice
```

For a ‘quick look’ on a PC Windows screen, the following program with mode=GMAPPED could be used:

```
GFILE *file;

file=gFopen("box.sav","r");
gMwin(Inst,hPrevInst);
gGetDrawing(file,4,GMAPPED,
                       GPROGRAM);
gCloseDevice();
```

```
integer :: file = 11

file=gFopen('box.sav',GREAD)
call gMwin
call gGetDrawing(file,4,GMAPPED,&
                         GPROGRAM)
call gCloseDevice
```

A scaling factor would be produced to ensure that the 200mm x 200mm drawing area would fit onto the PC Window, whose size would vary according to the resolution and monitor size. In other words the 10mm square would no longer be 10mm.

To reproduce the drawing on a plotter the following program with mode = GABSOLUTE could be used:

```
GFILE *file;

file=gFopen("box.sav","r");
gHp7475();
gGetDrawing(file,4,GABSOLUTE,
                     GMETAFILE);
gCloseDevice();
```

```
integer file

file=gFopen('box.sav',GREAD)
gHp7475
call gGetDrawing(file,4, &
              GABSOLUTE,GMETAFILE)
call gCloseDevice
```

This requests a drawing area of 200mm x 200mm and draws a 10mm square; in other words it is exactly as specified in the generating program.

SAVPIC Interpretation

SAVPIC code is interpreted using the routine:

gGetPicture(file, nseg)

Since it is assumed that gGetPicture() will be used to recall picture segments as if they were library objects, it is necessary for the routine to ensure that any drawing qualifiers (e.g. line styles) set within an object do not affect the calling program; thus gGetPicture() stores these on entry and then resets them on exit.

Consider, for instance, the following program to generate a metafile containing definitions of two character strings.

```
gSavpic();
gSetDeviceFilename
      (”string.sav”,0)
gOpenSeg(1);
gDisplayStr(“AB”);
gSetCharSize(5.0,5.0);
gDisplayStr(“CD”);
gCloseDevice();
```

```
call gSavpic
call SetDeviceFilename &
      (’string.sav’,0)
call gOpenSeg(1)
call gDisplayStr(‘AB’)
call gSetCharSize(5.0,5.0)
call gDisplayStr(‘CD’)
call gCloseDevice
```

The following interpreting program, displayed on a X Windows display should produce the results indicated:

Setting a large character size would generate the characters shown below:

ABCDXY

```
GFILE *file;

file=gFopen(”string.sav”,”r”);
gXwin();
gSetCharSize(10.0,10.0);
gGetPicture(file,1);
gDisplayStr(”XY”);
```

```
integer file

file=gFopen(’string.sav’,GREAD)
call gXwin
call gSetCharSize(10.0,10.0)
call gGetPicture(file,1)
call gDisplayStr(’XY’)
```

Setting a small character size would generate the characters shown below:

ABCDXY

```
GFILE *file;

file=gFopen("string.sav","r");
gXwin();
gSetCharSize(2.5,2.5);
gGetPicture(file,1);
gDisplayStr("XY");
```

```
integer file

file=gFopen('string.sav',GREAD)
call gXwin
call gSetCharSize(2.5,2.5)
call gGetPicture(file,1)
call gDisplayStr('XY')
```

Note the characters 'CD' have their definition saved within the picture segment and so remain the same in both cases.

When interpreting picture segments, gGetPicture() will always restore them in the same units as when they were created, i.e. equivalent to **mode** = GTRANSFORMED and **paper** = GPROGRAM in gGetDrawing(). However, their form when displayed can be changed by using any of the GINO transformation routines prior to calling gGetPicture().

Mixing SAVDRA Generators and Interpreters

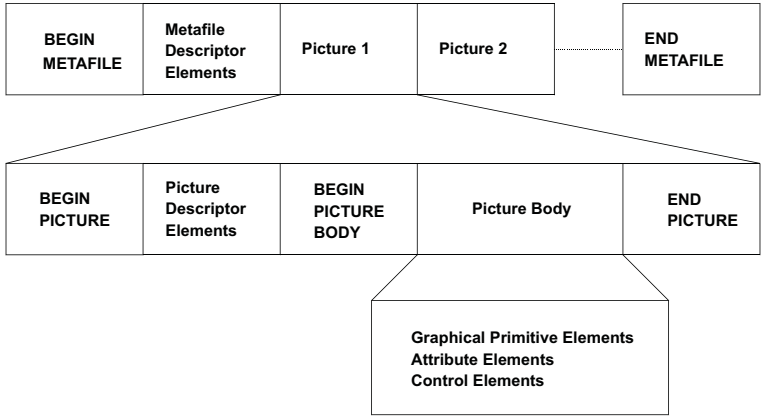
In general the interpreting routine gGetDrawing() will always be used for code that is generated using gSavdra() and routine gGetPicture() will be used for code that is generated using gSavpic(). However, it is possible for code that is generated using gSavdra() to be interpreted using the routine gGetPicture(). (Note: it is not possible to use gGetDrawing() for interpreting gSavpic() code). In this case, the segment headers present in the gSavdra() code will not be decoded by gGetPicture(), only the vector part of the segment will be interpreted. This may mean that the resulting graphic output will not be the same as if the code has been interpreted using gGetDrawing().

Workspace requirements for Drawing Interpretation

Both the gSavdra() and gSavpic() metafile generators can store complex polygons within the file. Therefore, when interpreting any gSavdra() metafile that may contain polygons, it is necessary to declare workspace for their retrieval (see page 33). This includes polygons created with gStartPolygon() as well as metafiles that contain gFillPolygonTo2D(), gFillPolygonBy2D() and gFillRect() polygons since these are converted to complex polygons upon storage in the metafile generators.

CGM Metafiles

A CGM metafile consists of the following structure:



Each element having a unique identifier followed by a variable number of parameters (possibly none).

Pictures are separated within a CGM file by using the GINO routine gNewDrawing().

CGM Interpretation

CGM metafiles can be interpreted in one of two ways, either as a whole or element by element. The first method corresponds closely to the gGetDrawing() facility in the previous section where the whole file is read in and interpreted onto the current output device (which may be another metafile). The alternative method gives the user the opportunity to examine the file element by element and interpret or skip over it as required.

To interpret a complete metafile the following routine is used:

gCGMInterpreter(code, file, nseg, mode, errlev)

where **code** specifies the encoding (GCGMCHAR=character, GCGMBINARY=binary), **file** is a pointer to a GINO file unit opened by gFopen() from which the file will be read, (**nseg** is reserved for future use), **mode** specifies the interpretation mode and **errlev** sets the error checking level (See 'Error Handling within CGM').

For abstract scaled metafiles (ie. those in which the pictures have no scaling information), the picture will be scaled to fit the current GINO window irrespective of the interpretation mode. For metric scaled metafiles (the type which GINO will always create), interpretation mode GABSOLUTE will draw the encoded pictures to the same physical size that they were generated. Mode GMAPPED will scale the pictures to fit the current GINO window as for abstract metafiles. Mode GTRANSFORMED will restore the metafile to the same size that it was generated, but subject to the current GINO transformation.

To interpret a character encoded CGM file onto an X Windows device the following program could be used:

```
#include <gino-c.h>
main()
{
  GFILE *file;

  gXwin();
  file=gFopen("file.cgm","r");
  gCGMInterpreter(GCGMCHAR,file,
            GALL,GMAPPED,GFULL);
  gCloseDevice();
}
```

```
program cgm
use gino_f90

integer :: file

call gXwin
file=gFopen('file.cgm',GREAD)
call gCGMInterpreter(GCGMCHAR, &
        file,GALL,GMAPPED,GFULL)
call gCloseDevice
stop
```

In order to examine a CGM metafile element by element, five routines are provided:

to open the file:

gOpenCGMFile(code, file, mode, errlev)

to get next element:

gGetCGMElement(element)

to skip over this element:

gSkipCGMElement(element)

to interpret this element:

gInterpretCGMElement(element)

to close the file:

gCloseCGMFile()

To interpret a binary encoded CGM file onto an X Windows device but skipping over all cell array primitives the following program could be used:

C code

```
#include <gino-c.h>
main()
{
   GFILE *file;
   int element;
   gXwin();
   file=gFopen("file.cgm","r");
   gOpenCGMFile(GCGMBINARY,file,GMAPPED,GFULL);
   gGetCGMElement(&element);
/* Check for end of metafile */
   while (element != 133) {
/* Check for cell array primitive */
      if (element != 40)
         gInterpretCGMElement(element);
      else
/*  Skip over cell array */
          gSkipCGMElement(element);
      gGetCGMElement(element);
   }
   gCloseCGMFile();
   gCloseDevice();
}
```

F90 code

```
program cgm
use gino_f90

   integer file
   integer element
   call gXwin
   file=gFopen('file.cgm',GREAD)
   call gOpenCGMFile(GCGMBINARY,file,GMAPPED,GFULL)
   call gGetCGMElement(element)
! Check for end of metafile
   do while (element .ne. 133)
! Check for cell array primitive
      if (element .ne. 40) then
         call gInterpretCGMElement(element)
      else
!  Skip over cell array
          call gSkipCGMElement(element)
      end if
      call gGetCGMElement(element)
   end do
   call gCloseCGMFile
   call gCloseDevice
   stop
```

CGM Elements

The full list of legal CGM element identifiers together with notes on their use by the GINO generator and interpreter is found in Appendix B.

Polygon Handling within CGM

All of CGM's filled graphical primitives are handled by GINO using its polygon definition and filling routines (see page 245). The CGM interpreter has a built in polygon workspace of 2000 words but this may be increased by calling gSetWorkspaceLimit() and gDefinePolygonWorkspace() prior to calling gCGMInterpreter() or gGetCGMElement() (see page 33).

e.g:

```
gSetWorkspaceLimit(20000);
gDefinePolygonWorkspace(10000);
```

```
call gSetWorkspaceLimit(1,20000)
call gDefinePolygonWorkspace &
                          (10000)
```

Note: Where polygon workspace is declared, users should be aware that polygon identifiers 33000 upwards are used by CGM. The polygon workspace will also be cleared after every polygon area unless the user has declared polygons (i.e. used the GINO routine gSetPolygonIdent()) before CGM fills the first polygon. If the user has been using polygon identifiers before requiring filled polygons but still requires CGM to clear the polygon workspace after every polygon, the user should call gSetPolygonIdent(0). This ensures that less polygon workspace is required as only one polygon area is stored at a time.

Error Handling within CGM

All errors encountered by the CGM interpreter are handled in the same way as other GINO errors except that they have been given a distinct range and are therefore identified by the prefix 'GINOCGM error/warning'. A full explanation of CGM errors is given in Appendix E, but they are grouped in the following sets:

Errors 1 to 19	Element found when interpreter in incorrect state
Errors 20 to 99	Unknown or illegal element found
Warnings 100 to 129	Invalid index
Warnings 130 to 150	Invalid colour definition
Warnings 200 to 250	Value has lost precision (ie. is outside range specified by relevant precision)
Warnings 300 to 399	Invalid attribute
Warnings 400 to 499	Invalid descriptor or control element
Errors 700 to 800	Data handling error (I/O error, buffer error)

For errors 1 to 19 the ‘state’ of the metafile refers to the stage of metafile interpretation which may be one of the following:

State 1	KMFCL	Metafile closed
State 2	KMFDS	Metafile description
State 3	KPIDS	Picture description
State 4	KPIOP	Picture Open
State 5	KPICL	Picture Closed
State 6	KPATX	Partial Text

The error level setting used in both gCGMInterpreter() and gOpenCGMFile() can be one of three values:

Errlev=0 No error checking

Errlev=1 Fast error checking - skip rest of element after first error

Errlev=2 Full error checking - continue processing element after error

CGM Limitations

The following limitations are imposed on the interpretation of CGM files:

The maximum number of points in a point list is 1024.

The maximum length of a character string is 256 characters.

The maximum number of colours interpreted is 2048 (this also applies to the number of elements in a cell array primitive).

The maximum internal buffer size (for binary elements) is 4096 bytes.

All precisions are catered for except 64-bit precision within binary encodings.

Image Metafiles

The third type of metafile that can be imported into GINO is the image or bitmap type. These are read in from the required metafile, into an integer array rather than placing the image straight on the current output device. In this way they can be manipulated and or processed by the application, and the (resulting) image can be displayed using the full control of the GINO image handling routines (see page 189).

Two routines are provided to interpret image metafiles, one to enquire the type and attributes and one to actually read the metafile into the integer work array. At present these facilities handle Windows BMP files, Windows ICO files, X Windows Dump files, JPEG and Portable Network Graphics (PNG) files.

The routine gEnqImageFile() can be used to enquire the type and attributes of an external image file by examining the file header only.

gEnqImageFile(file, type, xgrid, ygrid, nbpp, ncols)

where **type** is the metafile type, **xgrid** and **ygrid** give the image dimensions and **nbpp** and **ncols** return colour information. Once this information has been gathered (if it is not already known), the actual image can be read using the following routine:

gGetImageFile(type, file, coldef, offset, collim, xgrid, ygrid, npix, pixbuf)

This routine will read in the contents of an external image file into the integer array **pixbuf** ready for display by the routine gDrawPixelArea(). The interpretation of the colour table held in the image file (if one is present) is governed by the three colour definition arguments **coldef**, **offset**, **collim**. These allow an application to ignore the image file colour table altogether, load it into a specified colour range or map it to the existing GINO colour table.

The following program reads an image saved from Paintshop Pro. The image is 800 x 600 pixels and was saved with 256 colours. The GINO program first needs to declare space of 800x600=48000 integer words in an array, then reads the image with the routine gGetImageFile(). **coldef** is set to 1 so that GINO will define all the colours that are being used by the image.

The routine gDrawPixelArea() is then used to actually draw the image (in this case to an MWIN window). It uses the **xgrid**,**ygrid** values returned by gGetImageFile() and if the full image is required, sets **isx**,**isy** to 1,1. The image is positioned according to the first two values **ix**,**iy** which refer to the screen pixel position starting from the top left corner of the screen. This example draws the image starting at position 1,1.

C code

```
int pixbuf[48000],xgrid,ygrid;
:
gMwin();
gGetImageFile(1,'EGNS.BMP',1,0,0,&xgrid,&ygrid,
                                 48000,pixbuf)
gDrawPixelArea(1,1,xgrid,ygrid,1,1,xgrid,ygrid,pixbuf)
:
```

F90 code

```
integer pixbuf(48000),xgrid,ygrid
:
call gMwin
call gGetImageFile(1,'EGNS.BMP',1,0,0,xgrid,ygrid, &
                                        48000,pixbuf)
call gDrawPixelArea(1,1,xgrid,ygrid,1,1,xgrid,ygrid,pixbuf)
:
```

These routines handle 1, 4, 8, and 24 bit colour images.

Chapter 4

2D DRAWING

2D Drawing Introduction

GINO provides 2D line drawing facilities for:

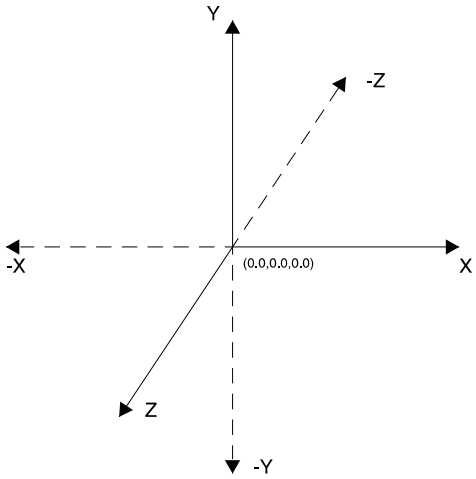
- Positioning
- Single straight lines
- Polylines
- Polyline sets
- Circular arcs
- Parametric curves
- B-spline curves

Pen

Historically, computer graphics initiated in the days of pen plotters and whilst today the drawing method can be a series of pixels on a raster screen, a plotter pen or a light beam etc. depending on the output device, the term “pen” will be used throughout.

Axes

The 2D coordinate system used is right-handed as shown below, with the X-axis horizontal, the Y-axis vertical.

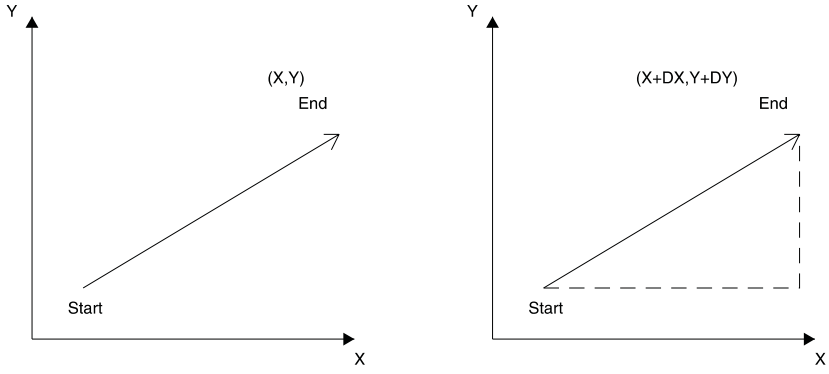


The Right Handed Coordinate System

Two-dimensional drawing can be anywhere within X,Y space, with the initial origin being the bottom left-hand corner of the drawing area. Three-dimensional drawing is covered separately, later in this document (see page 277).

2D Start and End Pen Position

All drawing starts from the position at which the pen was left by the previous drawing instruction - this is termed the start pen position. Initially, the position of the pen is at (X,Y) = (0.0,0.0). The arguments for all 2D drawing routines define the point at which the pen will be left after executing the routine. This is termed the “end pen position”. The end position of one routine becomes the start position for the next. The arguments can specify the absolute coordinates of the end pen position, or the end pen position relative to the start position.



Pen Position

2D Naming Conventions

The naming convention for the 2D drawing routines is as follows:

(a) The initial part indicates the routine:

gMove*	- positioning
gDrawLine*	- drawing straight lines
gDrawArc*	- drawing circular arcs
gDrawPolyline*	- drawing a series of straight lines
gDrawPolylineSet*	- drawing a set of polylines
gDrawAkima*	- drawing a curve using an averaging method due to Akima
gDrawCurve*	- drawing a piecewise parametric cubic curve
gDrawSpline*	- drawing a cubic spline curve

(a) The latter part indicates the type of coordinates:

To	- absolute
By	- relative

(c) The last part indicates dimension:

**2D	- two dimensions
**3D	- three dimensions (see page 277)

Positioning

The routines for “straight line movement” are:

gMoveTo2D(x, y)

gMoveBy2D(dx, dy)

Examples:

- To position the pen at point (1.5,2.5) the following statement could be used:

```
gMoveTo2D(1.5,2.5);
```

```
call gMoveTo2D(1.5,2.5)
```

- To increment the start pen position by **xa** in the X-direction and **ya** in the Y-direction the following statement could be used:

```
gMoveBy2D(xa,ya);
```

```
call gMoveBy2D(xa,ya)
```

Straight Lines

The routines for drawing straight lines are:

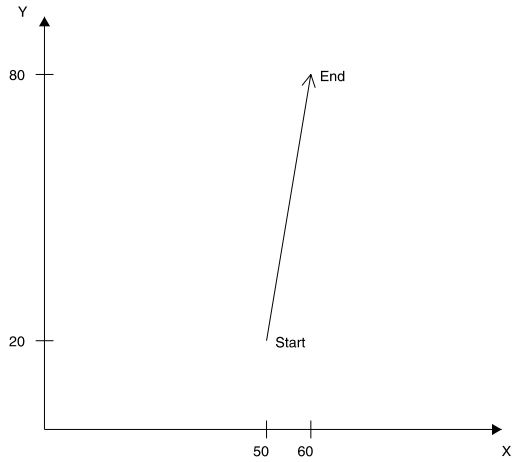
gDrawLineTo2D(x, y)

gDrawLineBy2D(dx, dy)

For example - to draw a straight line from the point (50.0,20.0) to the point (60.0,80.0) the following statements can be used:

```
gMoveTo2D(50.0,20.0);
gDrawLineTo2D(60.0,80.0);
```

```
call gMoveTo2D(50.0,20.0)
call gDrawLineTo2D(60.0,80.0)
```



Straight Line Drawing

Alternatively, the following statements can be used:

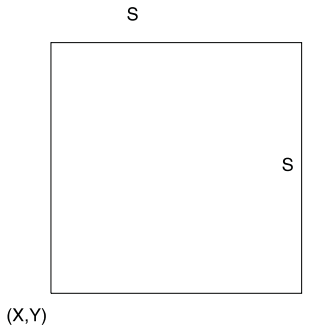
```
gMoveTo2D(50.0,20.0);
gDrawLineBy2D(10.0,60.0);
```

```
call gMoveTo2D(50.0,20.0)
call gDrawLineBy2D(10.0,60.0)
```

The following sequence of statements would draw a square of side S, positioned with the bottom left-hand corner at point (X,Y):

```
/* Position */
      gMoveTo2D(x, y);
/* Draw bottom line */
      gDrawLineBy2D(s,0.0);
/* Draw right vertical */
      gDrawLineBy2D(0.0,s);
/* Top line */
      gDrawLineBy2D(-s,0.0);
/* Draw left vertical */
      gDrawLineBy2D(0.0,-s);
```

```
! Position
      call gMoveTo2D(x, y)
! Draw bottom line
      call gDrawLineBy2D(s,0.0)
! Draw right vertical
      call gDrawLineBy2D(0.0,s)
! Draw top line
      call gDrawLineBy2D(-s,0.0)
! Draw left vertical
      call gDrawLineBy2D(0.0,-s)
```



Straight Line Drawing

Polylines

The routines for drawing multiple straight lines are:

gDrawPolylineTo2D(npts, points2)

gDrawPolylineBy2D(npts, points2)

where **points2** is an array of structures of type GPOINT containing two real elements (**points2.x** and **points2.y**).

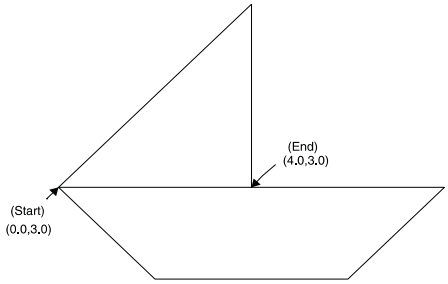
For example - to draw the six lines shown in the figure below, an array **points** of type GPOINT is initialized with six coordinate pairs as appropriate to the language:

C code

```
static GPOINT pt[6] = {2.0,1.0, 6.0,1.0, 8.0,3.0,
                       0.0,3.0, 4.0,7.0, 4.0,3.0};
      .

/* Move to start */
      gMoveTo2D(0.0,3.0);
/* Draw figure */
      gDrawPolylineTo2D(6,pt);
```

F90 code

```
type (GPOINT), dimension(6) :: pt = (/ &
      GPOINT{2.0,1.0), GPOINT(6.0,1.0), GPOINT(8.0,3.0), &
      GPOINT(0.0,3.0), GPOINT(4.0,7.0), GPOINT(4.0,3.0} /)
      .
! Move to start
     call gMoveTo2D(0.0,3.0)
! Draw figure
     call gDrawPolylineTo2D(6,pt)
```



Multiple Line Drawn by gDrawPolylineTo2D()

The same figure could have been produced using the routine gDrawPolylineBy2D() as follows:

C code

```
static GPOINT pt[6] = {2.0,-2.0, 4.0,0.0, 2.0,2.0,
                       -8.0,0.0, 4.0,4.0, 0.0,-4.0};
      .

/* Move to start */
      gMoveTo2D(0.0,3.0);
/* Draw figure */
      gDrawPolylineBy2D(6,pt);
```

F90 code

```
type (GPOINT), dimension(6) :: pt = (/ &
      GPOINT{2.0,-2.0), GPOINT(4.0,0.0), GPOINT(2.0,2.0), &
      GPOINT(-8.0,0.0), GPOINT(4.0,4.0), GPOINT(0.0,-4.0} /)
      .
! Move to start
     call gMoveTo2D(0.0,3.0)
! Draw figure
     call gDrawPolylineBy2D(6,pt)
```

Polyline Sets

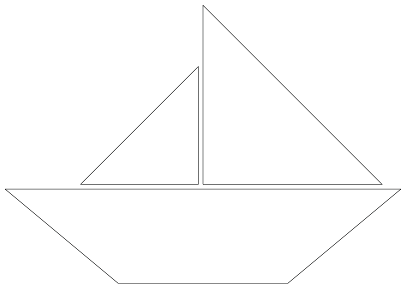
Polyline Set Definition

A polyline set consists of an array of polylines each of which consists of an integer number of vertices and a pointer to an array of 2D vertices.

Each polyline is complete within itself and does not make use of the current pen position. For this reason polygon sets can only use absolute coordinates.

An example of a 2-D polyline set consisting of a trapezium and two triangles is represented by the following coordinates and shown in the diagram below:

	1	2	3	4	5	6	7	8	9	10	11	12	13
x:	40.	160.	340.	460.	40.0	120.	245.	245.	120.	250.	440.	250.	250.
y:	140.	40.	40.	140.	140.	145.	270.	145.	145.	145.	145.	335.	145.
	<----------------------------------->					<-------------------------->				<-------------------------->			
Polyline sizes			5				4				4		



Polyline Set

Polyline Usage

Two dimensional polyline sets are drawn using the following routine.

gDrawPolylineSet2D(npol, polylines2)

where **npol** is the number of polylines contained in the GPOLYGON array **polylines2**.

The example polyline sets described previously can be implemented as follows.

C code

```
static GPOLYGON poly[3] = {5, 0, 4, 0, 4, 0};
static GPOINT points[13] = {
      40.0,140.0, 160.0,40.0, 340.0,40.0,
      460.0,140.0, 40.0,140.0,
      120.0,145.0, 245.0,270.0, 245.0,145.0, 120.0,145.0
      250.0,145.0, 440.0,145.0, 250.0,335.0, 250.0,145.0};
main()
{
   poly[0].verts=&points[0];
   poly[1].verts=&points[5];
   poly[2].verts=&points[9];

   gDrawPolylineSet2D(3,poly);
}
```

F90 code

```
type (GPOLYGON) :: poly(3)
type (GPOINT)   :: points(13) = (/ &
         GPOINT(40.0,140.0), GPOINT(160.0,40.0), &
         GPOINT(340.0,40.0), GPOINT(460.0,140.0), &
         GPOINT(40.0,140.0), &
         GPOINT(120.0,145.0), GPOINT(245.0,270.0), &
         GPOINT(245.0,145.0), GPOINT(120.0,145.0), &
         GPOINT(250.0,145.0), GPOINT(440.0,145.0), &
         GPOINT(250.0,335.0), GPOINT(250.0,145.0) /)

   poly(1)%nvert=5
   poly(1)%verts=>points(1:5)
   poly(2)%nvert=4
   poly(2)%verts=>points(6:9)
   poly(3)%nvert=4
   poly(3)%verts=>points(10:13)
   .
   gDrawPolylineSet2D(3,poly)
```

Circular Arcs

The routines for drawing circular arcs are:

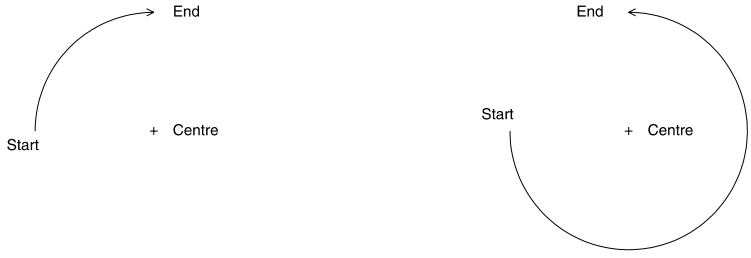
gDrawArcTo2D(xc, yc, xe, ye, sense)

gDrawArcBy2D(dxc, dyc, dxe, dye, sense)

All arcs are drawn from the start pen position. The radius of an arc is the distance from the start point to the centre. The end pen position or any point on the straight line from the centre through the end point of the arc may be specified. The end pen position will then be calculated.

Two-Dimensional Arcs

Specifying the start pen position, end pen position and centre enables two possible arcs to be drawn - the start and end points can be joined by either a clockwise or an anticlockwise movement. The direction is indicated by **sense**.

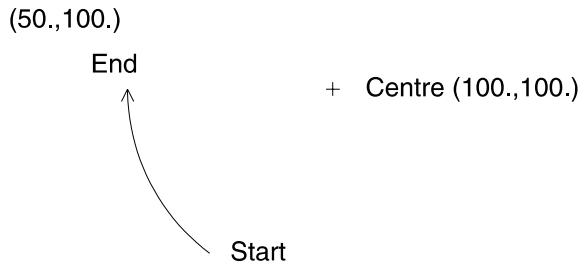


Clockwise/Anticlockwise

If the value of **sense** is GCLOCKWISE, then a clockwise arc is drawn, and if it is GANTICLOCKWISE, an anticlockwise arc is drawn.

Examples:

- To draw an arc centre (100.0,100.0) and end point (50.0,100.0):



Minor Chord

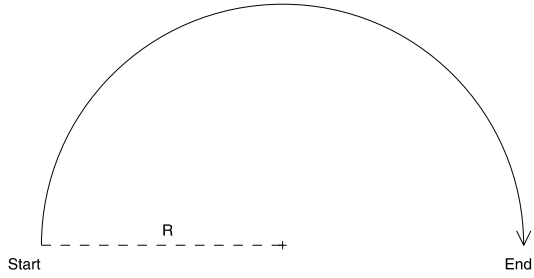
```
gMoveTo2D(70.0,60.0);
gDrawArcTo2D(100.0,100.0,
          50.0,100.0,GCLOCKWISE);
```

```
call gMoveTo2D(70.0,60.0)
call gDrawArcTo2D(100.0,100.0, &
          50.0,100.0,GCLOCKWISE)
```

- To draw a semicircle radius r:

```
gDrawArcBy2D(r,0.0,
             r+r,0.0,GCLOCKWISE);
```

```
call gDrawArcBy2D(r,0.0, &
             r+r,0.0,GCLOCKWISE)
```



Semicircular Arc

Drawing Circles

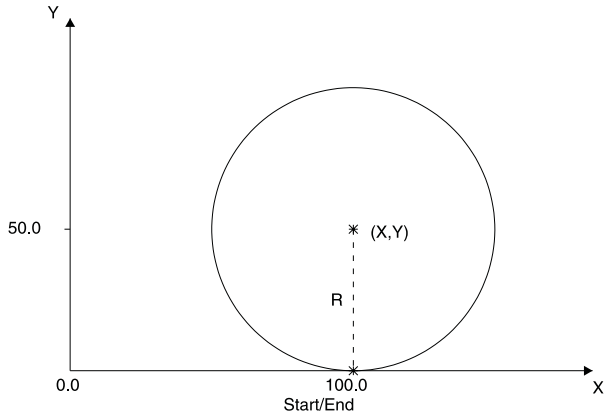
Circles can be drawn using the arc routines by specifying the end point of the arc as being the start pen position. The value of **sense** is immaterial.

Examples:

- To draw a circle from the point (100.0,0.0) with centre (100.0,50.0):

```
gMoveTo2D(100.0,0.0);
gDrawArcTo2D(100.0,50.0,
       100.0,0.0,GANTICLOCKWISE);
```

```
call gMoveTo2D(100.0,0.0)
call gDrawArcTo2D(100.0,50.0, &
       100.0,0.0,GANTICLOCKWISE)
```



Circular Arc

- To draw a circle centre (x,y) radius r:

```
/* Move to base of circle */
   gMoveTo2D(x,y-r);
/* Draw circle */
   gDrawArcBy2D(0.0,r,
      0.0,0.0,GANTICLOCKWISE);
```

```
! Move to base of circle
   call gMoveTo2D(x,y-r)
! Draw circle
   call gDrawArcBy2D(0.0,r, &
      0.0,0.0,GANTICLOCKWISE)
```

Hardware and Software Arcs

When using devices capable of drawing hardware arcs, software arcs may be selected by using the routine:

gSetArcMode(swi)

The argument switches hardware arcs on (**swi** = GHARD) and off (**swi** = GSOFT). Hardware arcs can be windowed and transformed and are subject to the current line mode, but remain unaffected by the control routines gSetArcIncrement() and gSetArcTolerance().

Software arcs should be selected for these routines to have effect.

Arc Control Routines

GINO arcs are drawn as a series of straight line chords. Enough chords are drawn to produce relatively smooth arcs. The number of line segments per arc can be controlled using one of the routines:

gSetArcIncrement(n)

gSetArcTolerance(tol)

Controlling the Number of Chords

The argument to routine gSetArcIncrement() specifies the number of straight line segments (or chords) per full circle for all subsequent ARC routines.

Examples:

- To specify that subsequent arcs are to be part of an eight sided polygon:

```
gSetArcIncrement(8);
```

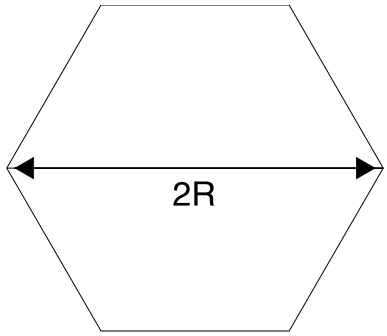
```
call gSetArcIncrement(8)
```

In this case, for example, subsequent semicircles would consist of four chords.

- To draw a hexagon centre (x,y) and with radius r:

```
gMoveTo2D(x-r, y);
gSetArcIncrement(6);
gDrawArcBy2D(r,0.0,0.0,0.0,
                GANTICLOCKWISE);
```

```
call gMoveTo2D(x-r, y)
call gSetArcIncrement(6)
call gDrawArcBy2D(r,0.0, &
       0.0,0.0,GANTICLOCKWISE)
```

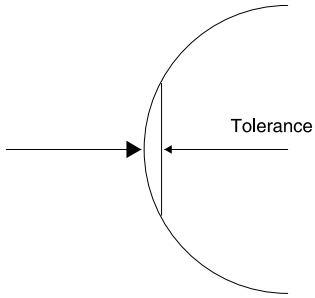


Arc Increments

Note that calling gSetArcIncrement() will disable hardware generated arcs.

Controlling Tolerance

The tolerance is the maximum distance allowed between the approximating chord and the true arc.



Arc Tolerance

The default distance is dependent on the output device. This can be changed by the use of gSetArcTolerance() to produce rougher or smoother arcs. The smoothness of the arc is ultimately dependent on the accuracy of the device. For example, to alter the tolerance to produce rough arcs, (i.e. set tolerance to 1mm) use:

```
gSetArcTolerance(1.0);
```

```
call gSetArcTolerance(1.0)
```

Notes:

(a) The default settings of tolerance and number of chords can be reset by calling gSetArcTolerance() or gSetArcIncrement() with zero arguments.

(b) The end position will always be on the circumference of the true arc.

(c) The most recently called of gSetArcTolerance() and gSetArcIncrement() dictates the appearance of the chords.

(d) If the user specifies a finer tolerance than is permitted by the resolution of the device, then gSetArcTolerance() reverts to half the minimum step size, that is to say the smoothest arc possible on that device.

Arc Settings

The current settings of the arc control parameters may be obtained using the routine:

gEnqArcState(swi, nincs, tol)

This returns the state of the hardware/software switch, the number of chords per full circle, and the tolerance. **nincs** is returned zero if gSetArcTolerance() was called more recently than gSetArcIncrement().

Use of Arc Routines

To draw an object a number of times, use a routine. By the use of routine arguments, the position, size and orientation of the object may be varied either locally from within the routine, or from the calling program.

For example, to draw a dumb-bell at position (100.0,100.0) width = 100.0, radius = 10.0, and length = 50.0:

```
dumb(100.,100.,10.,10.,50.);
```

```
call dumb(100.,100.,10.,10.,50.)
```

To draw a dumb-bell at position (100.0,130.0) of half the size:

```
dumb(100.,130.,5.,5.,25.);
```

```
call dumb(100.,130.,5.,5.,25.)
```

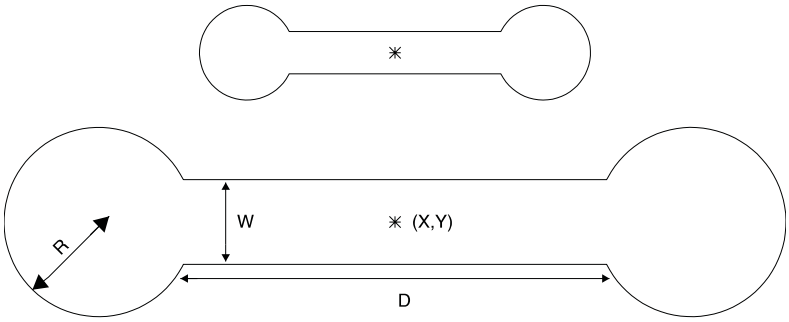
The following routine would draw the dumb-bell shown below:

C code

```
void dumb(float xc, float yc, float width, float radius,
          float length)
{
/* Move to absolute origin of object */
   gMoveTo2D(xc, yc);
/* Move to start of drawing */
   gMoveBy2D(-length/2.0,width/2.0);
/* Left-hand end */
   gDrawArcBy2D(-radius,-width/2.0,0.0,-width,1);
/* Bottom horizontal */
   gDrawLineBy2D(length,0.0);
/* Right-hand end */
   gDrawArcBy2D(radius,-width/2.0,0.0,-width,1);
/* Top horizontal */
   gDrawLineBy2D(-length,0.0);
}
```

F90 code

```
subroutine dumb(xc, yc, width, radius, length)
use gino_f90
real xc, yc, width, radius, length
! Move to absolute origin of object
   call gMoveTo2D(xc, yc)
! Move to start of drawing
   call gMoveBy2D(-length/2.0,width/2.0)
! Left-hand end
   call gDrawArcBy2D(-radius,-width/2.0,0.0,-width,1)
! Bottom horizontal
   call gDrawLineBy2D(length,0.0)
! Right-hand end
   call gDrawArcBy2D(radius,-width/2.0,0.0,-width,1)
! Top horizontal
   call gDrawLineBy2D(-length,0.0)
   return
```



Use of Arc Routines

Note the use of relative moves and lines within the routine.

Parametric Curves

A smooth curve can be drawn through a set of points using one of the routines:

gDrawAkimaTo2D(npts, points, beg, fin)

gDrawAkimaBy2D(npts, points, beg, fin)

gDrawCurveTo2D(npts, points, beg, fin)

gDrawCurveBy2D(npts, points, beg, fin)

Both sets of routines generate a piecewise parametric cubic curve drawn through each pair of points supplied either as absolute or relative coordinates. The first pair (gDrawAkimaTo2D()/ gDrawAkimaBy2D()) use an averaging method due to Akima, which produces a tighter curve, but can be less accurate for (single-valued) functional data. The second pair (gDrawCurveTo2D()/ gDrawCurveBy2D()) produce a looser curve, but is useful for contour drawing. Both sets are very accurate at drawing an approximation to a circle. A comparison of both methods can be seen in a later figure.

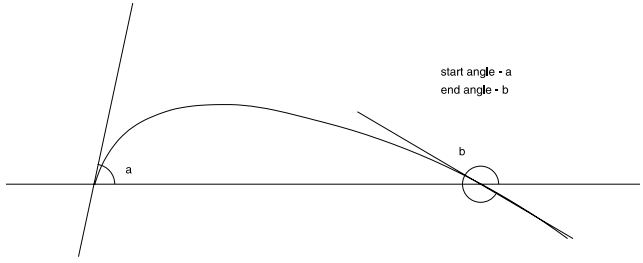
Curve End Conditions

By default, the end conditions of a curve are somewhat ill-defined as the undefined slopes may need to be set by additional data. If end point conditions are to be specified then **beg** and/or **fin** should be set to GXPOINT or GANGLE, and the end point information given through the routine gSetCurveAttribs2D():

gSetCurveAttribs2D(cbeg, sbeg, cfin, sfin, xbeg, ybeg, xfin, yfin)

Similarly, the end conditions following the drawing of a curve may be enquired by using the routine gEnqCurveAttribs2D(), which has the same arguments as gSetCurveAttribs2D(). The values returned in gEnqCurveAttribs2D() will be either those supplied by a previous call to gSetCurveAttribs2D() or those set by the curve drawing routines if no call to gSetCurveAttribs2D() has been made.

Both start and finish angles are made with the positive x-axis, see below.



Curve Drawing

The curve is drawn using straight line segments in the same way as software arcs are generated. As with arcs, the smoothness of the curve can be varied using GSetArcIncrement()/ gSetArcTolerance(). Due to the nature of the different algorithms GSetArcIncrement() controls the number of increments on the Akima curves and gSetArcTolerance() controls the tolerance of the standard curve.

In all the following examples of curve drawings, the same set of data points is used, and is set up as follows:

```
static GPOINT pt[6] =
   {3.0,3.0, 3.0,6.0,
    6.0,6.0, 4.5,4.5,
    7.0,2.0, 12.0,4.0};
```

```
type (GPOINT) :: pt(6) = (/&
GPOINT{3.,3.),GPOINT{3.,6.), &
GPOINT{6.,6.),GPOINT{4.5,4.5), &
GPOINT{7.,2.),GPOINT{12.,4.} /)
```

The data points are shown on the curves as asterisks. This is done by using the following call in each case:

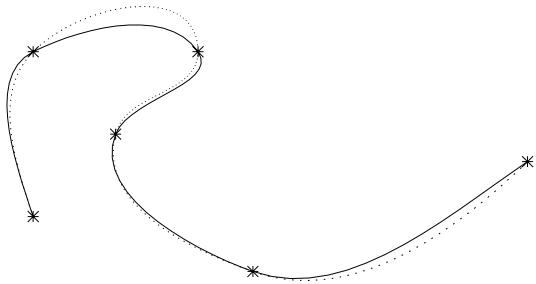
```
gDrawPolymarkerTo2D(6,pt,8);
```

```
call gDrawPolymarkerTo2D(6,pt,8)
```

Specifying No End Conditions

```
gDrawAkimaTo2D(6,pt,
                    GNONE,GNONE);
gDrawCurveTo2D(6,pt,
                    GNONE,GNONE);
```

```
call gDrawAkimaTo2D(6,pt, &
                        GNONE,GNONE)
call gDrawCurveTo2D(6,pt, &
                        GNONE,GNONE)
```

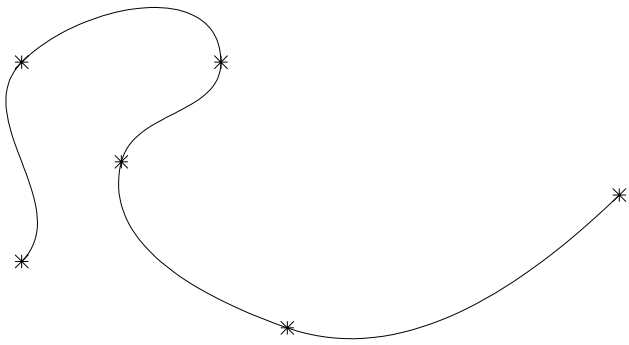


Akima and Curve (dashed) Comparison

Specifying Beginning Conditions as COS and SIN

```
alpha = 45.0 *3.142/180.0;
cosbeg = cos(alpha);
sinbeg = sin(alpha);
gSetCurveAttribs(cosbeg,sinbeg,
    0.0,0.0,0.0,0.0,0.0,0.0);
gDrawCurveTo2D(6,pt,
    GANGLE,GNONE);
```

```
alpha = 45.0 *3.142/180.0
cosbeg = cos(alpha)
sinbeg = sin(alpha)
call gSetCurveAttribs(cosbeg, &
  sinbeg,0.0,0.0,0.0,0.0,0.0,0.0)
call gDrawCurveTo2D(6,pt, &
  GANGLE,GNONE)
```

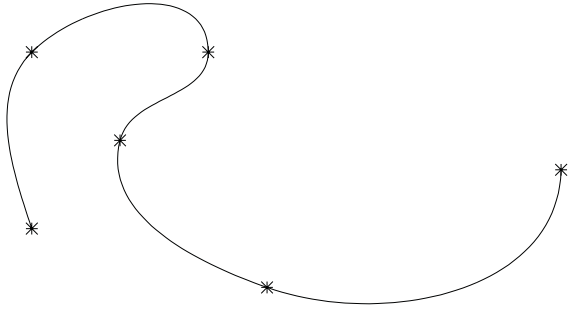


Curve with angular start conditions

Specifying End Conditions for Curve Finish Using an Extra Point

```
xf = 9.0;
yf = 7.0;
gSetCurveAttribs(0.0,0.0,
    0.0,0.0,0.0,0.0,xf,yf);
gDrawCurveTo2D(6,pt,
    GNONE,GXPOINT);
```

```
xf = 9.0
yf = 7.0
call gSetCurveAttribs(0.0,0.0, &
     0.0,0.0,0.0,0.0,xf,yf)
call gDrawCurveTo2D(6,pt, &
     GNONE,GXPOINT)
```

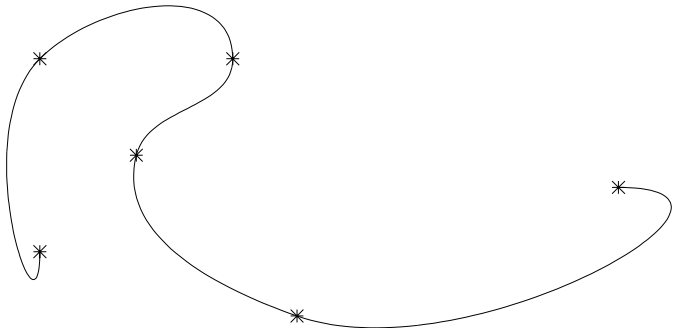


Extra point at end of curve

Specifying Both End Conditions

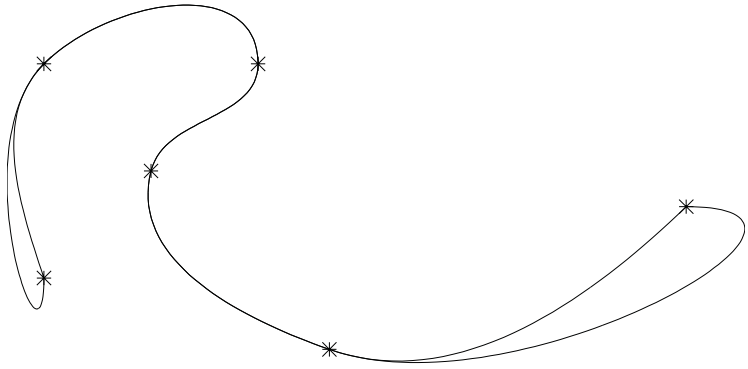
```
/*  ANGLE = 180 degrees */
alpha = 3.142;
cosfin = cos(alpha);
sinfin = sin(alpha);
xb = 3.0;
yb = 4.0;
gSetCurveAttribs(0.0,0.0,
    cosfin,sinfin,xb,yb,0.0,0.0);
gDrawCurveTo2D(6,pt,
    GXPOINT,GANGLE);
```

```
!  ANGLE = 180 degrees
alpha = 3.142
cosfin = cos(alpha)
sinfin = sin(alpha)
xb = 3.0
yb = 4.0
call gSetCurveAttribs(0.0,0.0, &
    cosfin,sinfin,xb,yb,0.0,0.0)
call gDrawCurveTo2D(6,pt, &
    GXPOINT,GANGLE)
```



Example with both ends specified

Highlighting the Effects of Different Values of IBEG and IFIN



Curve with no and both ends specified superimposed

Note that gDrawCurveBy2D() uses the values in the points array as relative points, so data points in the previous example would become:

```
static GPOINT pt[6] = {
   0.0,0.0,   0.0,3.0,
   3.0,0.0,  -1.5,-1.5,
   2.5,-2.5,  5.0,2.0};
```

```
type (GPOINT) :: pt(6) = (/&
GPOINT(0.,0.),GPOINT(0., 3.),&
GPOINT{3.,0.),GPOINT(-1.5,-1.5),&
GPOINT(2.5,-2.5),GPOINT(5.,2.} /)
```

Note also that gDrawCurveTo2D() does an absolute move to the first X,Y coordinate pair, whereas gDrawCurveBy2D() uses the current position as the first X,Y coordinate pair.

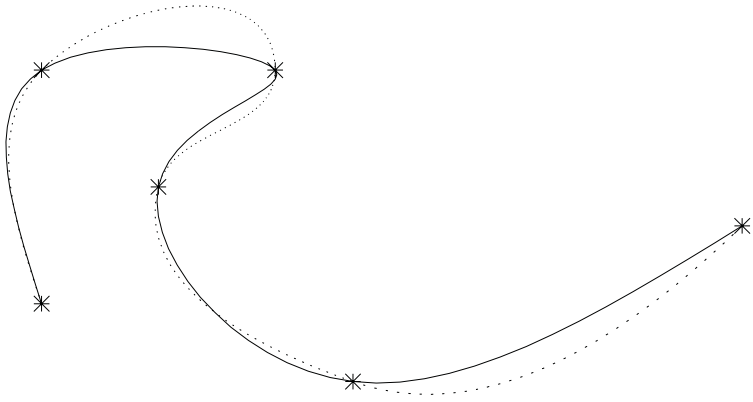
Spline Curves

In addition to the piecewise cubic curve drawing routines, GINO provides two routines to draw a smooth curve through a series of points using cubic splines:

gDrawSplineTo2D(npts, points2, beg, fin)

gDrawSplineBy2D(npts, points2, beg, fin)

In comparison to the piecewise cubic curve routines, the spline curves are generally tighter than the gDrawCurveTo2D()/gDrawCurveBy2D() set and are more accurate at fitting functional data than the gDrawAkimaTo2D()/ gDrawAkimaBy2D() set. For example, using the same six points set up in the previous example, gDrawCurveTo2D() (in dotted line) and gDrawSplineTo2D() (in solid line) give the following output with no end conditions set:



Comparison of spline with curve output

The routine gSetArcIncrement() controls the number of increments between each supplied data point.

Spline Curve End Conditions

End conditions can also be set for spline curves in the same way as for piecewise cubic curves, but it is usually necessary to set angular end conditions using scaled derivatives instead of simply sines and cosines.

For monotonically increasing data in either X or Y or Z, these can easily be calculated, or it is sufficient to set the X slope to 1.0 and the Y slope to y'(x) and the drawing routines will compute the correct value. For parametric data, estimates for the actual gradient (x'(t), y'(t)) are really required. End conditions can alternatively be set using an extra point in 2 or 3 dimensions in the same manner as the previous curve routines.

The routines gSetCurveAttribs2D()/gEnqCurveAttribs2D() are used to set and enquire the end conditions for 2D spline curves, and an additional pair of routines are used for the 3D equivalent function. Thus:

gSetCurveAttribs2D(dxbeg,dybeg,dxfin,dyfin,xbeg,ybeg,xfin,yfin)

gEnqCurveAttribs2D(dxbeg,dybeg,dxfin,dyfin,xbeg,ybeg,xfin,yfin)

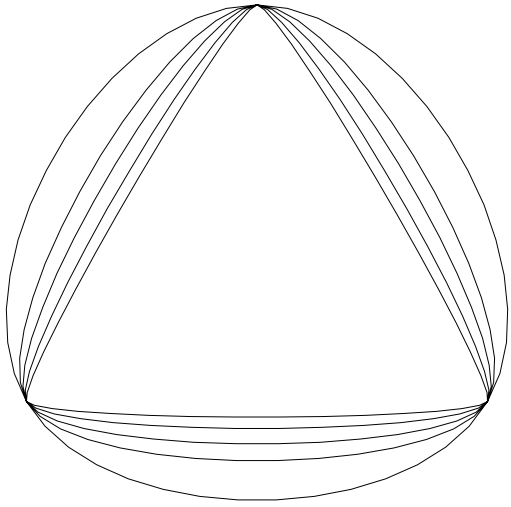
Spline Curve Tension Control

An alternative approach to smoothing is to use a spline in tension. The routines gSetSplineTension() and gEnqSplineTension() are used to set and enquire the current spline tension value which has a default of 0.0.

gSetSplineTension(ten)

gEnqSplineTension(ten)

As the value of **ten** increases, the curve moves closer to a polyline representing the supplied data points, with ultimate loss of smoothness. Values in the range 0 to 10 give reasonable results and it is also possible to use negative values down to -2 to give a more rounded shape to the curve. The curves in the diagram below show the effect of applying tension to a circle generated from 3 data points.

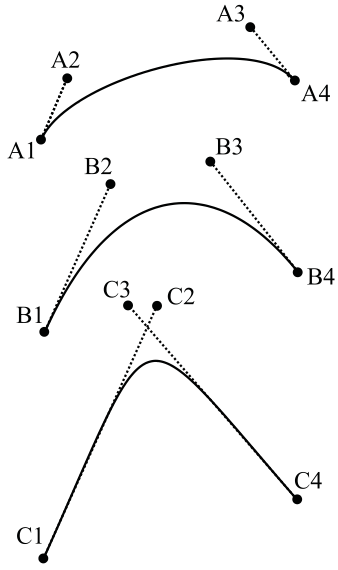


Spline Curve Tension

Bezier Curves

The Bezier curve routines offer a very different kind of curve control where the data supplied represents control points rather than points on the curve itself.

This can be illustrated by looking at a simple Bezier curve of degree four (cubic). As can be seen in the figure below, for each of the three Bezier curves, the points 2 & 3 deviate the line between the end points 1 and 4. Looking at the vectors between these points, the greater the magnitude, the bigger the deviation whilst their direction determines the tangent of the curve at the end points. Therefore, with the three cases below, as all the vectors between the end points and their adjacent control points are the same, the tangents at points A_1, B_1 and C_1 are the same as are the tangents at the points A_4, B_4 and C_4. This feature of Bezier curves is important when considering the joining of two such curves (see below).



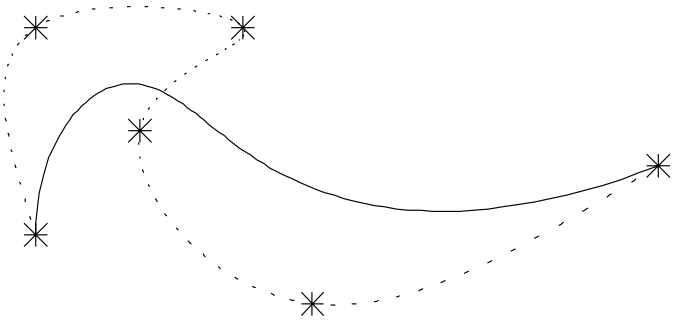
Two routines are provided to generate this type of curve in 2D:

gDrawBezierTo2D(npts, points2)

gDrawBezierBy2D(npts, points2)

As with the previous curve drawing routines, **points2** is an array of type GPOINT and **npts** are the number of control points stored in the array.

Supplying a set of 6 control points (displayed as asterisks) the following curve will be drawn. Note that the curve always starts at the first control point and ends at the last control point, but in all probability will not pass through any other supplied control point.

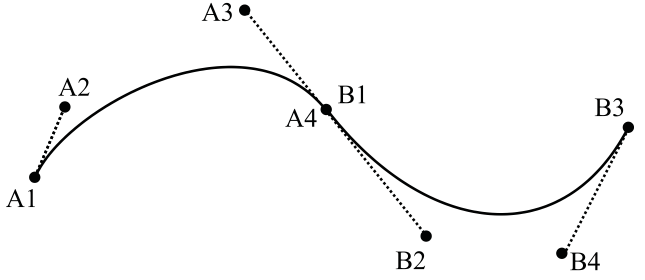


Bezier Curve (with Spline comparison)

End Conditions

There is no direct control over end conditions with Bezier curves, but the fact that the position of the control point adjacent to the first (or last) data point determines the tangent of the curve at that point (see above), curves can be smoothly joined by taking account of this feature.

Therefore to join two Bezier curves, the joining point must obviously be at the same coordinate position, and the distance and angle between the adjacent control points must also be the same, such that these two control points lie in a straight line. Thus in the figure below, points A_4 and B_1 must be the same and points A_3 and B_2 must form a line that intersects A_4 and B_1. This will give a seamless join and forms a piecewise Bezier curve.



Joining two Bezier Curves

Elevation and Reduction

Bezier curve drawing routines are most useful for interactive curve design where the control points can be manipulated to produce the desired shape of curve. Tighter curves can be produced where a greater number of control points are clustered. To assist in the design of appropriate curves, two auxiliary routines are also provided to increase and/or reduce the number of control points in a Bezier curve definition.

gElevateBezier2D(npts, points2)

gReduceBezier2D(npts, points2)

Both routines only operate on absolute coordinates and the first of them takes a set of control points as input and returns a new set with one more control point than was input, but which represents the same curve. Users must ensure that the **points** array is large enough to hold the additional point on return.

The second routine takes a set of control points and returns a new set with one less control point. Users should be aware that reducing the number of control points does not guarantee that the shape of the curve is maintained, but it will always be a close approximation.

Point Storage

With the drawing of arcs and curves, GINO generates many internal points in addition to those directly specified by the calling routine. There are cases where an application needs to know the location of these points in order to define a polygon boundary or carry out other graphical or mathematical manipulation on them.

GINO provides facilities for both these options by storing all the points between which vectors are drawn (whether it be for straight lines or the small vectors that make up arcs and curves) in an internal storage buffer. Depending on the desired use of the internal points, up to two separate buffer areas can be defined, one for polygon definition and one for simple point storage. In both cases the buffers are contained in the global GINO workspace area which should be defined at the start of an application through the routine gSetWorkspaceLimit() (see page 33).

Once the global workspace has been defined, the two possible storage areas are defined using one or both of the following routines:

gDefinePolygonWorkspace(nw)

gDefinePointWorkspace(nw)

where **nw** is the number of real words required for the appropriate buffer.

The routine gDefinePolygonWorkspace() is used to define polygon workspace and the subsequent storage of points for the definition of polygons is fully described later in this document (see page 245).

The routine gDefinePointWorkspace() defines a buffer for the simple storage of points that can be returned to the application. Each point occupies 4 words of storage, so allocating a workspace of 120 words will allow for the storage of 30 points. Note that the actual number of points generated by an arc or curve will depend on tolerance or tension of the particular arc or curve.

Point storage is started, restarted or paused using the routine:

gSetPointMode(switch)

and enquired using:

gEnqPointMode(switch)

where **switch** can be GOFF, GSPACE, GPICTURE or GRESTART. When the point storage mode is set to GSPACE or GPICTURE, points are stored as either untransformed (i.e. as supplied by the user) or transformed (i.e. as they appear on the drawing area with respect to the current viewport) coordinates respectively. Storage is switched off using the GOFF setting and restarted (in the current mode) when GSPACE or GPICTURE is used again.

If the point storage mode is changed from GSPACE to GPICTURE or vice-versa or GCLEAR is used in gSetPointMode(), all previously stored points are thrown away.

The stored points are returned to the user through the function:

nret=gReturnInternalPoints2D(nn, points2, np, polylines2, npts, npol)

where **points2** is an array of type GPOINT and **polylines2** is an array of type GPOLYGON. The arguments **nn** and **np** should be set to the size of these arrays. The arguments **npts** and **npol** return the number of points and polylines that actually exist in the internal workspace which may be more than those returned if the supplied arrays are not sufficiently large enough. The function itself returns the actual number of complete polylines that have been placed in the user supplied arrays.

In order to enquire how much data has been stored, gReturnInternalPoints2D() can be called with **nn** and **np** set to 1, in which case the total space for all the points and polylines can be allocated using the values of **npts** and **npol**. The function can then be called a second time to return all the stored data.

The routine returns the stored points both as a set of vertices in the **points2** array and a set of polylines in the **polylines2** array. This in fact represents the same data but in two different formats with the latter being a more accurate definition of the information stored. Note that the GPOLYGON structure contains pointers into the GPOINT array and that any 3rd dimension (Z coordinate) is ignored in the call to gReturnInternalPoints2D().

The following example shows a usage of the gReturnInternalPoints2D() routine:

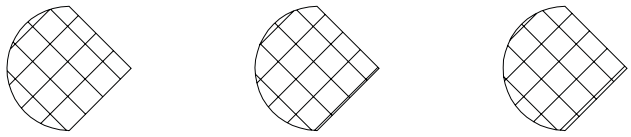
C Code

```
#include <qino-c.h>
#define NN 300
#define NP 4
GPOINT pts[NN];
GPOLYGON pol[NP];

#if defined(MWIN) || defined(WOGL)
int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpszCmdParam, int nCmdShow)
#else
int main ()
#endif
{
   int nret,npts,npol;

      qOpenGino();
      qMwin(hInstance,hPrevInstance);
/* Define global and point workspace */
      qSetWorkspaceLimit(2000);
      qDefinePointWorkspace(1000);
/* Set point storage mode */
      qSetPointMode(GSPACE);
/* Draw some graphics */
      qMoveTo2D(10.0,10.0);
      qDrawLineBy2D(5.0,5.0);
      qDrawLineBy2D(-5.0,5.0);
      qDrawArcBy2D(0.0,-5.0,0.0,-10.0,GANTICLOCKWISE);
      qMoveTo2D(0.0,10.0);
      qMoveTo2D(30.0,10.0);
      qDrawLineBy2D(5.0,5.0);
```

```
        qDrawLineBy2D(-5.0,5.0);
        qDrawArcBy2D(0.0,-5.0,0.0,-10.0,GANTICLOCKWISE);
        qMoveTo2D(40.0,10.0);
        qMoveTo2D(50.0,10.0);
        qDrawLineBy2D(5.0,5.0);
        qDrawLineBy2D(-5.0,5.0);
        qDrawArcBy2D(0.0,-5.0,0.0,-10.0,GANTICLOCKWISE);
/* Switch point storage off */
        qSetPointMode(GOFF);
/* Return internal storage */
        nret=qReturnInternalPoints2D(NN,pts,NP,pol,&npts,&npol);
/* Fill polygon set */
        qFillPolygonSet2D(6,1,GAREA,nret,pol);
        qSuspendDevice();
        qCloseGino();
}
```

F90 Code

```
use qino_f90
parameter (nn=300, np=4)
type (GPOINT) :: pts(nn)
type (GPOLYGON) :: pol(np)
      call qOpenGino
      call xxxxx
! Define global and point workspace
      call qSetWorkspaceLimit(1,2000)
      call gDefinePointWorkspace(1000)
! Set point storage mode
      call qSetPointMode(GSPACE)
! Draw some graphics
      call qMoveTo2D(10.0,10.0)
      call qDrawLineBy2D(5.0,5.0)
      call gDrawLineBy2D(-5.0,5.0)
      call qDrawArcBy2D(0.0,-5.0,0.0,-10.0,GANTICLOCKWISE)
      call qMoveTo2D(0.0,10.0)
      call gMoveTo2D(30.0,10.0)
      call qDrawLineBy2D(5.0,5.0)
      call qDrawLineBy2D(-5.0,5.0)
      call gDrawArcBy2D(0.0,-5.0,0.0,-10.0,GANTICLOCKWISE)
      call qMoveTo2D(40.0,10.0)
      call qMoveTo2D(50.0,10.0)
      call gDrawLineBy2D(5.0,5.0)
      call qDrawLineBy2D(-5.0,5.0)
      call qDrawArcBy2D(0.0,-5.0,0.0,-10.0,GANTICLOCKWISE)
! Switch point storage off
      call qSetPointMode(GOFF)
! Return internal storage
      nret=gReturnInternalPoints2D(nn,pts,np,pol,npts,npol)
! Fill polygon set
      call qFillPolygonSet2D(6,1,GAREA,nret,pol)
      call gSuspendDevice
      call qCloseGino
      stop
      end
```



Point Storage

The outline of each object is drawn with lines and arcs, then the points of each polygon is retrieved with gReturnInternalPoints2D() and the polygon set is filled with gFillPolygonSet2D().

2D Interpolation

GINO provides a facility to interpolate user suppled data or from previously drawn curves, lines or arcs using the above point storage mechanism. Passing a single data value with a set of 2D data points, the function gInterpolateData2D() can return all the intersections of the two using linear interpolation.

The function has the following form:

nint=gInterpolateData2D(nopt, ptint, npts, points2, nptout, ptout1)

where **nopt** can be GXDATA or GYDATA indicating the interpretation of the argument **ptint**, the value to be interpreted. The argument **npts** specifies the number of 2D data points supplied in the array **points2** (which is of type GPOINT) and **nptout** is the size of the output array **ptout1**.

The function returns the number of intersection points returned in the array **ptout1**. Where **nopt**=GXDATA this array will contain Y values and where **nopt**=GYDATA this array will contain X values. There may be zero, one or more than one depending on the form of the data, but it will never exceed **nptout** even though there may be more intersections possible from the supplied data.

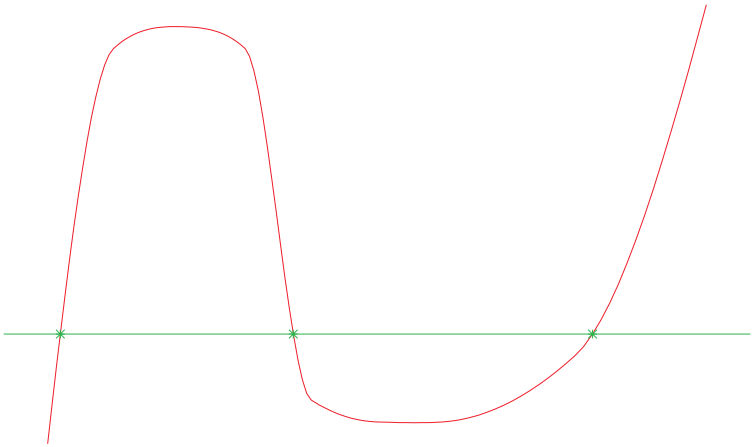
The following example shows the interpolation of a 2D curve:

C Code

```
#include <gino-c.h>
#define NP 9
#define NVERT 1000
#define NPOLY 2
#define NPTOUT 4
GPOLYGON poly[NPOLY];
GPOINT   points[NVERT];
GPOINT   data[NP] = { 20.0, 20.0,  35.0,110.0,  50.0,115.0,
                      65.0,110.0,  80.0, 30.0,  95.0, 25.0,
                     110.0, 25.0, 140.0, 40.0, 170.0,120.0 };
float xmin = 10.0;
float xmax = 180.0;
float ptout[NPTOUT];

#if defined(MWIN) || defined(WOGL)
int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpszCmdParam, int nCmdShow)
#else
int main ()
#endif
{
   float ptint;
   int npt,npol,n,ninter,ntemp;

/*  Initialise GINO and point storage */

   gOpenGino();
   xxxxx();
   gSetWorkspaceLimit(2000);
   gDefinePointWorkspace(1000);

/*  Draw curve and store points internally */

   gMoveTo2D(data[0].x,data[0].y);
   gSetLineColour(GRED);
   gSetPointMode(GSPACE);
   gDrawAkimaTo2D(NP,data,GNONE,GNONE);
   gSetPointMode(GOFF);
/*  Retrieve stored curve points */

   n=gReturnInternalPoints2D(NVERT,points,NPOLY,poly,&npt,&npol);

/*  Calculate intersections through interpolation */

   ptint=45.0;
   ninter=gInterpolateData2D(GYDATA,ptint,npt,points,NPTOUT,ptout);


/*  Show intersection points on the curve */

   gSetLineColour(GGREEN);
   gMoveTo2D(xmin,ptint);
   gDrawLineTo2D(xmax,ptint);
   for (ntemp=0;ntemp<ninter;ntemp++) {
      gMoveTo2D(ptout[ntemp],ptint);
      gDrawMarker(GSTAR);
   }

   gSuspendDevice();
   gCloseGino();
}
```

F90 Code

```
use gino_f90
parameter (NP=9,NVERT=1000,NPOLY=2,NPTOUT=4)
type (GPOLYGON) poly(NPOLY)
type (GPOINT) points(NVERT)
type (GPOINT)  :: data(NP) = (/ &
   GPOINT( 20.0, 20.0), GPOINT( 35.0,110.0), GPOINT( 50.0,115.0), &
   GPOINT( 65.0,110.0), GPOINT( 80.0, 30.0), GPOINT( 95.0, 25.0), &
   GPOINT(110.0, 25.0), GPOINT(140.0, 40.0), GPOINT(170.0,120.0) /)
real :: xmin = 10.0
real :: xmax = 180.0
real ptout(NPTOUT)
!
!  Initialise GINO and point storage
!
   call gOpenGino
   call xxxxx
   call gSetWorkspaceLimit(1,2000)
   call gDefinePointWorkspace(1000)
!
!  Draw curve and store points internally
!
   call gMoveTo2D(data(1)%x,data(1)%y)
   call gSetLineColour(GRED)
   call gSetPointMode(GSPACE)
   call gDrawAkimaTo2D(NP,data,GNONE,GNONE)
   call gSetPointMode(GOFF)
!
!  Retrieve stored curve points
!
   n=gReturnInternalPoints2D(NVERT,points,NPOLY,poly,npt,npol)
!
!  Calculate intersections through interpolation
!
   ptint=45.0
   ninter=gInterpolateData2D(GYDATA,ptint,npt,points,NPTOUT,ptout)
!
!  Show intersection points on the curve
!
   call gSetLineColour(GGREEN)
   call gMoveTo2D(xmin,ptint)
   call gDrawLineTo2D(xmax,ptint)
   do ntemp = 1, ninter
      call gMoveTo2D(ptout(ntemp),ptint)
      call gDrawMarker(GSTAR)
   end do
!
   call gSuspendDevice
   call gCloseGino
stop
end
```



Data Interpolation

Chapter 5

LINE ATTRIBUTES

Line Attributes Introduction

The appearance or style of a line is described in GINO by six attributes:

- Visibility
- Broken line type
- Colour
- Width
- Pen type
- Line end type

Together they define what is termed a line style. Lines are drawn subject to the current line style, which may be varied by any of a number of GINO routines. Line styles may be stored in a table for use later.

Routines Described in this Chapter

The routines described in this section are:

(a) For control of individual attributes of the current line

Line visibility	gSetLineVis()	gEnqLineVis()
Broken line type	gSetBrokenLine()	gEnqBrokenLine()
Colour	gSetLineColour()	gEnqLineColour()
Thick line generation	gSetLineWidthMode()	gEnqLineWidthMode()
Line width	gSetLineWidth()	gEnqLineWidth()

Line width scale factor	gSetLineWidthScaling()	gEnqLineWidthScaling()
Pen type	gSetPenType()	gEnqPenType()
Line end type	gSetLineEnd()	gEnqLineEnd()
Pen attributes		gEnqSelectedPen()
Broken line mode	gSetBrokenLineMode()	

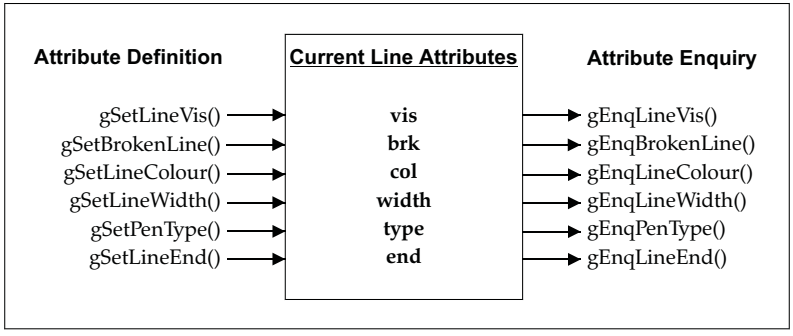
(b) Specification of attributes in the definition tables

Broken line type table	gDefineBrokenLineStyle()	gEnqBrokenLineStyle()
Line definition table	gDefineLineStyle()	gEnqLineStyle()
Line style selection	gSetLineStyle()	
Save current line style	gSaveLineStyle()	

For colour table definition see page 205.

Current Line Definition and Enquiry

The relationship of the current line attributes and the individual routines controlling them are as follows:



GINO uses the absolute value of the parameter given to define a current attribute, but GINO will issue a warning message if this value is negative. If a value falls outside its valid range or the request cannot be met, a default value will be provided.

In the case of line colour, width and type, the enquiry routines which complement each of the single attribute specifying routines return the attribute values specified by the user. These may not be the same as those actually implemented by an output device. The actual values of the hardware implementation are returned by gEnqSelectedPen().

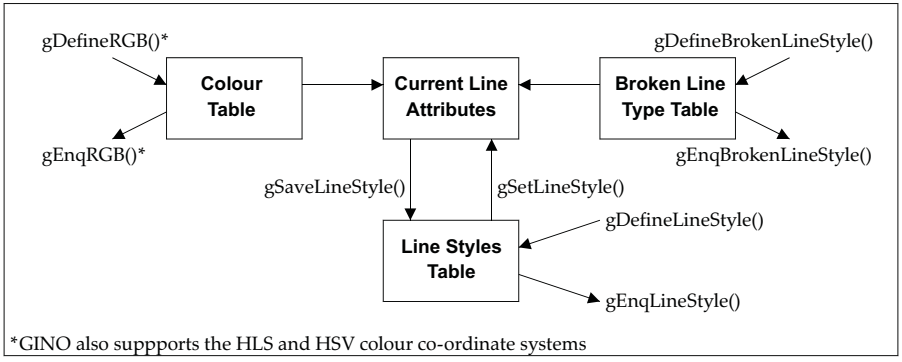
gEnqSelectedPen(col, width, type)

Note that in the case of direct colour devices, the return value of **col** will contain a 24bit RGB triplet irrespective of whether the line colour was selected using a colour identifier, a colour table index or a 24bit RGB triplet.

Drawing Attribute Tables

Tables are used to store specifications for colour values, broken line types and line styles. Definitions selected from these tables change the current line attributes.

The tables relate to the current line attributes in the following way:



The colour and broken line type tables contain definitions of their respective attributes. These definitions are implemented when selected by gSetLineColour() or gSetBrokenLine(), or when a complete line style is selected from the table of line definitions.

The line definitions table is different from the other two in that each entry defines a complete set of line attributes. This definition includes the identifiers which point to entries in the colour and broken line type tables.

Attribute definitions for colour and broken line type may be made and stored in their respective tables without affecting the current line. The exception is when the table entry to be changed was the one used to implement the current attribute value. In this case a change to the entry causes a change to the attribute. This effect happens only with the colour and broken line type tables.

Whenever a device is nominated these tables are initialized to a set of default values.

Individual Attributes

Changing Individual Attributes of the Current Line

Each current line attribute may be changed independently of the others. In the following example, the attributes are modified one by one and after each modification a line is drawn.

Drawing starts in the centre and the resulting output is shown below.

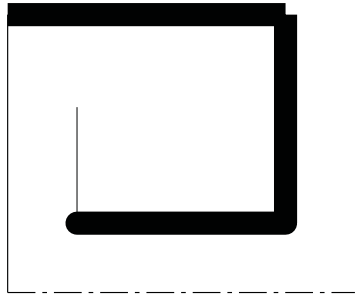
C code

```
/*  draw a line with the default attributes */
      gMoveTo2D(50.0,130.0);
      gDrawLineBy2D(0.0,-50.0);
/*  specify a line of 10mm wide with round ends*/
      gSetLineWidth(10.0);
      gSetLineEnd(GROUND);
      gDrawLineBy2D(90.0,0.0);
/*  change line ends to no ends */
      gSetLineEnd(GNONE);
      gDrawLineBy2D(0.0,90.0);
/*  change pen type */
      gSetPenType(3);
      gDrawLineBy2D(-120.0,0.0);
/*  change line width */
      gSetLineWidth(0.5);
      gDrawLineBy2D(0.0,-120.0);
/*  change broken line type */
      gSetBrokenLine(GSHORTCHAINED);
      gDrawLineBy2D(150.0,0.0);
/*  switch visibility off */
      gSetLineVis(GOFF);
      gDrawLineBy2D(0.0,150.0);
/*  switch visibility on */
      gSetLineVis(GON);
      gSetBrokenLine(GSOLID);
      gDrawLineBy2D(-150.0,0.0);
```

F90 code

```
!  draw a line with the default attributes
      call gMoveTo2D(50.0,130.0)
      call gDrawLineBy2D(0.0,-50.0)
!  specify a line of 10mm wide with round ends
      call gSetLineWidth(10.0)
      call gSetLineEnd(GROUND)
      call gDrawLineBy2D(90.0,0.0)
!  change line ends to no ends
      call gSetLineEnd(GNONE)
      call gDrawLineBy2D(0.0,90.0)
!  change pen type
      call gSetPenType(3)
      call gDrawLineBy2D(-120.0,0.0)
!  change line width
      call gSetLineWidth(0.5)
      call gDrawLineBy2D(0.0,-120.0)
!  change broken line type
      call gSetBrokenLine(GSHORTCHAINED)
      call gDrawLineBy2D(150.0,0.0)
!  switch visibility off
      call gSetLineVis(GOFF)
      call gDrawLineBy2D(0.0,150.0)
!  switch visibility on
      call gSetLineVis(GON)
      call gSetBrokenLine(GSOLID)
      call gDrawLineBy2D(-150.0,0.0)
```

The first gDrawLineBy2D() of this example draws a line with default attributes. The default attributes depend on the output device but they generally produce a line which is visible and solid. Colour, width and pen type defaults depend on the output device. For a colour raster device the line would typically be white on a black background and one scan line wide. The default line end type is no ends (line ends are shapes added to each end of the line).



Modification of line attributes

Each single-attribute controlling routine has a corresponding enquiry routine which returns the current value of the attribute as it was specified by the user. Attributes which have not been changed since GINO was called return their default values.

In the following pages each attribute is considered individually.

Line Visibility

gSetLineVis(vis)

gEnqLineVis(vis)

The current line may be drawn visible (default) or invisible. The routine gSetLineVis() switches between these two states. For example:

```
gSetLineVis(GOFF);
```

```
call gSetLineVis(GOFF)
```

switches line visibility off. Lines drawn subsequently will not appear on the output device.

The statement:

```
gSetLineVis(GON);
```

```
call gSetLineVis(GON)
```

will restore line visibility. The routine gSetLineVis() has no effect on any of the invisible drawing routines (e.g.gMoveBy2D(), gMoveTo3D()); that is visibility cannot be switched on for them.

The routine gEnqLineVis() returns the current state of line visibility.

Broken Line Type

gSetBrokenLine(brk)

gEnqBrokenLine(brk)

gSetBrokenLineMode(swi)

The broken pattern of the current line can be varied. The default pattern is a solid line. Routine gSetBrokenLine() is used to select a line type definition from a table of broken line types.

For example, the statement:

```
gSetBrokenLine(GLONGDASHED);
```

```
call gSetBrokenLine(GLONGDASHED)
```

makes broken line type ‘longdashed’ the current line type.

The setting of gSetBrokenLineMode() determines whether the line type selection is implemented from the software table (see gDefineBrokenLineStyle()) or from an output device’s hardware table (assuming it has one).

The default setting of gSetBrokenLineMode() (i.e. **swi**=GHARD or 0) gives hardware-implemented broken line types. The broken line types implemented from hardware tables are device dependent. When **swi** is set to GSOFT or 1, broken line types will be implemented from the software table. Up to 256 definitions (**brk** = 1 to 256) may be stored in this table.

If a device is incapable of generating its own broken lines, GINO will use software.

The result of calling gSetBrokenLine(), with **brk** greater than 256, depends entirely on the output device. If the hardware supports more than 256 definitions one of these may be selected. Otherwise line type defaults to a solid line. The identifier, **brk**, of the currently selected broken line type may be found by a call:

```
gEnqBrokenLine(brk);
```

```
call gEnqBrokenLine(brk)
```

Line Colour

The current line colour may be set or enquired using the following routines:

gSetLineColour(col)

gEnqLineColour(col)

where **col** is a colour identifier. This may be a pen number, colour number or index into a colour table depending on the current device (see page 46). In all cases however, when a device is first initialised the following set of colours are made available if possible:

Colour Identifier (col):	Colour Constant:	Colour:
0	GBACKGROUND	Background (device dependent)

1	GBLACK	Black
2	GRED	Red
3	GORANGE	Orange
4	GYELLOW	Yellow
5	GGREEN	Green
6	GCYAN	Cyan
7	GBLUE	Blue
8	GMAGENTA	Magenta
9	GBROWN	Brown
10	GWHITE	White

Thus, the statement:

```
gSetLineColour(3);
```

```
call gSetLineColour(3)
```

would select orange as the colour for the current line. Alternatively, the predefined constant GORANGE can be used to select colour 3. Colour identifier 0 or GBACKGROUND selects the background colour, assuming the device recognizes such a thing. This may be used as an erase facility by selecting the background colour gSetLineColour(0), and then drawing over a previously drawn line.

The actual colour displayed depends on the output device. Some devices (e.g. Plotters and monochrome displays) may support fewer colours than the standard GINO set in which case the selection of a colour identifier that is not available will default to some default colour (usually black). The range of colour identifiers and colour capabilities of the currently nominated device can be enquired by using the routine gEnqColourInfo() (see page 46).

On most devices the colour identifier is actually an index into a colour table which has been initialized to the above settings. Again, depending on the colour capabilities of the current device, these entries may be changed at any time within an application or new entries set to the range of colours required (see page 205).

On direct colour devices, the current line drawing colour may also be set in terms of a 24bit RGB triplet using the gTrueCol() function (see page 217).

The routine gEnqLineColour() returns the last requested colour identifier.

Line Width

By default, thick lines are generated using a mixture of hardware and software facilities depending on the capabilities of the device and the type of thick line being generated. In this mode, thick, broken lines with non-standard line ends are generated by software emulation to ensure the correct output. However, the user may opt to force either hardware of software generation to improve performance or to ensure complete accuracy, using the routine:

gSetLineWidthMode(swi)

where **swi** can be GHARDWARE, GMIXWARE (the default) or GSOFTWARE. Selecting hardware generation where no such capability exists will result in single stroke lines for all thicknesses of line. The software emulation method is determined by the device driver writer and is set according to the characteristics of the device; ie. for pen plotters, parallel lines will be used to build up the thick line, for raster devices, multiple horizontal/vertical lines are used or polygon fill may be selected where available. The user can enquire the current line width mode and the software emulation method through the routines gEnqLineWidthMode() and gEnqDeviceState() respectively.

gEnqLineWidthMode(sw)

The routines to actually set or enquire the current line thickness are:

gSetLineWidth(width)

gEnqLineWidth(width)

The line thickness, **width**, is in current drawing units and is not subject to any modelling or viewing transformation. The routine gEnqLineWidth() returns the currently selected line width which may not match the actual hardware line width appearing on the device due to hardware rounding or limitations. To determine the actual width of lines appearing on the device use gEnqSelectedPen().

As specified above, the line width is defined in current drawing units and not subject to any modelling or viewing transformations. It is possible to set an independent line width scaling factor through the routine:

gSetLineWidthScaling(s)

This provides the user with a method to scale the line width by a consistent scaling factor which can match the GINO scaling transformation for picture coordinates.

The current value of the scale factor can be inquired by the routine:

gEnqLineWidthScaling(s)

Drawing Mode

gSetPenType(type)

gEnqPenType(type)

gSetPenType() can be used to select a different drawing mode (see Appendix B for available drawing modes for each particular device):

Pen Identifier (**type**):	Constant:	Pen Type:	
0	GDEFAULT	Undefined	
6	GERASER	Eraser	
7	GNOT	NOT mode	}
8	GAND	AND mode	}Binary raster writing
9	GOR	OR mode	}modes for screen devices
10	GXOR	XOR mode	}
>10		Device dependent	

The routine gEnqPenType() enquires the current drawing mode. This may be used to determine the mode selected by the output device, or for routine interrogation of the current mode selected by the user.

If the device does not recognize the drawing mode selected, and is therefore unable to implement it, the actual drawing mode made available will not be the same as that requested. What is actually used may be identified by calling gEnqSelectedPen().

Line Ends

gSetLineEnd(end)

gEnqLineEnd(end)

The appearance of the ends of a line may be changed by the routine gSetLineEnd(). Line ends are shapes which are added onto the ends of a line. This is particularly useful when used in conjunction with thick lines as line ends help to give smooth and tidy joins between lines (see below). GINO offers three types of line end:

- No ends **end**=GNONE (0)
- Square ends **end**=GSQUARE (1)
- Round ends **end**=GROUND (2)

Line end types

For example:

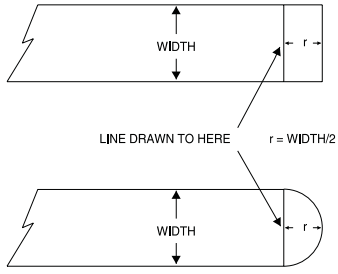
C code

```
gSetLineWidth(10.0);
x=60.0;
y=150.0;
for (i=0; i<3; i++)
{
/*  set line end type */
  gSetLineEnd(i);
/*  draw */
  gMoveTo2D(x,y);
  gDrawLineBy2D(-40.0,0.0);
  gDrawLineBy2D(0.0,-70.0);
  gDrawLineBy2D(40.0,-60.0);
  x += 60.0;
}
```

F90 code

```
call gSetLineWidth(10.0)
x=60.0
y=150.0
do i=0,2

!  set line end type
  call gSetLineEnd(i)
!  draw
  call gMoveTo2D(x,y)
  call gDrawLineBy2D(-40.0,0.0)
  call gDrawLineBy2D(0.0,-70.0)
  call gDrawLineBy2D(40.0,-60.0)
  x=x+60.0
end do
```

With no ends, line length is exactly as specified in the line drawing routines. However, square ends and round ends are added to the line. The amount added is calculated from the line width. For a square end this is a rectangle of WIDTH/2 by WIDTH, and for round ends it is a semicircle of radius WIDTH/2 as shown below.



Line end construction

The effect of **end** greater than 2 depends on the output device. If the device does not support other end types, the line defaults to no ends.

The routine gEnqLineEnd() returns the current line end setting.

Use of Current Attribute Enquiry Routines

In addition to routine interrogation to see which values are selected for the current line attribute, the enquiry routines may be usefully employed in other ways.

Example 1 - a current line attribute may be saved and subsequently restored:

C code

```
/*  Enquire, saving current line width as a variable */
   gEnqLineWidth(&saved_width);
/*  Set new current line width */
   gSetLineWidth(1.0);
/*  Draw line of the new width */
   gDrawLineTo2D(20.00,20.00);
/*  Restore saved line width */
```

F90 code

```
!  Enquire, saving current line width as a variable
   call gEnqLineWidth(saved_width)
!  Set new current line width
   call gSetLineWidth(1.0)
!  Draw line of the new width
   call gDrawLineTo2D(20.00,20.00)
!  Restore saved line width
   call gSetLineWidth(saved_width)
```

Example 2 - finding the default value of an attribute. Note that the enquiry should be made after device nomination and before any attempt to redefine the current attributes:

C code

```
/*  nominate device */
   gHp7475();
/*  enquire default (undefined) pen type */
   gEnqPenType(&type);
```

F90 code

```
!  nominate device
   call gHp7475
!  enquire default (undefined) pen type
   call gEnqPenType(type)
```

These techniques are generally applicable to the single attribute routines.

Attribute Tables

Attribute Definition Tables

GINO uses three attribute definition tables. These contain definitions of colour value, broken line settings, and complete line attribute sets. The tables are initialized whenever device nomination occurs (see page 39). The contents of the tables may be redefined, with values chosen by the user, by means of table definition routines.

Colour definition and the colour table are dealt with later in this document (see page 205).

Broken Line Types Table

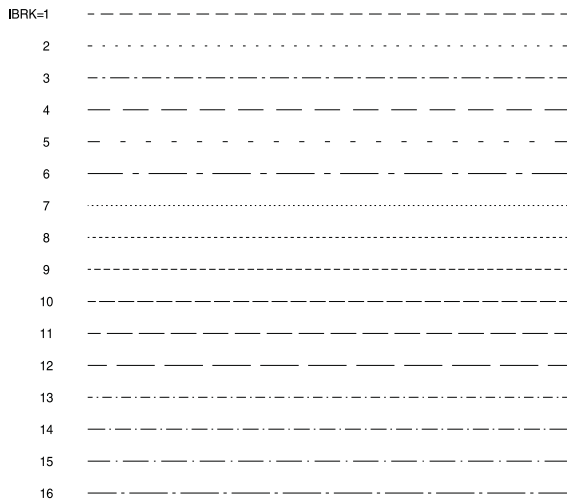
Up to 256 different broken line definitions can be stored in the broken line type table using the following two routines to define and enquire their settings:

gDefineBrokenLineStyle(brk, rep)

gEnqBrokenLineStyle(brk, rep)

where **brk** is the table entry and **rep** is a structure of type GBRKSTY which contains the broken line definition. The definitions stored in this table are implemented as the current broken line attribute when selected by calls to gSetBrokenLine() or, indirectly, to gSetLineStyle().

When a device is nominated the table entries (**brk** =1 to 16) are initialized with 16 different broken line types as follows:



These are repeated through the 256 entries in the complete table. Any attempt to use broken line styles outside the range 1 to 256 will result in solid lines being drawn.

The routine gDefineBrokenLineStyle() enables the user to redefine the line types stored in the table.

For example, the statements:

```
static GBRKSTY rep =
  {GDISCONTCHAIN,20.0,10.0,4.0};

gDefineBrokenLineStyle(7,&rep);
```

```
type (GBRKSTY) :: rep=GBRKSTY( &
GDISCONTCHAIN,20.0,10.0,4.0)

call
gDefineBrokenLineStyle(7,rep)
```

would define, and store in table entry 7, a line, which when selected, would cause the current line to look like the following after calling this command:

```
gSetBrokenLine(7);
```

```
call gSetBrokenLine(7)
```

Resulting in the following;

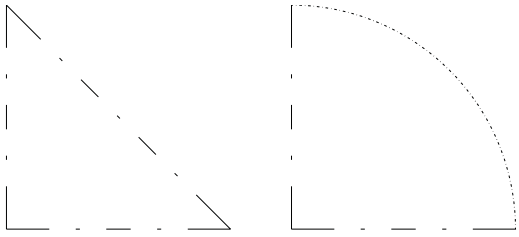
The arguments for the routine gDefineBrokenLineStyle() have the following meanings:

brk is the broken line type identifier and is used by gEnqBrokenLine(), gSetBrokenLine() and gDefineLineStyle(). It identifies entries in the table.

The structure GBRKSTY has four elements: **rep.mode**, **rep.repeat**, **rep.dash** and **rep.dot**, where **rep.mode** describes the sort of broken style in this way:

= GSOLID	defines a solid line
= GDISCONTDASH	defines a dashed, discontinuously drawn line
= GCONTDASH	defines a dashed, continuously drawn line
= GDISCONTCHAIN	defines a chained, discontinuously drawn line
= GCONTCHAIN	defines a chained, continuously drawn line

Discontinuously drawn lines are treated independently of each other so that any pattern (defined by **rep.repeat**, **rep.dash** and **rep.dot**) is centred along the line; the beginning and end dashes being made the same length. If the length of the line to be drawn is less than the repeat length, the pattern is scaled down to give one complete repeat sequence. In continuous mode the pattern simply carries on from one line to the next from wherever it has got to. Discontinuous mode gives clearly defined corners. Discontinuous mode scales down the pattern for small line segments (as are drawn for arcs) and therefore continuous mode can give a better result on curved lines. This effect can vary depending on the hardware capabilities of the device. Illegal values of **rep.mode** result in GINO issuing a warning message and a solid line being defined. **rep.repeat** specifies the pattern repeat length. **rep.dash** and **rep.dot** specify the lengths of the drawn elements which constitute the pattern.



Chained, discontinuous,
MODE=GDISCONTCHAIN

Chained, continuous,
MODE=GCONTCHAIN

rep.dash is the length of the first drawn element. **rep.dot** is the length of the second drawn element of a chained line. Space elements are made equal to (repeat-dash-dot)/2. If **rep.mode** specifies a dashed line mode, any value given for **rep.dot** will be ignored.

When a chained line is specified, **rep.dash** +**rep.dot** must not exceed repeat. If they do, GINO outputs a warning message and defaults to a solid line.

In the following example, a dashed line is defined and stored in table entry 8. This definition is then selected for the current line:

```
.
/*  define line type 8 */
static GBRKSTY rep =
 {GDISCONTDASH,18.0,13.0,0.0};

.
gDefineBrokenLineStyle(8, &rep);
/*  select line type 8 */
gSetBrokenLine(8);
/*  draw a line */
gMoveTo2D(30.0,50.0);
gDrawLineBy2D(145.0,0.0);
.
.
```

```
.
!  define line type 8
type (GBRKSTY) :: rep = &
     GBRKSTY(GDISCONTDASH, &
             18.0,13.0,0.0)
.
call gDefineBrokenLineStyle(8, &
     rep)
!  select line type 8
call gSetBrokenLine(8)
!  draw a line
call gMoveTo2D(30.0,50.0)
call gDrawLineBy2D(145.0,0.0)
.
.
```

This would produce a line that looks like this:

If a broken line definition is changed by gDefineBrokenLineStyle() while it is implemented for the current line, the current line's broken pattern will also be modified to conform to the new definition.

The broken line type table may be examined by calls to gEnqBrokenLineStyle(). For example:

```
GBRKSTY rep5;

gEnqBrokenLineStyle(5,&rep5);
```

```
type (GBRKSTY) rep5

call gEnqBrokenLineStyle(5,rep5)
```

will return the values defining the line type of table entry 5 in **rep5.mode**, **rep5.repeat**, **rep5.dash** and **rep5.dot**.

Continuous v Discontinuous

By default all broken linestyles, defined when a device is initialized, are discontinuous (i.e. either GDISCONTDASH or GDISCONTCHAIN). Where an application uses a lot of small line segments (eg. arcs, curves etc.) it is often preferable to use the continuous broken lines styles and it would be necessary to redefine all the required broken linestyles to achieve the desired results. A shortcut to this is provided in the following routine:

gSwitchBrokenLineStyles(switch)

where **switch** can be GCONTDASH or GDISTCONTDASH. When this routine is called, ALL the entries in the broken line table are set to be either continuous or discontinuous as requested. All other settings in the table (i.e. repeat length, dash and dot lengths) are not affected by this call.

Line Definition Table

Up to 256 different line definitions can be stored in the line definition table using the following two routines to define and enquire their settings:

gDefineLineStyle(line, rep)

gEnqLineStyle(line, rep)

where **line** is the table entry and **rep** is a structure of type GLINSTY which contains the line definition. A complete line definition involves the specification of values for each of the six line attributes.

Whenever a device is nominated, all 256 table entries are initialized with the following values:

rep.vis	rep.brk	rep.col	rep.width	rep.type	rep.end
1	0	line	0.2(mm)	0	0

The line styles table does not have the same default settings for each entry. An entry may be redefined by calling routine gDefineLineStyle(). The valid range for gDefineLineStyle's arguments is the same as for the individual attribute routines gSetLineVis(), gSetBrokenLine(), gSetLineColour(), gSetLineWidth(), gSetPenType() and gSetLineEnd().

The statements:

```
static GLINSTY line1 = {
  GVISIBLE, GLONGDASH, GYELLOW,
  0.6, GDEFAULT, GROUND};

gDefineLineStyle(1,&line1);
```

```
type (GLINSTY) :: line1=GLINSTY(
  GVISIBLE, GLONGDASH, GYELLOW, &
   0.6, GDEFAULT, GROUND)

call gDefineLineStyle(1,line1)
```

would define a line, identified in the table as **line**=1, which was visible (**line1.vis**=GVISIBLE), discontinuous dashed (**line1.brk**=GLONGDASH, assuming GINO default broken types), yellow (**line1.col**=GYELLOW, assuming standard GINO colours), 0.6mm wide (**line1.width**=0.6), drawn with a default pen type (**line1.type**=GDEFAULT), and has round ends (**line1.end**=GROUND). This example assumes that the values of the entries selected in the colour and broken line tables have not been changed since device nomination. If entries in these tables are ever redefined it is the new values that will be selected via the line definition table.

Table entries may be examined by using gEnqLineStyle(). **line** identifies the entry of interest and its parameter values are returned in **rep.vis**, **rep.brk**, etc. If **line**=0, the current line attributes are returned.

The absolute value of **line** is used for gDefineLineStyle() and gEnqLineStyle(). If the absolute value is greater than 256, a warning message is output and no further action is taken.

Changing the Current Line Attributes

The routine gDefineLineStyle() can be used to redefine the current line attributes by setting its first argument, the line style identifier **line**, to zero:

```
static GLINSTY rep={
  GVISIBLE, GLONGCHAINED, GBLUE,
  0.2, GDEFAULT, GNONE};

gDefineLineStyle(0,&rep);
```

```
type (GLINSTY) :: rep=GLINSTY(&
  GVISIBLE,GLONGCHAINED,GBLUE, &
  0.2, GDEFAULT, GNONE)

call gDefineLineStyle(0,rep)
```

This single call is equivalent to a call to each of the 6 individual attribute controlling routines:

```
gSetLineVis(GVISIBLE);
gSetBrokenLine(GLONGCHAINED);
gSetLineColour(GBLUE);
gSetLineWidth(0.2);
gSetPenType(GDEFAULT);
gSetLineEnd(GNONE);
```

```
call gSetLineVis(GVISIBLE)
call gSetBrokenLine(GLONGCHAINED)
call gSetLineColour(GBLUE)
call gSetLineWidth(0.2)
call gSetPenType(GDEFAULT)
call gSetLineEnd(GNONE)
```

In the following example all the attributes of the current line are changed with the exception of colour. These values are reported to the program by using gEnqLineStyle() with **line** set to zero so that it points to the current line.

C code

```
   int       saved_colour;
   GBRKSTY   def, newdef;
/* Save the current line colour */
   gEnqLineColour(&saved_colour);
/* Redefine current line but using saved colour id. */
   def.vis = GVISIBLE;
   def.brk = GLONGCHAINED;
   def.col = saved_colour;
   def.width = 0.6;
   def.type = GDEFAULT;
   def.end = GROUND;
   gDefineLineStyle(0,&def);
/*  enquire about new current line */
    gEnqLineStyle(0,&newdef);
```

F90 code

```
   integer saved_colour
   type (GBRKSTY) def, newdef
!  Save the current line colour
   call gEnqLineColour(saved_colour)
!  Redefine current line but using saved colour id.
   def%vis = GVISIBLE
   def%brk = GLONGCHAINED
   def%col = saved_colour
   def%width = 0.6
   def%type = GDEFAULT
   def%end = GROUND
   call gDefineLineStyle(0,def)
!  enquire about new current line
   call gEnqLineStyle(0,newdef)
```

Retrieving and Storing Current Line Attributes

Two routines are provided to set and save a line style definition:

gSetLineStyle(line)

gSaveLineStyle(line)

The routine gSetLineStyle() selects a complete line attribute specification (identified by **line**) from the table of line definitions and implements this for the current line. For example:

```
gSetLineStyle(5);
```

```
call gSetLineStyle(5)
```

causes line definition 5 to be selected as the current line style. If the definition identifier **line** is outside the range 1 to 256, no change is made to the current line but a warning message is issued.

The routine gSaveLineStyle() saves the complete specification of the current line attributes to the line definition table. For example:

```
gSaveLineStyle(10);
```

```
call gSaveLineStyle(10)
```

copies the values of all the current line attributes into entry 10 of the line definition table. This line specification can be reselected later. For example:

```
gSetLineStyle(10);
```

```
call gSetLineStyle(10)
```

In the example below, entries are made in the line definition table in two ways. First by using gDefineLineStyle() to assign values to entries 1 and 2, and second by using gSaveLineStyle() to copy current line attribute values into entry 3.

C code

```
   static GLINSTY line1 = {1, 2, 2, 0.2, 2, 0},
                  line2 = {1, 6, 2, 3.0, 2, 1};
   gDefineLineStyle(1, &line1);
   gDefineLineStyle(2, &line2);
/*  select a set of line attributes */
   gSetLineStyle(2);
/*  draw a line */
   gMoveTo2D(20.0, 100.0);
   gDrawLineBy2D(200.0, 0.0);
/*  change with line width and broken style */
   gSetLineWidth(0.2);
   gSetBrokenLine(GLONGCHAINED);
/*  draw result */
   gMoveTo2D(20.0, 80.0);
   gDrawLineBy2D(200.0, 0.0);
/*  save line to table entry 3 */
   gSaveLineStyle(3);
/*  select another line from table and draw it */
   gSetLineStyle(1);
   gMoveTo2D(20.0, 60.0);
   gDrawLineBy2D(200.0, 0.0);
/*  reselect saved line and draw */
   gSetLineStyle(3);
   gMoveTo2D(20.0, 40.0);
   gDrawLineBy2D(200.0, 0.0);
```

F90 code

```
      type (GLINSTY) :: line1 = GLINSTY(1,2,2,0.2,2,0), &
                        line2 = GLINSTY(1,6,2,3.0,2,1)
      call gDefineLineStyle(1, line1)
      call gDefineLineStyle(2, line2)
/*  select a set of line attributes
      call gSetLineStyle(2)
/*  draw a line
      call gMoveTo2D(20.0, 100.0)
      call gDrawLineBy2D(200.0, 0.0)
/*  change with line width and broken style
      call gSetLineWidth(0.2)
      call gSetBrokenLine(GLONGCHAINED)
/*  draw result
      call gMoveTo2D(20.0, 80.0)
      call gDrawLineBy2D(200.0, 0.0)
/*  save line to table entry 3
      call gSaveLineStyle(3)
/*  select another line from table and draw it
      call gSetLineStyle(1)
      call gMoveTo2D(20.0, 60.0)
      call gDrawLineBy2D(200.0, 0.0)
/*  reselect saved line and draw
      call gSetLineStyle(3)
      call gMoveTo2D(20.0, 40.0)
      call gDrawLineBy2D(200.0, 0.0)
```

Chapter 6

CHARACTERS

Character Introduction

GINO provides the ability for drawing characters using hardware and software fonts, and a large variety of symbols and special characters, together with routines for specifying font, size, justification, shape, orientation and fill style.

The routines described in this section are:

a) For outputting:

ASCII characters	gDisplayAsciiChar()	
Character strings	gDisplayStr()	
	gPrintf()	
	gDisplayStrPolylineTo2D()	
	gDisplayStrPolylineBy2D()	
	gFitCharStr()	
Numbers	gDisplayRealExponent()	gConvertRealExponent()
	gDisplayRealFixed()	gConvertRealFixed()
	gDisplayRealFloat()	gConvertRealFloat()
	gDisplayInteger()	gConvertInteger()
Special symbols	gDrawMarker()	

b) For specifying and enquiring:

Font availability		gEnqHardFontList()
Font style	gSetCharFont()	
	gEnqFontStyle()	
	gSetFontFillStyle()	
	gSetFontWeight()	
	gSetFontSpacing()	
	gSetFontForm()	
Character attributes	gSetHardChars()	gEnqCharAttribs()
	gSetMixedChars()	
	gSetSoftChars()	
	gSetHardCharSize()	
	gSetCharSize()	
	gSetCharSizePoint()	
	gSetStrAngle()	
	gSetItalicAngle()	
	gSetCharTransformMode()	
Text blocks	gStartTextBlock()	gEnqTextBlockAttribs()
	gMoveToNextLine()	
	gSetInterlineSpace()	
String attributes	gSetStrJustify()	gEnqStrJustify()
	gSetStrUnderscore()	gEnqStrUnderscore()
	gSetStrExponent()	gEnqStrExponent()
Null character form	gDefineNullChar()	
Escape character	gSetEscapeChar()	gEnqEscapeChar()
Enquiry	gEnqCharTransform()	
	gReturnStrInfo()	

Character Mode - Hardware v Software

GINO provides the following modes for character output:

Hardware - Characters are entirely generated by the device. This produces the most presentable output and is usually the quickest, but output may vary from device to device.

Software Untransformable - Characters are generated by GINO using solid straight lines and area filling, according to the requested character attributes. The characters cannot be transformed.

Software Transformable - As above but generated with the current line style (Dashed pattern and Thickness), transformed by the current transformation (rotation, scale and shear - if any) and windowed/masked by the current window or mask settings.
The quality of software characters are not as good as hardware, but the output on all devices will be exactly the same.

By default, GINO starts up in an untransformable 'mixed' mode (gSetMixedChars()), whereby, characters will be output by hardware if the size is available, otherwise will be output by software. On devices where only a limited number of sizes are available, this will result in some strings being drawn with a hardware font and some with a software font which may look untidy.
To change the mode, call one of the following routines:

gSetHardChars()	Characters always generated by the device, but size may only be the nearest available if hardware only supports limited sizes.
gSetMixedChars()	Characters are generated by the device if within 10% of the size requested, 1 degree of specified orientation and 5 degrees of specified italic angle, otherwise software characters selected.
gSetSoftChars()	Characters generated by GINO exactly as specified, but without applying any GINO transformation.
gSetCharTransformMode()	Characters generated by GINO exactly as specified and applying current transformation.

Output of Characters

Single ASCII Characters

Any ASCII character may be output using:

gDisplayAsciiChar(code)

where **code** is an integer in the range 0 - 255, and is the decimal ASCII representation of the required character. ASCII characters are shown in the font tables in Appendix C.

Example:

```
/* Output the character A */
      gDisplayAsciiChar(65);
```

```
! Output the character A
      call gDisplayAsciiChar(65)
```

The routine gDisplayAsciiChar() is useful for outputting non-printable characters. Single ASCII characters are not affected by the string justification routine but are affected by the character attribute routines gSetCharSize(), gSetStrAngle(), gSetItalicAngle(), and gSetStrUnderscore().

Character Strings

Character strings may be output using one of the routines:

gDisplayStr(string)

gPrintf(char format, ...) **[C/C++ only]**

The routine gDisplayStr() passes the argument **string** to the output device. It may include any combination of characters in the ASCII set, and any special characters permitted by the system.

```
gDisplayStr("Hello World");
```

```
call gDisplayStr("Hello World")
```

The routine gPrintf() passes a format string together with optional arguments and expands it before passing it to the output device via. the former routine. gPrintf() therefore provides the format control of printf() together with the character control of gDisplayStr() for graphical character output.

```
gPrintf("Command %s - Section %d.%d",command,sect,subsect);
```

Output of Numbers

Integer and real numbers may be output using the character routines:

gDisplayInteger(number, nwidth)

gDisplayRealFloat(value, nwidth)

gDisplayRealFixed(value, nwidth, nplace)

gDisplayRealExponent(value, nwidth, nplace)

The format of the output is controlled by the argument **nwidth** and, in the case of gDisplayRealFixed() and gDisplayRealExponent(), **nplace** which is the number of decimal places required. The character string attributes apply to the output of numbers.

Field Width

nwidth specifies the character field width of the output. No output is generated if **nwidth** is zero. If a number is too large for the specified field width, a string of asterisks is output. The field width is limited to 32 characters.

The sign of **nwidth** determines whether a number is right- or left-justified within the field width. Any spare character positions are filled with spaces. If **nwidth** is greater than zero, the output is right-justified. If **nwidth** is less than zero, the output is left-justified.

A leading zero, when appropriate, will be output before a decimal point, provided there is room for it.

Example:

C code

```
/* Output nyear left-justified in a filed of 4 */
   gDisplayInteger(nyear,-4);
/* Output rate in a fixed-point format with 2
   decimal figures, right-justified in a field of 10*/
   gDisplayRealFixed(rate,10,2);
/* Output fact in floating-point form with 8
   decimal figures, left-justified in a field of 15. */
   gDisplayRealExponent(fact,-15,8);
/* Output sales in floating-point form,
   right-justified in a field of 12. */
   gDisplayRealFloat(sales,12);
```

F90 code

```
!  Output nyear left-justified in a filed of 4
   call gDisplayInteger(nyear,-4)
!  Output rate in a fixed-point format with 2
!  decimal figures, right-justified in a field of 10
   call gDisplayRealFixed(rate,10,2)
!  Output fact in floating-point form with 8
!  decimal figures, left-justified in a field of 15.
   call gDisplayRealExponent(fact,-15,8)
!  Output sales in floating-point form,
!  right-justified in a field of 12.
   call gDisplayRealFloat(sales,12)
```

```
gDisplayInteger(28282,10);                  =    28282

gDisplayInteger(28282,-10);                 =28282

gDisplayRealFixed(28.282,10,3);             =   28.282

gDisplayRealFixed(28.282,10,1);             =     28.3

gDisplayRealExponent(28.282,10,2);          =  0.28E  2

gDisplayRealExponent(28.282,-10,4);         =0.2828E  2

gDisplayRealFloat(28.282,10);               =0.2828E  2

gDisplayRealFloat(28.282,5);                =**
```

Numeric output

Conversion of Numbers to Character Strings

The following routines convert numbers to character strings:

gConvertInteger(number, nwidth, string)

gConvertRealFloat(value, nwidth, string)

gConvertRealFixed(value, nwidth, nplace, string)

gConvertRealExponent(value, nwidth, nplace, string)

The numbers are stored as strings in the same format as the output produced by the routines gDisplayInteger(), gDisplayRealFloat(), gDisplayRealFixed() and gDisplayRealExponent(). The length of the resultant string is limited to 32 characters. If the format of the number exceeds this then it is truncated to 32 characters and a warning message is output.

The position of the number in the character array **string** is determined by the value of **nwidth**. If **nwidth** is positive then the number is right justified. If **nwidth** is less than zero then the number is left justified. Nothing is returned if **nwidth** is zero for any of the routines.

The character string, once returned, may be used in other string handling routines. For example, passing the string to the routine gReturnStrInfo() would allow the size of the number, output in the current font, to be returned. The string may be concatenated with other number strings, text strings, and escape characters.

Character Fonts

Font styles

The default font (font 0) on all devices is a fixed width font. A fixed width font may be produced by hardware if the device has the capability or GINO's default software font will be used.

The default font can be changed using the following routine:

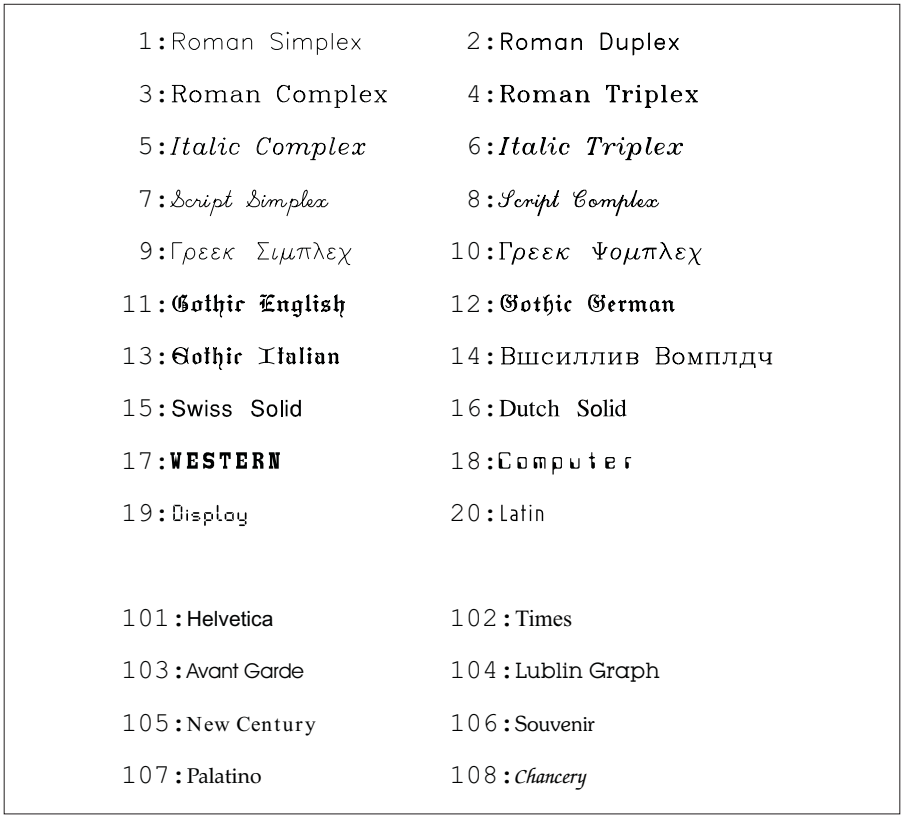
gSetCharFont(font)

Values of **font** greater than zero determine the style of font to be used. Font numbers 1 to 99 specify software font styles. Numbers between 100 and 108 specify a set of registered hardware fonts, if font numbers 101-108 are not available a software equivalent font will be selected. Numbers greater than 108 access device specific hardware fonts.

The current character mode setting can affect the hardware or software accessing of fonts. For all software fonts the software character mode applies (as if gSetSoftChars() had been called). For the current registered hardware fonts the character mode can be controlled by the gSetSoftChars() / gSetHardChars() routines, with an approximate software equivalent addressing shortfalls in the specific hardware capabilities. Non-registered hardware fonts (108+) operate in the hardware character mode only (as if gSetHardChars() had been called).

The availability of hardware fonts on any particular device can be enquired with the routine gEnqHardFontList (described later). Appendix B also details the font styles available for a specific device.

Fonts 70 to 79 are primarily symbol characters which are provided for use with the gDrawMarker() routine, which outputs a character centred upon the current character position, however they can also be used with gDisplayStr() providing a mapping between ASCII characters and symbols.



GINO Software and Hardware Fonts

GINO Software Fonts

0	GDEFAULT
1	GRoman_Simplex
2	GRoman_Duplex
3	GRoman_Complex

4 GRoman_Triplex
5 GItalic_Complex
6 GItalic_Triplex
7 GScript_Simplex
8 GScript_Complex
9 GGreek_Simplex
10 GGreek_Complex
11 GGothic_English
12 GGothic_German
13 GGothic_Italian
14 GCyrillic_Complex
15 GSwiss_Solid *
16 GDutch_Solid *
17 GWestern *
18 GComputer *
19 GDisplay *
20 GLatin *
21 GGreek_Font_1
22 GGreek_Font_2
23 GGreek_Font_3
24 GGreek_Font_4
25 GGreek_Font_5

Symbol Fonts

70 GMaths_Symbols * (DIN 6776)
71 GHershey_Maths_Symbols
72 GHershey_Symbols_1
73 GHershey_Symbols_2
74 GSymbol1_normal *
75 GSymbol1_thick *
76 GSymbol1_filled *
77 GSymbol2 _normal *
78 GSymbol2 _filled *
79 GGINO_Dingbats *

Hardware and Software Fonts

100	GCourier [Hardware only]
101	GHelvetica
102	GTimes
103	GAvant_Garde
104	GLublin_Graph
105	GNew_Century_Schoolbook
106	GSouvenir
107	GPalatino
108	GChancery
>108	Device specific hardware fonts (see Appendix B)

Notes:

(i) * indicates the font is defined as a polygon and can be filled (see below).

(ii) All fonts are proportionally spaced except font numbers 0 and 100.

Font Fill Style

When hardware fonts or software fonts that are defined as polygons are used, the default fill style is solid fill in the current pen colour. This can be varied by using the following routine:

gSetFontFillStyle(style)

where the structure **style** contains the following integer elements: **style.type**, **style.ffill**, **style.fline, style.bfill**, **style.bline**. **style.type** defines the type of filling that is required. **style.ffill** and **style.bfill** define the foreground and background filling style and **style.fline** and **style.bline** define the foreground and background line style if a filled font style is selected. If either **style.ffill** or **style.bfill** = GNOFILL the foreground or background filling respectively is omitted. The default setting for **style** is {GFILLED, GSOLID, GCURRENT, GNOFILL, GCURRENT}.

The font style parameters can be set for hardware fonts, but the number of foreground and background fill and line styles may be limited, whereas other styles may be provided. Refer to Appendix B for the particular device.

```
style = {GOUTLINE};
style = {GFILLED,GSOLID,GCURRENT,GNOFILL};
style = {GOUTFIL,3,GCURRENT,GNOFILL};
style = {GOUTFIL,8,GCURRENT,11,GCURRENT};
```

Examples of Software Filled Text

Font Weight

Many hardware fonts have a weighting factor which increases or decreases either the boldness of a font or the thickness of the vectors by which the font is drawn (see Appendix B for the capability of the current device). This weight factor can be set using:

gSetFontWeight(weight)

where **weight** is a positive or negative integer which will adjust the weight factor of both hardware and software fonts. The following values are suggested for standard weights:

Value	Font weight
-6	Extra Thin
-3	Thin
0	Normal
+3	Bold
+6	Extra Bold

Other values of **weight** may have a corresponding effect, depending on the resolution of the device. ‘Normal’ font weight refers to the default pen width.

For GINO's software fonts, gSetFontWeight() adjusts the thickness of the vectors used in drawing the characters or their boundary but it does not affect the shape of any polygons, therefore gSetFontWeight() will not affect strings drawn with font fill style type = GFILLED.

Fixed Pitch Control

It is possible to force the output of both hardware and software proportional fonts to appear as if they were fixed pitch. This is achieved with the routine:

gSetFontSpacing(space)

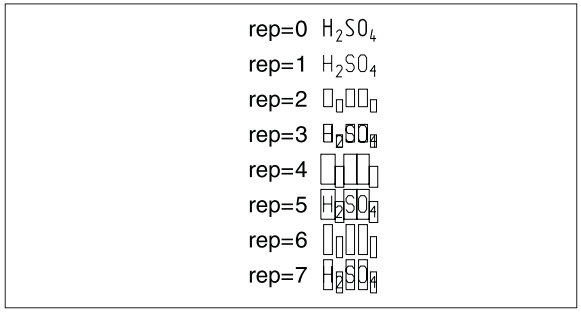
If **space** = GNORMAL then the font is output as defined, either fixed pitch or proportional. If **space** = GFIXEDPITCH then the font is forced to be fixed pitch.

Software Font Representation

As the display of software fonts can be time consuming, GINO provides a means to simplify the representation of these fonts during program development.

gSetFontForm(rep)

The default setting for **rep** is zero, where the requested software font is output as requested. Other settings of **rep** display either a box representing the character width and height or a box and the same character but drawn in the default GINO software font. Font weight is ignored when **rep**> 0. The variety of boxes either include or omit left and right bearings and other height limits of the current software font. The routine gSetFontForm() has no effect on font 0. (Hardware, Greek and Symbol fonts are not output in the GINO software font, for odd values of **rep**).



Various Font Representations

Font Enquiry

The routine gEnqFontStyle() is provided to enquire all of the above font attributes.

gEnqFontStyle(font, style, weight, space, rep)

where **font** is set by gSetCharFont(), **style** is set by gSetFontFillStyle(), **weight** is set by gSetFontWeight(), **space** is set by gSetFontSpacing() and **rep** is set by gSetFontForm().

The routine gEnqHardFontList() returns a list of the hardware fonts available with the current device.

gEnqHardFontList(list, n, count)

where **list** is an integer array of length **n**. gEnqHardFontList() returns a list of hardware font numbers as used by gSetCharFont(). The total number of hardware fonts available on the current device is returned in **count**.

Character Attributes

Default Character Settings

The default settings for the output of characters are:

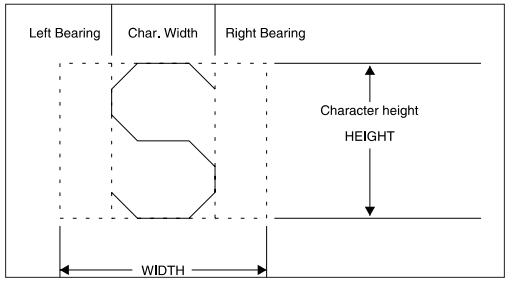
- font 0 - (Fixed width hardware font if available, otherwise software)
- left justified
- not underlined
- size 3mm x 3mm or nearest hardware equivalent
- angle 0 degrees
- italic 0 degrees

Character Size

The default width and height of characters depends on the output device and may be altered using the routine:

gSetCharSize(width, height)

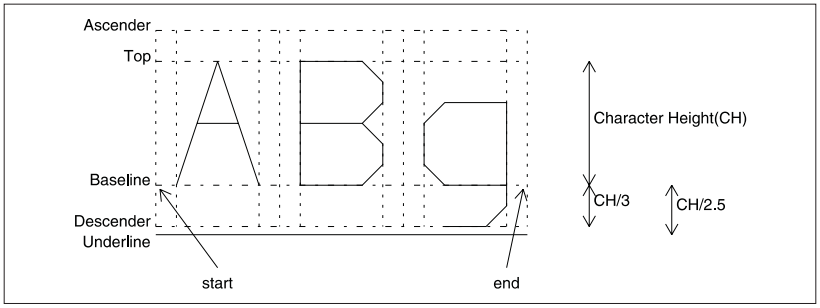
For fixed pitch software fonts the size of the character box is the same for all characters, there is no left bearing and the right bearing is equivalent to one third of the current GINO character width. For proportional fonts the character box will vary for each character and the size relates approximately to the upper case letter M. The specified width equates to the left and right bearing together with the individual character width.



Fixed Pitch Character Box Size

If lower case characters with tails or descenders are being drawn (i.e. g, j, p, q or y), the box is extended as shown in the figure below. In this case the pen is left at the right-hand corner of the box at baseline level.

Also note that other than for software characters, if any part of each character box in a string (including ascenders or descenders) lies outside the current clipping window, the whole character is not output. Therefore trying to output a string that only contains upper-case letters along the Y=0.0 line, will still result in no output because the character box extends below the clipping margin.



Character Box Dimensions

For example to specify the character box to be 20mm wide and 10mm high:

```
gSetCharSize(20.0,10.0);
```

```
call gSetCharSize(20.0,10.0)
```

If gSetHardChars() has been called, the characters may not be drawn to the exact size requested. If the device does not support all character sizes, the nearest available hardware character size is selected. If the device cannot generate characters at all, they will be generated by GINO to the exact size requested.

An alternative method of specifying a character size is to use the routine:

gSetCharSizePoint(points)

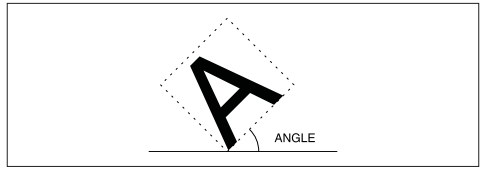
where **points** is the requested character size in printers points (1/72nd inch). Using this routine is equivalent to setting a square character size using gSetCharSize() with the same implications in terms of character mode as documented above.

Character Orientation

The orientation of characters may be specified using the routine:

gSetStrAngle(angle)

angle specifies the angle in degrees between the characters to be drawn and the horizontal (or the current X-axis if it has been transformed). The character strings are rotated about the bottom left-hand corner of the character string. Positive rotation is anti-clockwise as indicated in the figure below.



Character Orientation

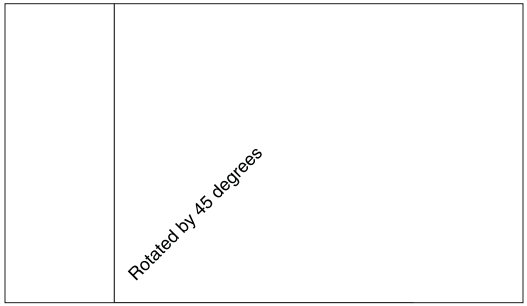
The pen is left at the bottom right-hand corner of the rotated box to ensure that consecutive character strings are concatenated.

Example:

```
gSetStrAngle(45.0);
gMoveTo2D(x,y);
gDisplayStr("Rotated by ");
gDisplayStr("45 degrees");
```

```
call gSetStrAngle(45.0)
call gMoveTo2D(x,y)
call gDisplayStr('Rotated by ')
call gDisplayStr('45 degrees')
```

would produce the output shown below.



String Rotation

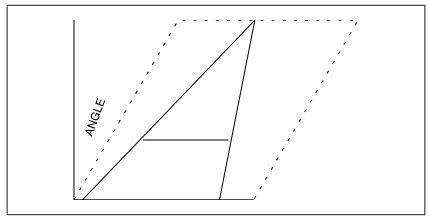
If gSetHardChars() has been called and angled character strings cannot be produced on the device, an approximation is provided in the form of a stepped character string.
Note that Characters strings can also be rotated if Software Transformable character mode has been selected with gSetCharTransformMode(GON) and a GINO transformation is currently active (see page 239).

Italic Characters

Italic characters may be selected using the routine:

gSetItalicAngle(slant)

slant specifies the angle in degrees between the slope of the characters and the vertical (or the current Y-axis if transformable characters are switched on). A positive angle represents a clockwise slope.



Italic Character

```
gSetCharSize(5.0,6.0);
gSetItalicAngle(30.0);
gDisplayStr("Italics");
```

```
call gSetCharSize(5.0,6.0)
call gSetItalicAngle(30.0)
call gDisplayStr('Italics')
```

would produce this output:

Italics

If gSetHardChars() has been called and the device cannot draw italicized characters, then characters will be drawn non-italicized.

Current Character Settings Enquiry

The current character settings can be examined by calling the routine:

gEnqCharAttribs(rep)

where **rep** is a structure of type GCHASTY. The default character settings are set up each time a device is nominated and gEnqCharAttribs() will return these if the call is made before any of the character settings are changed. The default character settings are device dependent (see Appendix B). If gSetHardChars() has been called, the character settings returned by gEnqCharAttribs() may differ from the requested settings.

To enquire the current angle and italic effects on a character string, the routine gEnqCharTransform() requires the width and height of the character string and returns the relative distance subject to the current angle and italic transformations.

gEnqCharTransform(dx, dy, point)

This routine is useful for evaluating the area that a particular string will occupy. By passing the length of the current string and the current character height, the end position of the string is given relative to the start.

Underlining of Characters

Characters may be underlined with a solid line using the routine:

gSetStrUnderscore(swi)

Underlining is switched on for all subsequent character and string output when **swi** is set to GON. The underlining occurs at 0.4 * the character height below the characters baseline.

The current setting for underlining can be enquired using the routine:

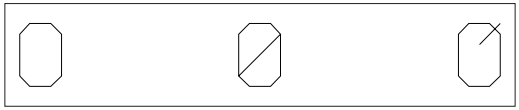
gEnqStrUnderscore(swi)

Representation of Zero Character

The routine gDefineNullChar() offers alternative representations of the zero character for the GINO default software font.

gDefineNullChar(nul)

where **nul** is in the range 0 - 2. The zero can be represented in the following forms:



Line Attributes affecting Characters

The Line attribute of colour always affects the drawing of characters, however the visibility, dashed-line style, line thickness, pen-type and line-end type only affect characters if Software Transformable Characters have been selected with gSetCharTransformMode() (see page 239).

Character String Attributes

There are a number of attributes which affect a complete character string:

- Justification
- Text Blocks
- Exponent and Index Settings
- Escape Characters

Justification

Character strings may be left, right or centre justified.

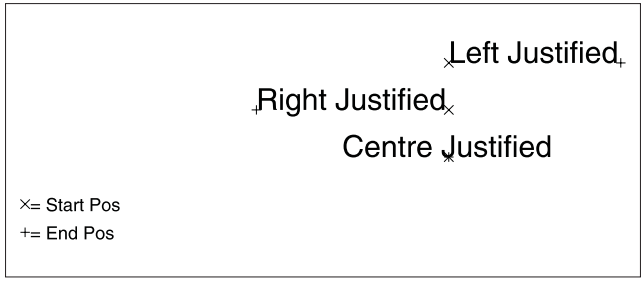
gSetStrJustify(jus)

gEnqStrJustify(jus)

For left justified characters - **jus** = GLEFT (the default), the start position is at the bottom left-hand corner of the character box, and the pen is left at the bottom right-hand corner of the box for subsequent output.

For centre justified characters - **jus** = GCENTRE, the start position is at the centre of the character box, and the pen is left at the centre of the box for subsequent output.

For right justified characters - **jus** = GRIGHT, the start position is at the bottom right-hand corner of the character box, and the pen is left at the bottom left-hand corner of the box for subsequent output.



Justified Strings

When outputting numeric output, take care not to confuse the justification of the number within the field width (which is controlled with the sign of the field width; <0 = left-justified, >0 = right-justified) and the justification about the current point which is controlled by gSetStrJustify. Outputting centred numbers with **jus** =GCENTRE will only work if the number is exactly the same length as the indicated field width.

Text Blocks

Text blocks can be created by using the routines:

gStartTextBlock(xbeg, ybeg)

gMoveToNextLine()

The **xbeg** and **ybeg** coordinates define the position of the first line of a text block. Subsequent line positions are set using the gMoveToNextLine() routine or by using the *N escape sequence. The distance between the lines of a text block can be set by changing the inter-line spacing with the routine:

gSetInterlineSpace(factor)

where **factor** is a factor of the current character height. The default setting of the inter-line spacing is 2.0 * character height, but this may be set to any positive or negative real value, allowing overwriting or lines to be placed above each other.

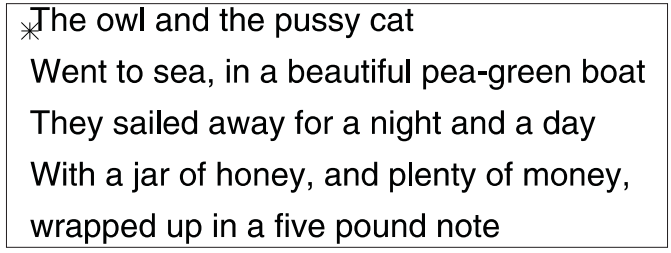
C code

```
gStartTextBlock(10.0,100.0);
gDrawMarker(42);
gDisplayStr("The owl and the pussy cat");
gMoveToNextLine();
gDisplayStr("Went to sea, in a beautiful pea-green boat");
gMoveToNextLine();
gDisplayStr("They sailed away for a night and a day");
gMoveToNextLine();
gDisplayStr("With a jar of honey, and plenty of money,");
gMoveToNextLine();
gDisplayStr("wrapped up in a five pound note");
gMoveToNextLine();
```

F90 code

```
call gStartTextBlock(10.0,100.0)
call gDrawMarker(42)
call gDisplayStr('The owl and the pussy cat')
call gMoveToNextLine
call gDisplayStr('Went to sea, in a beautiful pea-green boat')
call gMoveToNextLine
call gDisplayStr('They sailed away for a night and a day')
call gMoveToNextLine
call gDisplayStr('With a jar of honey, and plenty of money,')
call gMoveToNextLine
call gDisplayStr('wrapped up in a five pound note')
call gMoveToNextLine
```

A call to gMoveToNextLine() without initializing the start position through gStartTextBlock() will generate a warning message. The start position of the text block will be taken as the current pen position upon the call to gMoveToNextLine().



Use of Text Block Routines

The current text block settings can be obtained by calling the routine:

gEnqTextBlockAttribs(xbeg, ybeg, factor)

xbeg, **ybeg** return the current text line position within the text block.

Exponents and Indices

Exponents and indices can be drawn within character strings using the escape sequences *E and *I as described. The position and size of exponents and indices are set or enquired with the routines:

gSetStrExponent(relcw, relch, posexp, posind)

gEnqStrExponent(relcw, relch, posexp, posind)

where **relcw** and **relch** are the relative character width and height of both exponents and indices. **posexp** is the relative character height above the baseline at which the exponents are drawn and **posind** is the relative character height below the baseline at which indices are drawn.

Escape Characters

An escape mechanism enables various control functions to be specified in character strings. Initially, the escape character is an asterisk *. The following controls are provided:

'*.'	String terminator
'*L'	Shift to lowercase
'*U'	Shift to upper-case
'*F*nnn*'	Change to font nnn (eg: change to font 3: *F003)
'*FS'	Sets the GINO font to the temporary string font
'*FR'	Restores the font which was current when the string routine was called
'*N'	Move to next line of text block
'*E'	Set exponent (0.6*height above baseline)
'*I'	Set Index (0.3*height below baseline)
'*A'	Align position (also resets exponents, indices, underline and weight)
'*B'	Move back to last align position on baseline
'*O'	Position next character over previous character at exponent size setting
'*S'	Underline following characters
'*\'	Set italic -15 deg
'*|'	Set italic 0 deg
'*/'	Set italic +15 degrees
'*W'	Bold following characters
'*:'	Umlaut facility (must be followed by an A,O,U, or S)
'*:S'	Displays German sz character if available in current font
'**'	Output the escape character

The functionality of these facilities are detailed in the reference document in the description of gDisplayStr(). For all strings specified as character variables or arrays, it is advisable that the string be terminated by '*.'.

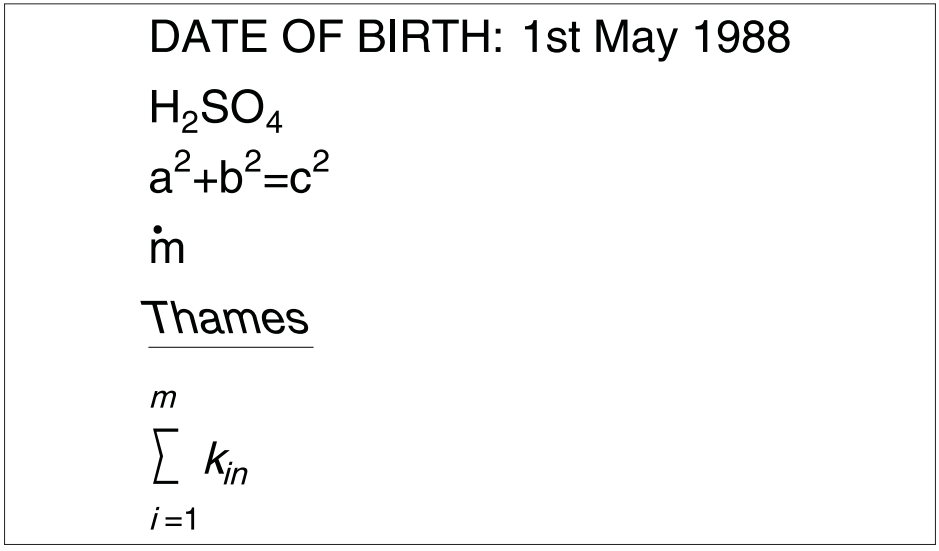
The following sequence of statements produces the characters shown in the figure below:

C code

```
gDisplayStr("DATE OF BIRTH: ");
gDisplayStr("1*LST *UM*LAY 1988");
gMoveToNextLine();
gDisplayStr("H*I2*ASO*I4*.");
gMoveToNextLine();
gDisplayStr("a*E2*A+b*E2*A=c*E2*.");
gMoveToNextLine();
gDisplayStr("m*B*E*F070T*.");
gMoveToNextLine();
gDisplayStr("*S*\Thames*.");
gMoveToNextLine();
gDisplayStr("*F070S*FS*B*I*I*I*I*/i*| ="
    "1*B*E*E*E*E*/m*Ak*Iin*.");
```

F90 code

```
call gDisplayStr(DATE OF BIRTH: ')
call gDisplayStr('1*LST *UM*LAY 1988')
call gMoveToNextLine
call gDisplayStr('H*I2*ASO*I4*.')
call gMoveToNextLine
call gDisplayStr('a*E2*A+b*E2*A=c*E2*.')
call gMoveToNextLine
call gDisplayStr('m*B*E*F070T*.')
call gMoveToNextLine
call gDisplayStr('*S*\Thames*.')
call gMoveToNextLine
call gDisplayStr('*F070S*FS*B*I*I*I*I*/i*| =' // &
     '1*B*E*E*E*E*/m*Ak*Iin*.')
```



Use of Escape Sequences

Changing the Escape Character

When it is not convenient to use '*' as the escape character, another character (for example '!') can be selected using:

gSetEscapeChar(cha)

The escape character may be set to any character in the ASCII set excepting those used for escape sequences themselves.

```
gSetEscapeChar("!");
```

```
call gSetEscapeChar('!')
```

Escape Character Enquiry

The currently selected escape character may be obtained by using:

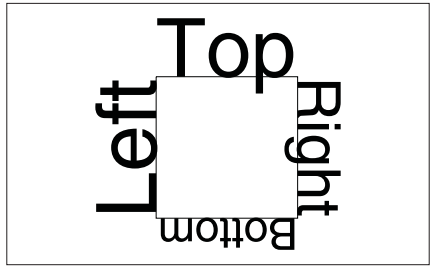
gEnqEscapeChar(cha)

Character Strings Adjusted to Fit a Specified Width

To output a string to fit between any two arbitrary points, the routine gFitCharStr() can be used:

gFitCharStr(string, x1, y1, x2, y2, fit)

When **fit** = GB2P, the string angle is adjusted so that the string lies along the arbitrary line between the two points. When **fit** = GSIZE the character size is also adjusted so that the string exactly fits between the two points (other values of **fit** are reserved for future development).



Strings adjusted to fit between two points

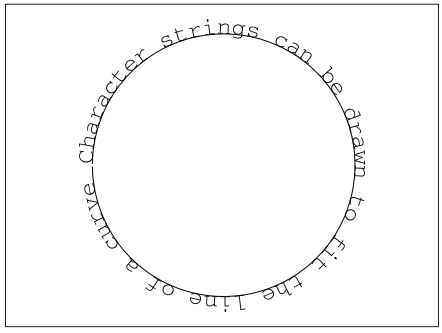
Character Strings Drawn Along a Curve

Two routines are provided to place character strings along the line of a curve:

gDisplayStrPolylineTo2D(npts, points, string)

gDisplayStrPolylineBy2D(npts, points, string)

The current character size and justification applies to the string which may result in the string being clipped.



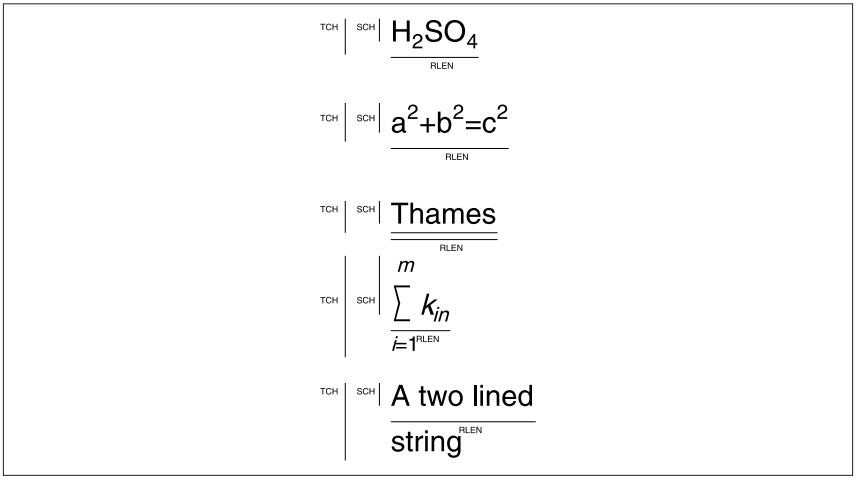
Example of character string drawn along a curve

Returning Information about a String

It is often useful to be able to determine the length of a character string before it is output. With fixed pitch fonts this can be calculated by multiplying the number of characters by the actual character width, but where the string contains proportional fonts or a number of escape sequences it is very difficult to calculate this value. The routine gReturnStrInfo() can be used to return the length and other information relating to a character string.

gReturnStrInfo(string, rlen, nnl, tch, sch, nesc)

where **string** is the input string to be enquired. **rlen** is the maximum length of that string in current units. **nnl** returns the number of lines contained in the string (which is normally 1 unless the *N escape sequence has been used). **tch** returns the total character height, which always includes space for descenders and underlining and can include space for indices and multiple lines if the string contains them. **sch** returns the maximum height above the base line of the string. **nesc** returns the number of characters discounting the escape sequences in the string.



Information returned by gReturnStrInfo()

Country Specific Characters

GINO provides provision for outputting certain non-standard ASCII characters in either software or hardware character mode.

Euro Symbol

The Euro symbol has been added into software fonts 101 and 102 at position 127 and can be output either by inserting the character directly from the keyboard-key or <alt><key> sequence into the source code or by selecting font 101/102 and position 127 using the routines gSetCharFont() and gDisplayAsciiChar(). The hardware symbol can be output using the mwin, wogl, xwin, glx and postscript drivers by using fonts 100-108 and inserting the character directly from the keyboard-key or <alt><key> sequence. Note that the hardware Euro symbol will only appear on Windows 98 rel. 2 or later, with the DW-Motif Euro kit under OpenVMS, or with Postscript Level III firmware.

German Umlaut characters

The German umlaut characters have been added into various software fonts at positions 25-31 and can be output by inserting the character directly from the keyboard-key or <alt><key> sequence into the source code or by selecting the required font and character position using the routines gSetCharFont() and gDisplayAsciiChar(). The hardware symbols can be output using the mwin, wogl, xwin, glx and postscript drivers by using fonts 100-108 and inserting the character directly from the keyboard-key or <alt><key> sequence. The umlaut characters can also be output by using the GINO 'escape' character '*:' followed by a,o,u or s and this will produce the correct character either in hardware or software depending on the font and character mode selected.

Symbols

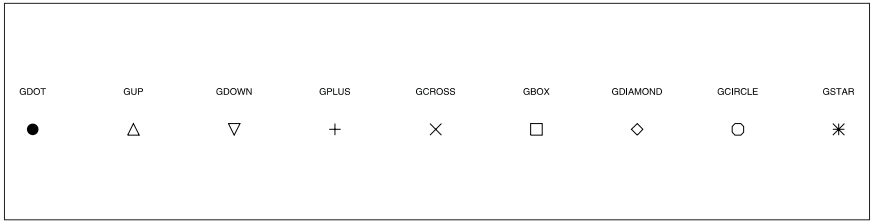
Any hardware or software character together with an additional 9 symbols may be output as a centred symbol by using the routine gDrawMarker():

gDrawMarker(nsym)

where **nsym** is used to indicate the symbol number that is required. Apart from the first nine symbols, numbered 0 - 8, indicated below, other characters may be output with the gDrawMarker() routine using the symbol number printed in the bottom left corner in the character tables in Appendix C.

Symbols are drawn the same size and shape as characters defined by the routines gSetCharSize(), etc. However, if non-horizontal or italic symbols are required, GINO transformations must be used with software transformable characters in operation.

These are the first nine GINO symbols:



9 Pre-defined Symbols

Symbol 0 is a dot of radius half the character width.

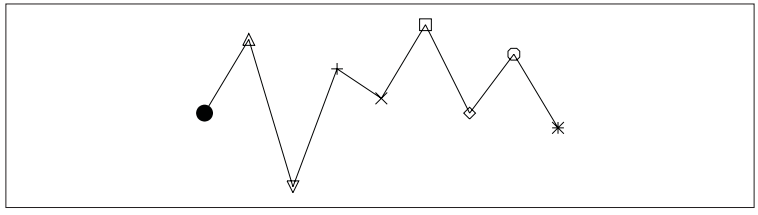
Note that the mode setting routines gSetHardChars(), gSetSoftChars() and gSetMixedChars() also affect the output of symbols.

Positioning Symbols

Symbols are centred about the start pen position, and the pen is left at the centre of the symbol. For example - the following statements join the nine points held in the arrays X and Y, drawing a different symbol at each point as shown in the figure below.

```
gSetCharSize(6.0,6.0);
gMoveTo2D(x[0],y[0]);
gDrawMarker(0);
for (n=1, n<9, n++) {
   gDrawLineTo2D(x[n],y[n]);
   gDrawMarker(n);
}
```

```
call gSetCharSize(6.0,6.0)
call gMoveTo2D(x(1),y(1))
call gDrawMarker(0)
do n=2,9
  call gDrawLineTo2D(x(n),y(n))
  call gDrawMarker(n-1)
end do
```



Multiple Symbols

Routines for drawing multiple symbols in 2D or 3D are:

gDrawPolymarkerTo2D(npts, points2, nsym)

gDrawPolymarkerBy2D(npts, points2, nsym)

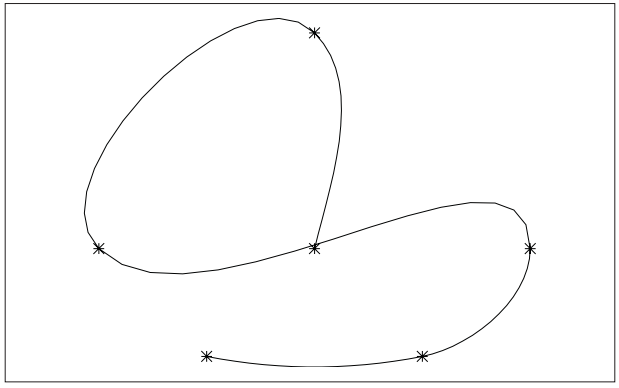
gDrawPolymarkerTo3D(npts, points3, nsym)

gDrawPolymarkerBy3D(npts, points3, nsym)

The following example draws an asterisk (symbol number 8) at each of 6 points stored in the **pt** array and joins them with an Akima curve:

```
gDrawPolymarkerTo2D(6,pt,8);
gDrawAkimaTo2D(6,pt,GNONE,GNONE);
```

```
call gDrawPolymarkerTo2D(6,pt,8)
call gDrawAkimaTo2D(6,pt, &
       GNONE,GNONE)
```



Multiple Symbol Drawing

Chapter 7

AREA FILLING

Area Filling Introduction

GINO provides area filling facilities for simple and complex polygons using solid fill or various hatch styles. Simple polygons are defined as a series of points, either singly or in sets, in 2D space, Up to a limit of 2048 points. These are described in this chapter. Complex polygons are constructed using the 2D or 3D drawing routines (lines, curves etc.)and stored in internal workspaces. They can be given identifiers for selective filling and picking and can be used for defining polygonal windows and/or masks. These are described later in this document (see page 245).

The appearance of the fill for all these routines depends on the hatching style and the line style in which it is drawn. GINO offers 16 default hatch styles (out of a possible 256), all of which may be redefined by the user.

A third type of polygon, called a facet, is also available in GINO for use with lighting and shading options (see page 295).

Filling a Rectangle

The routine for filling rectangles is :

gFillRect(fill, line, limit)

The fill style determines how the rectangular area is filled. If **fill** = GSOLID, the area is solid filled by hardware or by horizontal lines drawn touching each other. If **fill** = GBOUNDARY, only the boundary outline is drawn. Other default hatch fill styles are shown later in this section. The style of the hatch line is selected by **line** from the line definition table. If **line** = GCURRENT, the current line style is selected. Line styles can modify the appearance of a hatch style. This is particularly apparent with the attributes broken line type and colour.

For example, a solid fill by software (**fill** = GSOLID) is changed if a broken line type is selected. If **line** is set out of range (i.e. greater than 256), the filling line style defaults to the current line. A solid fill also results if **fill** is set out of range (i.e. greater than 256).

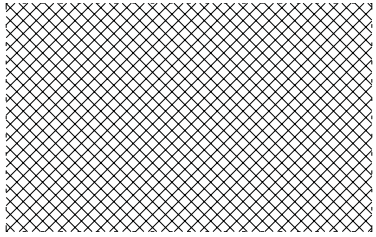
The diagonally opposite corners of the rectangle are specified by the structure **limit**. These are picture space coordinates. **limit.xmin** may be greater than **limit.xmax** and **limit.ymin** may be greater than **limit.ymax**.

For example, the statements:

```
static GLIMIT limit =
       {50.0,130.0,50.0,100.0};

gFillRect(6,GCURRENT,&limit);
```

```
type (GLIMIT) :: limit = &
    GLIMIT(50.0,130.0,50.0,100.0)

call gFillRect(6,GCURRENT,limit)
```

would produce a rectangle filled with hatch style 6 (see default hatch styles table) in the current line style.

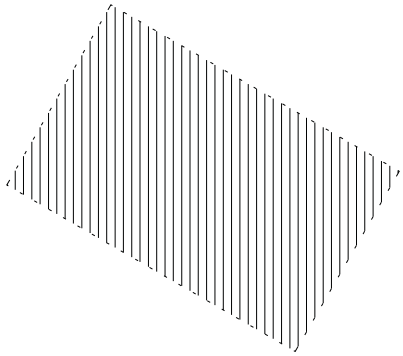


A Filled Rectangle

The rectangle, like other output primitives, is subject to the current transformation, which if any rotations of scales are present, will result in a non-rectangular shape. For example, if a call to gFillRect() is made when a rotation of -30.0 degrees is the current transformation, the area shown in the figure below will be filled thus:

```
static GLIMIT limit =
       {0.0,80.0,100.0,160.0};

gRotate2D(-30.0);
gFillRect(GHOLLOW,3,&limit);
gFillRect(2,GCURRENT,&limit)
```

```
type (GLIMIT) :: limit = &
    GLIMIT(0.0,80.0,100.0,160.0)

call gRotate2D(-30.0)
call gFillRect(GHOLLOW,3,limit)
call gFillRect(2,GCURRENT,limit)
```



The Effect of Transformation on gFillRect

Filling Single Polygons

The routines for filling single polygons are:

gFillPolygonTo2D(fill, line, inv, npts, points)

gFillPolygonBy2D(fill, line, inv, npts, points)

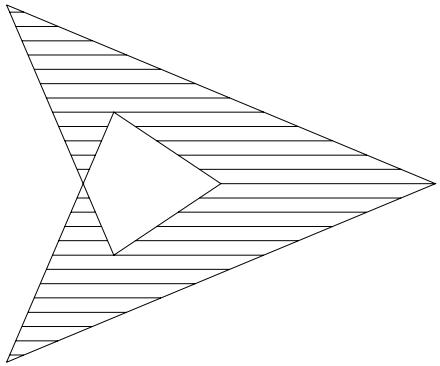
The routines gFillPolygonTo2D() and gFillPolygonBy2D() fill a single polygon with **npts** vertices. Each polygon includes the current pen position and either the absolute or relative points in the array of **points** being structures of GPOINT. An extra point is added if necessary to ensure the polygon is closed before filling. The fill style and line style are defined in the same way as gFillRect(), using **fill** and **line** arguments.

inv specifies which area is to be filled. When **inv**=GAREA the interior of the polygon is filled and when **inv**=GINVERSE the exterior area up to the current window limits is filled, leaving the interior empty. If the polygon is self intersecting, unfilled areas can be created within a polygon.

The following examples show the application of 2-D polygon filling:

```
static GPOINT pts[6] =
    {300.0,200.0, 180.0,150.0,
     210.0,220.0, 240.0,200.0,
     210.0,180.0, 180.0,250.0};
      .
      .
gMoveTo2D(180.0,250.0);
gDrawPolylineTo2D(6,pts);
gFillPolygonTo2D(1,4,GAREA,
      6,pts);
```

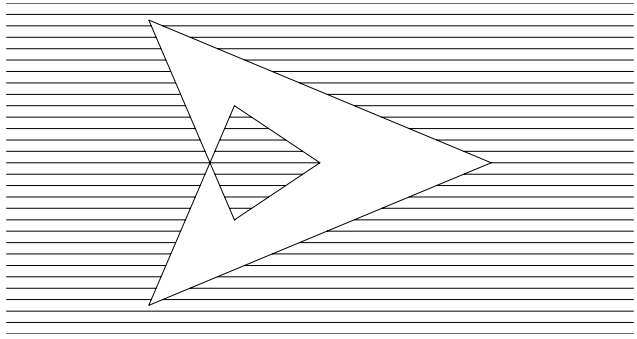
```
type (GPOINT) :: pts(6) = (/ &
      GPOINT(300.0,200.0), &
      GPOINT(180.0,150.0), &
      GPOINT(210.0,220.0), &
      GPOINT(240.0,200.0), &
      GPOINT(210.0,180.0), &
      GPOINT(180.0,250.0} /)
      .
call gMoveTo2D(180.0,250.0)
call gDrawPolylineTo2D(6,pts)
call gFillPolygonTo2D(1,4,GAREA,&
            6,pts)
```



Polygon Fill with Normal Fill

```
gMoveTo2D(180.0, 250.0);
gDrawPolylineTo2D(6,pts);
gFillPolygonTo2D(1,4,GINVERSE,
      6,pts);
```

```
call gMoveTo2D(180.0, 250.0)
call gDrawPolylineTo2D(6,pts)
call gFillPolygonTo2D(1,4, &
      GINVERSE,6,pts)
```



Polygon Fill with Inverse Fill

Filling Polygon Sets

Polygon Set Definition

A polygon set consists of an array of polygons each of which consists of an integer number of vertices and a pointer to an array of 2D vertices. The storage of such a 2D polygon set requires the following data structure in either C/C++ or Fortran 90.

```
typedef struct {
   int nvert;
   GPOINT *verts;
} GPOLYGON;
```

```
type GPOLYGON
   sequence
   int :: nvert
   type (GPOINT),dimension(:), &
        pointer :: verts
end type
```

Each polygon is complete within itself and does not make use of the current pen position. For this reason polygon sets can only use absolute coordinates. GINO ensures that every polygon is closed by adding the first point to the end of each polygon if it is not contained in the definition supplied by the application program.

An example of a 2-D polygon set consisting of a trapezium and two triangles is shown in the diagram below:

1 2 3 4 5 6 7 8 9 10

```
x:  40.   160.  340.  460.      120.  245.  245.      250.  440.  250.
y:  140.  40.   40.   140.      145.  270.  145.      145.  145.  335.

<----------------------------------> <-------------------------> <---------------------->

Polygon sizes
                4                       3                       3
```

Polygon Usage

Two dimensional polygon sets are filled using the following routine:

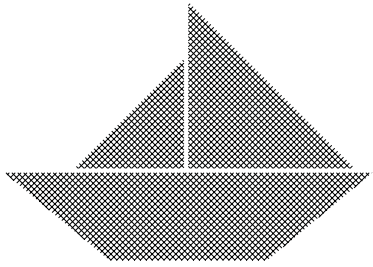
gFillPolygonSet2D(fill, line, inv, npol, polygons)

GINO applies the current viewing transformations to the polygon to create a single polygon which is filled the same way as normal 2-D filling. Therefore if any polygon within the set overlaps any other, or itself, then the generated polygon could be windowed.

The fill style and line style are defined in the same way as gFillRect(), using **fill** and **line** arguments.

The argument **inv** specifies which areas are to be filled once the single polygon has been generated. When **inv** = GAREA the interior of the polygon is filled, and when **inv** = GINVERSE the exterior area up to the current window limits is filled, leaving the interior empty.

The example polygon sets described previously can be implemented as follows.



Filled polygon set

C code

```
#include gino-c.h

GPOLYGON poly[3] = {4, 0, 3, 0, 3, 0};
GPOINT points[10] = {
      40.0,140.0, 160.0,40.0, 340.0,40.0, 460.0,140.0,
      120.0,145.0, 245.0,270.0, 245.0,145.0,
      250.0,145.0, 440.0,145.0, 250.0,335.0};

   poly[0].verts=&points[0];
   poly[1].verts=&points[4];
   poly[2].verts=&points[7];

   gFillPolygonSet2D(6,1,GAREA,3,poly);
```

F90 code

```
use gino_f90

type (GPOLYGON) :: poly(3)
type (GPOINT)   :: points(10) = {/ &
         GPOINT(40.0,140.0), GPOINT(160.0,40.0), &
         GPOINT(340.0,40.0), GPOINT(460.0,140.0), &
         GPOINT(120.0,145.0),GPOINT(245.0,270.0), &
         GPOINT(245.0,145.0), GPOINT(250.0,145.0), &
         GPOINT(440.0,145.0), GPOINT(250.0,335.0) /)

   poly(1)%nvert=4
   poly(1)%verts=>points(1:4)
   poly(2)%nvert=3
   poly(2)%verts=>points(5:7)
   poly(3)%nvert=3
   poly(3)%verts=>points(8:10)
   .
   gFillPolygonSet2D(6,1,GAREA,3,poly)
```

Filling Modes

Depending on capabilities of the graphics device the filling mode can be switched between hardware and software with the following routine:

gSetFillMode(sw)

By default the device is requested to execute the fill in hardware (because hardware is more efficient) but if it cannot, GINO will fill the polygon with a series of lines. This default case corresponds to a call to gSetFillMode(GHARD) and the results may depend on the device. To force a software fill every time, a call to gSetFillMode(GSOFT) should be made.

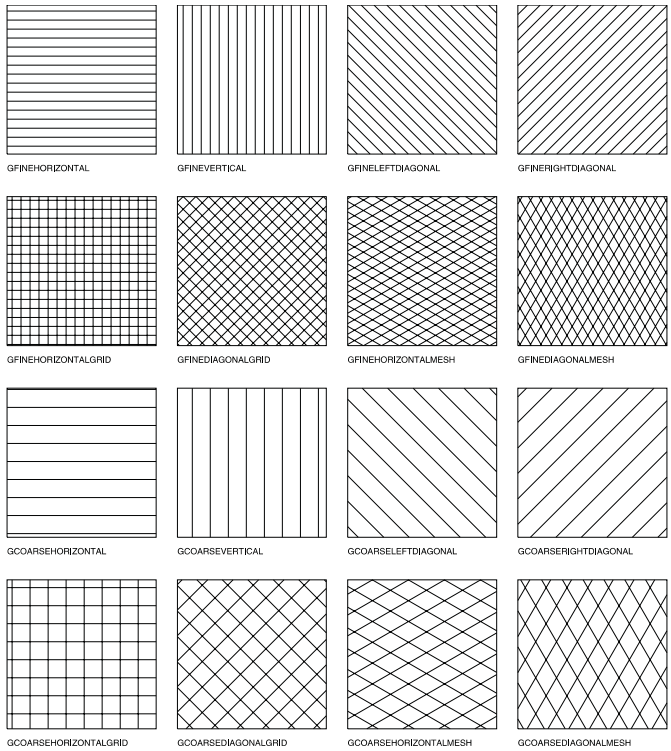
Hatch Style Definition

gDefineHatchStyle(fill, rep)

gEnqHatchStyle(fill, rep)

Up to 256 hatch styles (**fill**=1 to 256) may be defined and stored in GINO's hatch style table. This table is initialized with the set of 16 values shown in the table below (repeated throughout the 256 entries) each time an output device is nominated. These styles, when drawn with a solid line, appear as in the figure below the table which displays the hatch styles.

no.	name	pitch	angle	xshift	yshift	xshear	xhatch
1	GFINEHORIZONTAL	2.0	0.0	0.0	0.0	0.0	0
2	GFINEVERTICAL	2.0	90.0	0.0	0.0	0.0	0
3	GFINELEFTDIAGONAL	2.0	-45.0	0.0	0.0	-45.0	0
4	GFINERIGHTDIAGONAL	2.0	45.0	0.0	0.0	45.0	0
5	GFINEHORIZONTALGRID	2.0	0.0	0.0	0.0	0.0	1
6	GFINEDIAGONALGRID	2.0	45.0	0.0	0.0	0.0	1
7	GFINEHORIZONTALMESH	2.0	-30.0	0.0	0.0	30.0	1
8	GFINEDIAGONALMESH	2.0	60.0	0.0	0.0	30.0	1
9	GCOARSEHORIZONTAL	4.0	0.0	0.0	0.0	0.0	0
10	GCOARSEVERTICAL	4.0	90.0	0.0	0.0	0.0	0
11	GCOARSELEFTDIAGONAL	4.0	-45.0	0.0	0.0	-45.0	0
12	GCOARSERIGHTDIAGONAL	4.0	45.0	0.0	0.0	45.0	0
13	GCOARSEHORIZONTALGRID	4.0	0.0	0.0	0.0	0.0	1
14	GCOARSEDIAGONALGRID	4.0	45.0	0.0	0.0	0.0	1
15	GCOARSEHORIZONTALMESH	4.0	-30.0	0.0	0.0	30.0	1
16	GCOARSEDIAGONALMESH	4.0	60.0	0.0	0.0	30.0	1



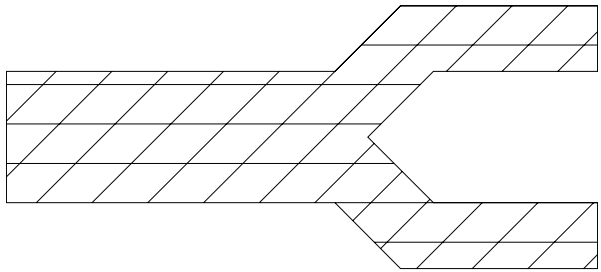
Default hatch styles

The user may redefine entries in the hatch styles table by calling gDefineHatchStyle(). For example:

```
static GHATSTY rep =
   {6.0,0.0,0.0,0.0,450.0,1};

gDefineHatchStyle(2,&rep);
```

```
type (GHATSTY) :: rep = &
 GBRKSTR(6.0,0.0,0.0,0.0,450.0,1)

call gDefineHatchStyle(2,rep)
```

would redefine table entry 2. If this hatch style was then selected by a polygon fill routine, using a solid line, the resulting fill would look like that shown in the figure below.



Hatch Style Definition

The structure GHATSTY has six elements which fully define hatch style: **rep.pitch**, **rep.angle**, **rep.xshift**, **rep.yshift**, **rep.xshear** and **rep.xhatch**. The elements are best understood in terms of the hatch origin and the local axes. The hatch origin is the origin of the local axes and the local axes are the picture axes rotated and shifted. See the figures below. In the following descriptions of the parameters, examples are drawn from the default values set in the hatch style table by GINO.

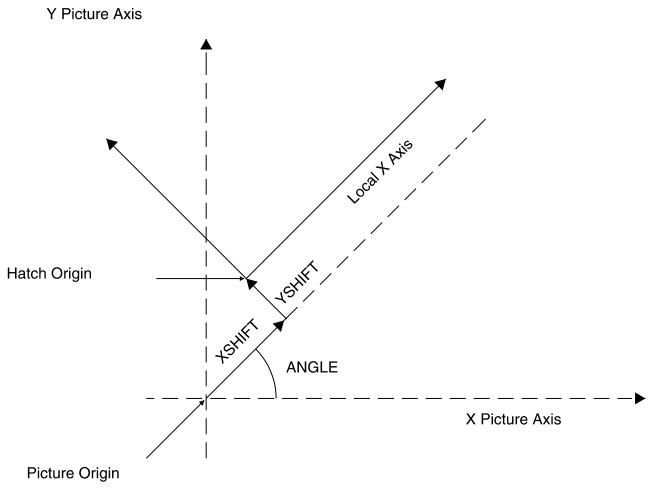
rep.pitch specifies the distance between hatch lines.

rep.angle is the rotation of the local axes about the picture origin. Positive rotation is anticlockwise.

rep.xshift moves the hatch origin in the direction of the local X axis. The effect of **rep.xshift** is apparent if cross hatching is selected.

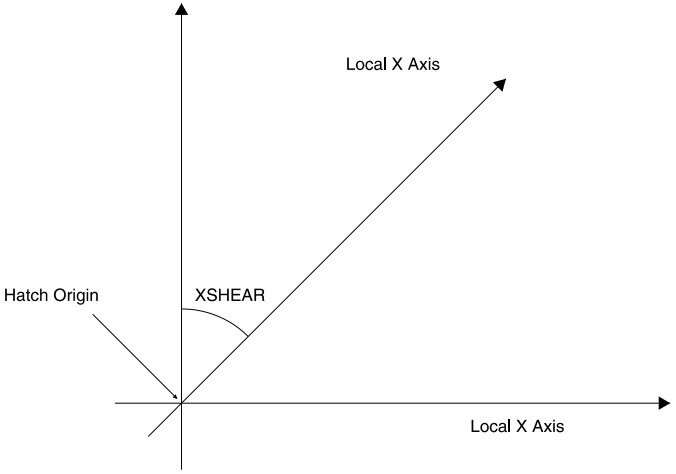
rep.yshift moves the hatch origin in a direction perpendicular to the X axis.

rep.xshift and **rep.yshift** may be used to align the hatching pattern within a polygon.



The effect of Angle and Shift

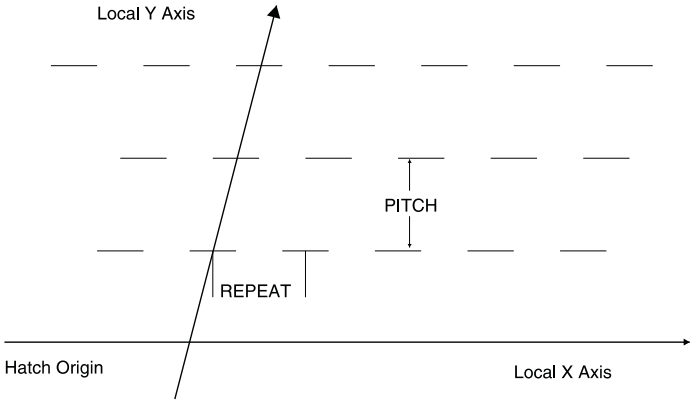
rep.xshear (see figure below) is the angle by which the local Y axis is sheared. **rep.yshift** remains perpendicular to the local X axis. Positive shear is clockwise. Shear is apparent only if cross hatching or a broken fill line is selected. When the broken line has continuous mode selected, the dashes will be centred about the local axes.



The effect of Shear

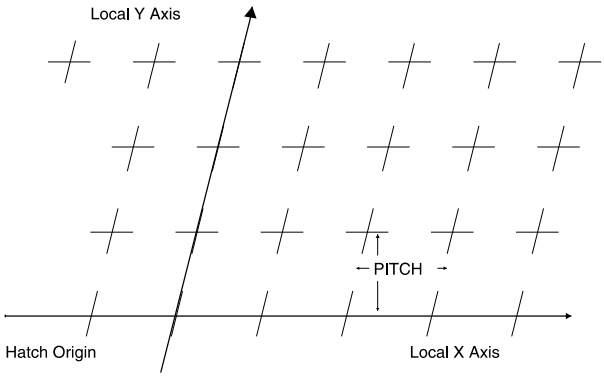
rep.xhatch switches cross hatching on and off. Hatch lines are drawn parallel to the local axes and start at the hatch origin.

Single hatch lines (**rep.xhatch**=GOFF) are drawn parallel to the local X axis.



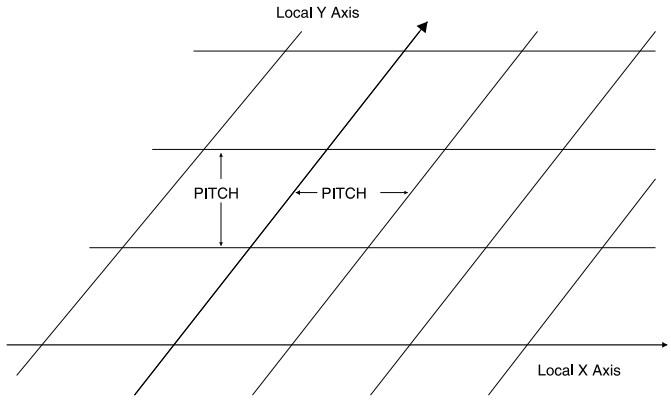
Single Hatch lines (continuous broken lines)

Cross hatching (**rep.xhatch**=GON) is drawn parallel to both local X and Y axes.



Cross Hatch lines (continuous broken lines)

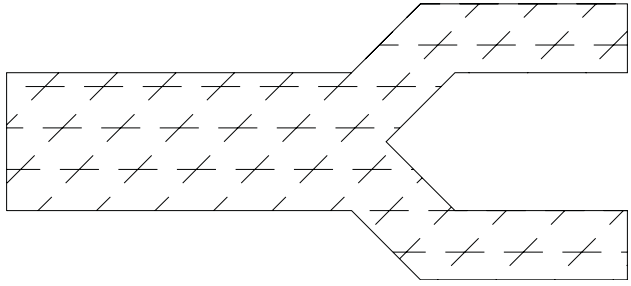
When **rep.xhatch**=GON both sets of hatch lines are drawn a distance **rep.pitch** apart. **rep.pitch** is the distance between hatch lines**.**



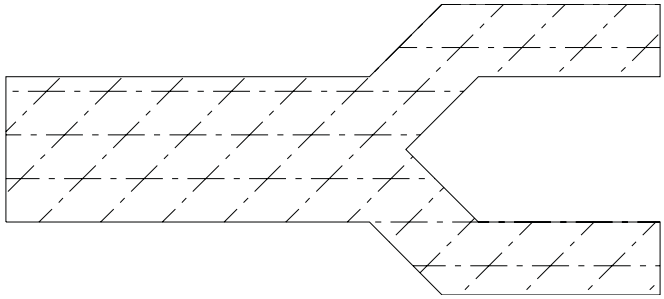
Pitch for cross hatching

A hatch pattern is generated with a hatch line coinciding with the local X axis. Subsequent lines are drawn parallel to the local X axis at a distance **rep.pitch** apart. If **rep.xhatch** =GON a second set of hatch lines are drawn parallel to the local Y axis (which may be sheared) with one of them coincident with the axis itself. Both sets of hatch lines may be thought to extend to infinity in both directions though in practice GINO stops where there are no more polygons to be filled or when clipping limits are reached.

When cross hatching with continuous mode broken lines (i.e. **brk.mode** = GCONTDASH or GCONTCHAIN, see gDefineBrokenLineStyle()), the repeat length is adjusted so that dashes bisect where the hatch lines cross each other. Dash and dot lengths are adjusted proportionally. See the figures below.



Cross hatching with broken line (mode=GCONTDASH)



Cross hatching with broken line (mode=GCONTCHAIN)

Compound hatching effects may be produced by several polygon fills of the same polygonal area. The hatch definition and line style may be changed for each fill. The two examples of GINO code at the end of this section produce the compound hatching effects in the figures below. Notice the following points:

a. Repeat and dash lengths of gDefineBrokenLineStyle() are related to the pitch distance to give horizontal and vertical symmetry to the patterns.

b. The bottom left hand corner of each polygon is a whole number of repeat lengths from the X and Y picture axis. The hatch pattern therefore appears to have its origin in this corner of the polygon. To change this, alter **rep.xshft** and **rep.yshft**.

c. The broken lines specified by gDefineBrokenLineStyle() are continuous. This means that the fill pattern is generated without reference to the edges of the polygon. If discontinuous mode is chosen, for example, change the contents of **brk.mode** from GCONTDASH to GDISCONTDASH in the broken line structure **brk** in Example 1 code; the fill would appear as in the figure below.

Example 1

C code

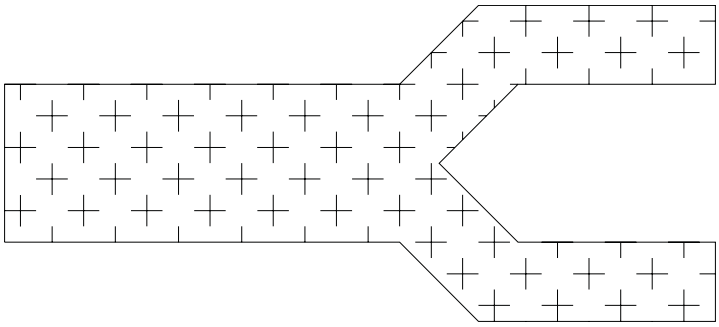
```
#include <gino-c.h>
#include <math.h>
#ifndef PI
#define PI 3.141592654
#endif
```

```
main ()
{
static GPOINT pts[13]= {
       30.0,80.0, 80.0,80.0, 90.0,90.0, 120.0,90.0,
       120.0,80.0, 95.0,80.0, 85.0,70.0, 95.0, 60.0,
       120.0,60.0, 120.0,50.0, 90.0,50.0, 80.0,60.0,
       30.0,60.0};

       GBRKSTY brk = { GCONTDASH, 8.0, 4.0, 0.0};
       GHATSTY hat = { 4.0, 0.0, 0.0, 0.0, 0.0, 0};

      gOpenGino();
/* Nominate device  */
      xxxxx();
      gNewDrawing();
/* Set software fill and broken line generation modes */
      gSetFillMode(GSOFT);
      gSetBrokenLineMode(GSOFT);
/* Define broken line style */
      gDefineBrokenLineStyle(7, &brk);
/* Redefine shear angle to match broken line repeat length */
      hat.xshear=(atan2(brk.repeat,2.0*hat.pitch))*180.0/PI;
/* Define 2 hatch styles */
      gDefineHatchStyle(1, &hat);
      hat.angle=90.0;
      gDefineHatchStyle(2, &hat);
/* Draw boundary and fill polygon */
      gMoveTo2D(30.,60.);
      gDrawPolylineTo2D(13, pts);
      gSetBrokenLine(7);
      gFillPolygonTo2D(1, GCURRENT, GAREA, 13, pts);
      gFillPolygonTo2D(2, GCURRENT, GAREA, 13, pts);
/* Close down device and GINO */
      gCloseDevice();
      gCloseGino();
}
```

F90 code

```
program hatch1
user gino_f90

type (GPOINT) :: pts(13)= (/ &
 GPOINT( 30.0,80.0), GPOINT( 80.0,80.0), GPOINT( 90.0,90.0),&
 GPOINT(120.0,90.0), GPOINT(120.0,80.0), GPOINT( 95.0,80.0),&
 GPOINT( 85.0,70.0), GPOINT( 95.0,60.0), GPOINT(120.0,60.0),&
 GPOINT(120.0,50.0), GPOINT( 90.0,50.0), GPOINT( 80.0,60.0),&
 GPOINT(30.0,60.0} /)
type (GBRKSTY) :: brk = GBRKSTY( GCONTDASH, 8.0, 4.0, 0.0)
type (GHATSTY) :: hat = GHATSTY( 4.0, 0.0, 0.0, 0.0, 0.0, 0)
integer :: PI = 3.141592654
      call gOpenGino
! Nominate device
      call xxxxx
      call gNewDrawing
! Set software fill and broken line generation modes
      call gSetFillMode(GSOFT)
      call gSetBrokenLineMode(GSOFT)
! Define broken line style
      call qDefineBrokenLineStyle(7, brk)
! Redefine shear angle to match broken line repeat length
      hat%xshear=(atan2(brk%repeat,2.0*hat%pitch))*180.0/PI
```

```
! Define 2 hatch styles
      call gDefineHatchStyle(1, hat)
      hat%angle=90.0
      call gDefineHatchStyle(2, hat)
! Draw boundary and fill polygon
      call gMoveTo2D(30.,60.)
      call gDrawPolylineTo2D(13, pts)
      call gSetBrokenLine(7)
      call gFillPolygonTo2D(1, GCURRENT, GAREA, 13, pts)
      call gFillPolygonTo2D(2, GCURRENT, GAREA, 13, pts)
! Close down device and GINO
      call gCloseDevice
      call gCloseGino
      stop
      end
```



Compound fill of example 1

Example 2

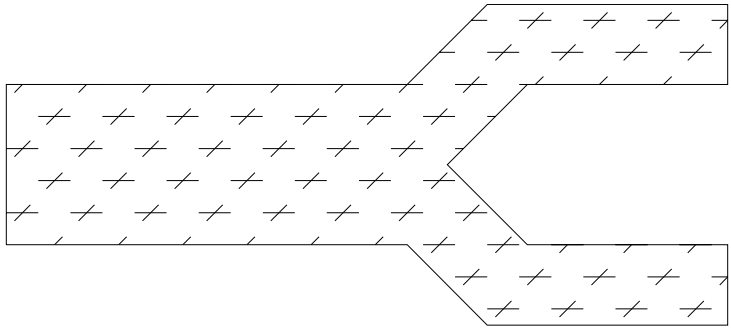
C code

```
GBRKSTY brk = { GCONTDASH, 8.0, 4.0, 0.0];
GHATSTY hat = { 4.0, 0.0, 0.0, 0.0, 0.0, 0};
.
.
   gDefineBrokenLineStyle(7, &brk);
/* redefine shear angle to match broken line repeat
   length */
   hat.xshear=(atan2(brk.repeat,2.0*hat.pitch))*180.0/PI;
/* define first hatch style */
   gDefineHatchStyle(1, &hat);
/* define second broken line style */
   brk.repeat=sqrt((brk.repeat**2.0)/2.0);
   brk.dash=brk.repeat/2.0;
   qDefineBrokenLineStyle(8, &rep);
/* define second hatch style */
   hat.pitch=brk.repeat;
   hat.angle=45.0;
   hat.xshear=0.0;
   gDefineHatchStyle(2, &hat);
```

```
/* draw boundary and fill polygon */
   gMoveTo2D(30.,60.);
   gDrawPolylineTo2D(13, pts);
   gSetBrokenLine(7);
   gFillPolygonTo2D(1, GCURRENT, GAREA, 13, pts);
   gSetBrokenLine(8);
   gFillPolygonTo2D(2, GCURRENT, GAREA, 13, pts);
   .
   .
```

F90 code

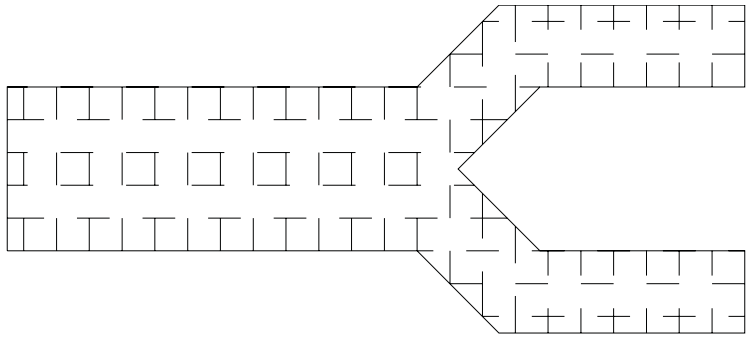
```
type (GBRKSTY) :: brk = GBRKSTY( GCONTDASH, 8.0, 4.0, 0.0)
type (GHATSTY) :: hat = GHATSTY( 4.0, 0.0, 0.0, 0.0, 0.0, 0)
.
.
      call gDefineBrokenLineStyle(7, brk)
! redefine shear angle to match broken line repeat
! length
      hat%xshear=(atan2(brk%repeat,2.0*hat%pitch))*180.0/PI
! define first hatch style
      call gDefineHatchStyle(1, hat)
! define second broken line style
      brk%repeat=sqrt((brk%repeat**2.0)/2.0)
      brk%dash=brk%repeat/2.0
      call gDefineBrokenLineStyle(8, rep)
! define second hatch style
      hat%pitch=brk%repeat
      hat%angle=45.0
      hat%xshear=0.0
      call gDefineHatchStyle(2, hat)
! draw boundary and fill polygon
      call gMoveTo2D(30.,60.)
      call gDrawPolylineTo2D(13, pts)
      call gSetBrokenLine(7)
      call gFillPolygonTo2D(1, GCURRENT, GAREA, 13, pts)
      call gSetBrokenLine(8)
      call gFillPolygonTo2D(2, GCURRENT, GAREA, 13, pts)
      .
      .
```



Compound fill of example 2

```
GBRKSTY brk = {
    GDISCONTDASH, 8.0, 4.0, 0.0];
```

```
type (GBRKSTY) :: brk = GBRKSTY(&
     GDISCONTDASH, 8.0, 4.0, 0.0)
```



Compound Fill of example 1 (discontinuous mode)

Hatch Style Enquiry

The routine gEnqHatchStyle() returns hatch style definitions from the software table of hatch styles.

The particular table entry of interest is identified by **fill**.

gEnqHatchStyle(fill, rep)

E.g:

```
GHATSTY hat;

gEnqHatchStyle(7,&hat);
```

```
type (GHATSTY) hat

call gEnqHatchStyle(7,hat)
```

returns the hatch style specified by entry 7.

Multiple Hatch Styles

Complex hatch patterns may be defined using multiple hatch styles and filling the area the same number of times as the number of hatch styles, as shown in the examples below:

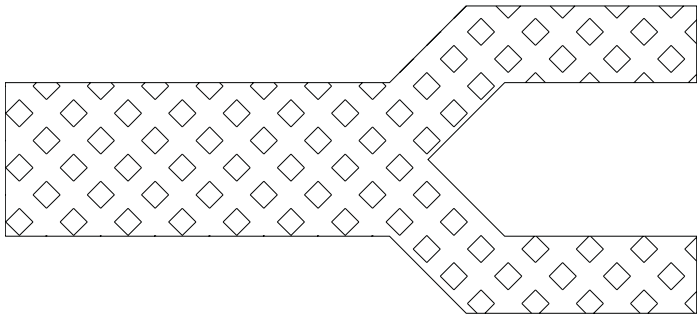
Box Hatch style

C Code

```
GHATSTY hatch1 = {2.5,45.0,1.25,0.0,0.0,0};
GHATSTY hatch2 = {2.5,135.0,1.25,0.0,0.0,0};
GBRKSTY brk1 = {GCONTDASH,5.0,2.5,0.0};
.
.
      gDefineHatchStyle(1,&hatch1);
      gDefineHatchStyle(2,&hatch2);
      gDefineBrokenLineStyle(1,&brk1);

      gMoveTo2D(30.,60.);
      gDrawPolylineTo2D(13,pts);
      gSetBrokenLine(1);
      gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts);
      gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts);
```

F90 Code

```
type (GHATSTY) :: hatch1 = GHATSTY(2.5,45.0,1.25,0.0,0.0,0)
type (GHATSTY) :: hatch2 = GHATSTY(2.5,135.0,1.25,0.0,0.0,0)
type (GBRKSTY) :: brk1 = GBRKSTY(GCONTDASH,5.0,2.5,0.0)
.
.
      call gDefineHatchStyle(1,hatch1)
      call gDefineHatchStyle(2,hatch2)
      call gDefineBrokenLineStyle(1,brk1)
!
      call gMoveTo2D(30.,60.)
      call gDrawPolylineTo2D(13,pts)
      call gSetBrokenLine(1)
      call gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts)
      call gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts)
!
```



Box Hatch Style

Brick Hatch Style

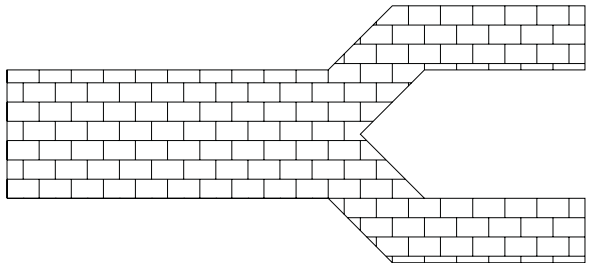
C Code

```
GHATSTY hatch1 = {3.0,0.0,0.0,0.0,0.0,0};
GHATSTY hatch2 = {2.5,90.0,1.5,0.0,0.0,0};
GBRKSTY brk1 = {GCONTDASH,6.0,3.0,0.0};
.
.
      dr=0.017453293;
/* Brick shear */
      hatch2.xshear=atan(3.0/2.5)/dr;
      gDefineHatchStyle(1,&hatch1);
      gDefineHatchStyle(2,&hatch2);
      gDefineBrokenLineStyle(1,&brk1);

      gMoveTo2D(30.,60.);
      gDrawPolylineTo2D(13,pts);
      gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts);
      gSetBrokenLine(1)
      gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts);
```

F90 Code

```
type (GHATSTY) :: hatch1 = GHATSTY(3.0,0.0,0.0,0.0,0.0,0)
type (GHATSTY) :: hatch2 = GHATSTY(2.5,90.0,1.5,0.0,0.0,0)
type (GBRKSTY) :: brk1 = GBRKSTY(GCONTDASH,6.0,3.0,0.0)
.
.
      dr=0.017453293
! Brick shear
      hatch2%xshear=atan(3.0/2.5)/dr
      call gDefineHatchStyle(1,hatch1)
      call gDefineHatchStyle(2,hatch2)
      call gDefineBrokenLineStyle(1,brk1)
!
```

```
call gMoveTo2D(30.,60.)
call gDrawPolylineTo2D(13,pts)
call gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts)
call gSetBrokenLine(1)
call gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts)
```



Brick Hatch Style

Honeycomb Hatch Style

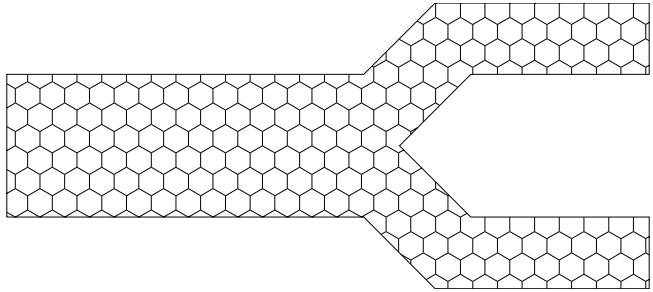
C Code

```
GHATSTY hatch1 = {1.732,30.0,1.0,0.0,0.0,0};
GHATSTY hatch2 = {1.732,-30.0,-1.0,0.0,0.0,0};
GHATSTY hatch3 = {1.732,90.0,-1.0,0.0,0.0,0};
GBRKSTY brk1 = {GCONTDASH,6.0,2.0,0.0};
.
.
```

```
      dr=0.017453293;
/*! Honeycomb shear */
      hatch1.xshear=atan(1.732)/dr;
      hatch2.xshear=atan(1.732)/dr;
      hatch3.xshear=atan(1.732)/dr;

      call gDefineHatchStyle(1,&hatch1);
      call gDefineHatchStyle(2,&hatch2);;
      call gDefineHatchStyle(3,&hatch3)
      call gDefineBrokenLineStyle(1,&brk1);

      call gMoveTo2D(30.,60.);
      call gDrawPolylineTo2D(13,pts);
      call gSetBrokenLine(1);
      call gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts);
      call gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts);
      call gFillPolygonTo2D(3,GCURRENT,GAREA,13,pts);
```

F90 Code

```
type (GHATSTY) :: hatch1 = GHATSTY(1.732,30.0,1.0,0.0,0.0,0)
type (GHATSTY) :: hatch2 = GHATSTY(1.732,-30.0,-1.0,0.0,0.0,0)
type (GHATSTY) :: hatch3 = GHATSTY(1.732,90.0,-1.0,0.0,0.0,0)
type (GBRKSTY) :: brk1 = GBRKSTY(GCONTDASH,6.0,2.0,0.0)
.
.
!
      dr=0.017453293
! Honeycomb shear
      hatch1%xshear=atan(1.732)/dr
      hatch2%xshear=atan(1.732)/dr
      hatch3%xshear=atan(1.732)/dr
!
      call gDefineHatchStyle(1,hatch1)
      call gDefineHatchStyle(2,hatch2)
      call gDefineHatchStyle(3,hatch3)
      call gDefineBrokenLineStyle(1,brk1)
!
      call gMoveTo2D(30.,60.)
      call gDrawPolylineTo2D(13,pts)
      call gSetBrokenLine(1)
      call gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts)
      call gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts)
      call gFillPolygonTo2D(3,GCURRENT,GAREA,13,pts)
!
```



Honeycomb Hatch Style

Trellis Hatch Style

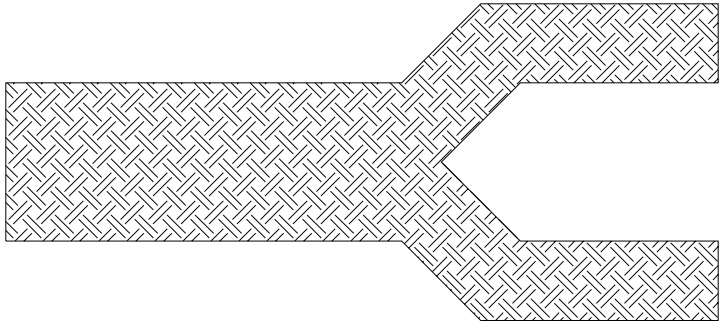
C Code

```
GHATSTY hatch1 = {2.5,45.0,0.0,-0.4,45.0,0};
GHATSTY hatch2 = {2.5,45.0,0.0,0.4,45.0,0};
GHATSTY hatch3 = {2.5,135.0,2.5,-0.4,45.0,0};
GHATSTY hatch4 = {2.5,135.0,2.5,0.4,45.0,0};
GBRKSTY brk1 = {GCONTDASH,5.0,3.35,0.0};
.
.
      gDefineHatchStyle(1,&hatch1);
      gDefineHatchStyle(2,&hatch2);
      gDefineHatchStyle(3,&hatch3);
      gDefineHatchStyle(4,&hatch4);
      gDefineBrokenLineStyle(1,&brk1);

      gMoveTo2D(30.,60.);
      gDrawPolylineTo2D(13,pts);
      gSetBrokenLine(1);
      gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts);
      gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts);
      gFillPolygonTo2D(3,GCURRENT,GAREA,13,pts);
      gFillPolygonTo2D(4,GCURRENT,GAREA,13,pts);
```

F90 Code

```
type (GHATSTY) :: hatch1 = GHATSTY(2.5,45.0,0.0,-0.4,45.0,0)
type (GHATSTY) :: hatch2 = GHATSTY(2.5,45.0,0.0,0.4,45.0,0)
type (GHATSTY) :: hatch3 = GHATSTY(2.5,135.0,2.5,-0.4,45.0,0)
type (GHATSTY) :: hatch4 = GHATSTY(2.5,135.0,2.5,0.4,45.0,0)
type (GBRKSTY) :: brk1 = GBRKSTY(GCONTDASH,5.0,3.35,0.0)
.
.
```

```
            call gDefineHatchStyle(1,hatch1)
            call gDefineHatchStyle(2,hatch2)
            call gDefineHatchStyle(3,hatch3)
            call gDefineHatchStyle(4,hatch4)
            call gDefineBrokenLineStyle(1,brk1)
      !
            call gMoveTo2D(30.,60.)
            call gDrawPolylineTo2D(13,pts)
            call gSetBrokenLine(1)
            call gFillPolygonTo2D(1,GCURRENT,GAREA,13,pts)
            call gFillPolygonTo2D(2,GCURRENT,GAREA,13,pts)
            call gFillPolygonTo2D(3,GCURRENT,GAREA,13,pts)
            call gFillPolygonTo2D(4,GCURRENT,GAREA,13,pts)
      !
```



Trellis Hatch Style

Complex Polygonal Definition, Drawing and Filling

Complex multi-polygons with no limits to the number of vertices may be drawn and filled. The definition of these polygons, there use and other facilities are described later in this document (see page 245).

Chapter 8

IMAGE HANDLING

Image Handling Introduction

This section will describe the use and control of images within GINO. An image is a picture described by a two-dimensional array of colour values that is to be displayed in a rectangular area on the device. Routines are provided to address the device at the device dependent, pixel level (ie. the smallest addressable unit on the currently nominated device) or at the device independent cell array level.

Images may be written to or read from the device, copied from one area to another or read in from a number of external image metafile types (see page 73). Users should however refer to Appendix B to see if such image facilities are available on the device being used as not all facilities are available on all devices.

The following routines are described in this section:

Read/Write single pixels	gGetPixel()/gDrawPixel()
Read/Write images	gGetPixelArea()/gDrawPixelArea()
Draw cell array	gDrawCellArray()
Define pixel data characteristics	gDefinePixelPacking()
Return picture coordinate of pixel position	gEnqPosOfPixel()
Return pixel coordinate of picture position	gEnqPixelPos()
Define image transformation	gSetPixelTransform()
Replicate image rectangle	gSetPixelReplication()
Return pixel resolution of device	gEnqPixelResolution()
Enquire pixel scaling attributes	gEnqPixelAttribs()
Enquire pixel data characteristics	gEnqPixelPacking()
Switch image display characteristics	gSetPixelDisplayMode()

Copy pixel areas	gCopyPixelArea()

Pixel Coordinate System

The GINO pixel coordinate system has its origin in the upper left corner of the available drawing area. This means that the Y value is reversed in comparison to the regular GINO picture coordinate system. Values for pixel coordinates range from 0 to the maximum X value minus one and the maximum Y value minus one, for the resolution of the current device.

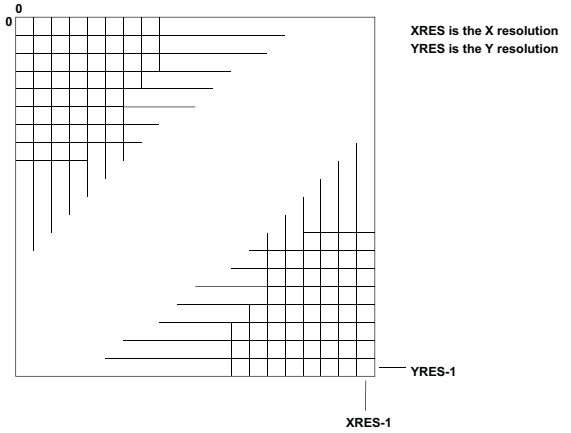


Image Coordinate System

The pixel resolution of the current device can be found using the routine:

gEnqPixelResolution(nxpix, nypix)

Pixel coordinate values are not affected by GINO transformations or window limits, and the display of images will be clipped to the current device limits. The drawing of pixel information will not affect the current pen position for GINO's drawing in picture coordinates.

Reading and Writing Single Pixels

Two basic routines are provided for the reading and writing of single pixels on a screen or printer:

gGetPixel(ix,iy,pix)

gDrawPixel(ix,iy,pix)

where **ix**,**iy** is the pixel coordinate to be read or written to, and **pix** is the pixel data. This may be a colour index value or, if the device is in direct colour mode (see page 46), a 24bit packed true colour value containing the required red, green and blue components.

Image Display

The two routines to display images are:

gDrawCellArray(x1, y1, x2, y2, npixx, npixy, isc, isr, idx, idy, pixbuf)

gDrawPixelArea(ix, iy, npixx, npixy, isx, isy, idx, idy, pixbuf)

The routine gDrawCellArray() draws the supplied image in a rectangular area where **x1**,**y1** and **x2**,**y2** specify the bottom left and top right corners in the current drawing units, whereas the routine gDrawPixelArea() draws the supplied image at the specified pixel position **ix**,**iy**.

Therefore in the former case the corner points represent two points in picture space which are transformed according to the current GINO transformation and viewing state to generate the corresponding points in pixel coordinates. The image is then scaled to fit the transformed points. Note that the image is always displayed in a rectangle with its sides parallel to the drawing area - the image is never skewed. This routine offers a simple way to display an image that fits a defined rectangle in a device independent way, at the expense of possible image distortion or loss of data.

In the latter case(gDrawPixelArea()) the point **ix**, **iy** represents the top left corner of the image and the image is displayed relative to this anchor point according to the current pixel transformation settings (see below). In the default case, each data value represents the colour of a single pixel and is displayed as such. However, unless special steps are taken the image will cover different sized areas on different devices according to its resolution.

Image Data

Image data is passed in a single-dimension 32 bit integer array, **pixbuf**, containing **npixx** * **npixy** words which may contain data in a variety of forms:

- 24 bit packed RGB values (one per word)
- Unpacked colour indices (one per word)
- Packed colour indices

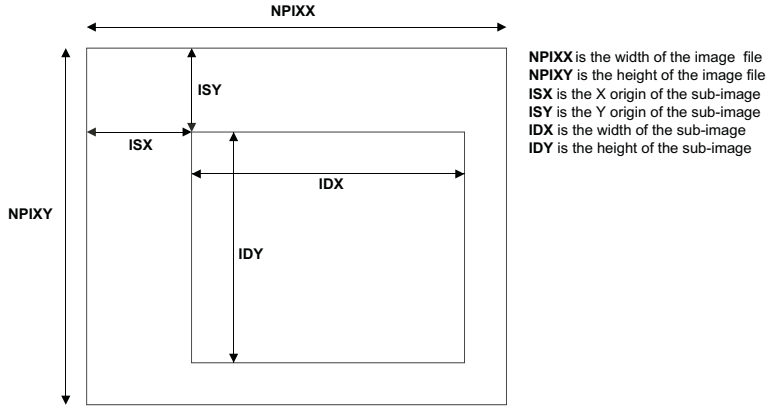
In the case of 24bit RGB triplets and unpacked colour indices, the **pixbuf** array will contain **npixx** * **npixy** data values, which when output using gDrawPixelArea() represent **npixx** * **npixy** pixels on the drawing area. Note that 24bit RGB triplets can only be used on devices operating in Direct Colour mode (see page 46) and are therefore not device independent.

Colour indices are pointers into the current GINO colour table and are usually up to 8bits in size. Whilst passing one index per word is rather wasteful in terms of storage, it if often the simplest way of handling image data. However, GINO also handles images containing packed colour indices of virtually any format (see below).

In all three cases the data is assumed to be passed row by row starting with the top left data value.

Sub Images

If only a subset of the image data is required then **isx** and **isy** are defined to be the start position of the subset within the image with **idx** and **idy** being the size of the subset. If the full image is required then **isx** and **isy** must be one, with **idx** and **idy** being equal to **npixx** and **npixy** respectively.



Coordinate System within an Image

C code

```
int pixbuf[64440];
:
gDrawPixelArea(0,0,358,180,1,1,358,180,pixbuf);
gDrawPixelArea(60,210,358,180,60,30,50,50,pixbuf);
gDrawPixelArea(230,240,358,180,230,60,128,100,pixbuf);
:
```

F90 code

```
integer pixbuf(64440)
:
call gDrawPixelArea(0,0,358,180,1,1,358,180,pixbuf)
call gDrawPixelArea(60,210,358,180,60,30,50,50,pixbuf)
call gDrawPixelArea(230,240,358,180,230,60,128,100,pixbuf)
:
```

The complete image and two sub images

Pixel Packing

Where colour indices are being used, the image may contain packed data or data stored in a non standard ordering. The required format is specified using the following routine:

gDefinePixelPacking(nbp, nrp, npw, ndir, dir)

The first four parameters determine how the pixel data bit information is packed within the integer array. **nbp** is the number of bits per pixel which is also known as the image depth. **nrb** is the number of relevant bits which represents how many bits within each pixel is to be used for displaying the image. **npw** represents the number of pixels that are represented within each integer word. Finally **ndir** specifies the direction that the bit information is oriented. The value +1 indicates a normal direction while -1 indicates the reverse direction. An example of these values is represented in the figure below which uses 4 byte words, giving a maximum of 32 bits that can be used within each integer.

00101100	00100111	00110101	00110011	00011101	00110011	00001110	00101101	NBP=8
44	39	53	51	29	51	14	45	NRB=6
First Word 740767027				Second Word 489885229				NPW=4 NDIR=1

Storage of pixel data within integer words

The values represent a format where there are four pixel values stored in each integer word. Each pixel has eight data bits but only the six least significant bits are to be used.

The variable **dir** specifies the order that the pixel data is to be accessed within the array to correctly display the image. It can take the value 1 to 8 which determines the start position and whether the data has been stored row by row or column by column.

dir = 1	Start top left with data accessed row by row (default)
dir = 2	Start top left with data accessed column by column
dir = 3	Start top right with data accessed row by row
dir = 4	Start top right with data accessed column by column
dir = 5	Start bottom left with data accessed row by row
dir = 6	Start bottom left with data accessed column by column
dir = 7	Start bottom right with data accessed row by row
dir = 8	Start bottom right with data accessed column by column

The figure below shows the output of **dir** on a square image file. Care should be taken when using this variable with non-square images, if the value of **npixx**, **npixy** is inappropriate then the image will become incomprehensible.

C code

```
int pixbuf[10000];
:
gDefinePixelPacking(8, 8, 1, 1, dir);
gDrawPixelArea(0,0,100,100,1,1,100,100,pixbuf);
:
```

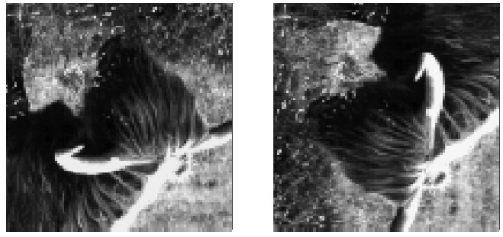
F90 code

```
integer pixbuf(10000)
:
call gDefinePixelPacking(8, 8, 1, 1, dir)
call gDrawPixelArea(0,0,100,100,1,1,100,100,pixbuf)
:
```

The value of **dir** that the user should use for scanned data depends on the scanning characteristics of the equipment used. The output effect of **dir** shown below only indicates what happens to a data file stored in the default format, when output with different values of **dir**. If the scanning equipment uses scans and stores the data in a format other than, 'first pixel top left with subsequent pixels following along the row', then the correct value of **dir** will need to be selected.


DIR=1(left) DIR=2(right)


DIR=3(left) DIR=4(right)



DIR=5(left) DIR=6(right)

DIR=7(left) DIR=8(right)

Image Display Mode

On some devices image operations can be slow. The following command will allow pixel images to be hidden:

gSetPixelDisplayMode(mode)

This routine is for use during the development of applications where the layout is initially more important than the actual image itself. The default value for **mode** is GON which will display the image if the device is capable of displaying it. A value of GOFF will turn off the image so that nothing will be displayed and a value of GBOUNDARY will also turn off the image but will draw a bounding rectangle to represent the image size and position. The output of errors and warnings applicable to the displaying of an image are not affected by this switch.

Pixel Coordinate Conversion

Whilst cell arrays are transformed and scaled according to the current GINO transformation, pixel areas are not. It is useful however to be able to convert between GINO's picture coordinate system and the device's pixel coordinate system in order to integrate the two image styles with each other and with other GINO graphics.

In order to convert values in the pixel coordinate system to the GINO picture coordinate system and vice versa, the routines gEnqPosOfPixel() and gEnqPixelPos() need to be used. Note that as the pixel coordinate system has its origin at the top left of the drawing area, there is not a direct scaling factor between the two coordinate systems.

gEnqPosOfPixel(ix, iy, point)

gEnqPixelPos(xsc, ysc, pix)

The routine gEnqPosOfPixel() accepts two integer pixel values in **ix** and **iy** and returns the corresponding picture coordinate in the structure **point**, while gEnqPixelPos() accepts two real values in **xsc** and **ysc**, representing a picture coordinate and returns the corresponding integer pixel coordinate in the structure **pix**. As pixel coordinates are integer values, **pix.ix** and **pix.iy** are integers and returned as rounded values. This will cause a rounding of values if gEnqPosOfPixel() is consequently called with the values returned from gEnqPixelPos().

Pixel Transformations

The following routines allow greater control of pixel images drawn using the routine gDrawPixelArea(). They do not affect images drawn using gDrawCellArray().

Pixel Rotation and Scaling

Where simple scaling and rotation of pixel images is required, the following routine can be used:

gSetPixelTransform(ori, xsca, ysca)

The variable **ori** will change the orientation of the image about the anchor point in steps of 90 degrees in an anti-clockwise direction. If a rectangular image is rotated by 90 or 270 degrees the X and Y dimensions are swapped (see figure below).

C code

```
int pixbuf[10000];
:
gSetPixelTransform(rot,1.0,1.0);
gDrawPixelArea(0,0,100,100,1,1,100,100,pixbuf);
:
```

F90 code

```
integer pixbuf(10000)
:
call gSetPixelTransform(rot,1.0,1.0)
call gDrawPixelArea(0,0,100,100,1,1,100,100,pixbuf)
:
```

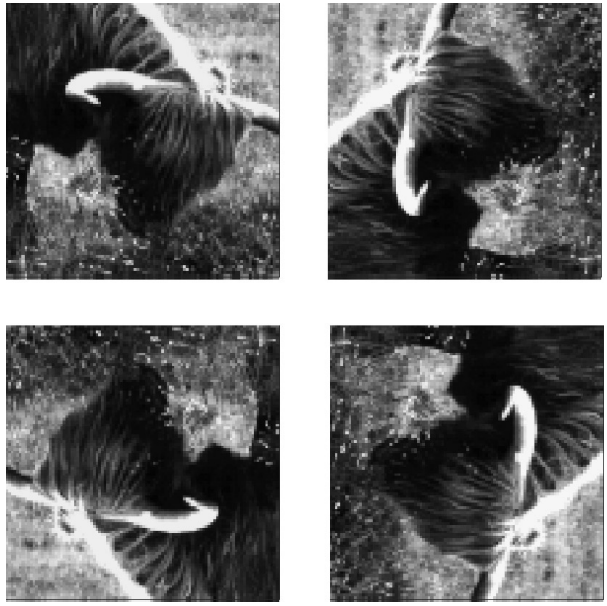


Image Rotation

The variables **xsca** and **ysca** are scaling factors. The default is 1.0 for both giving a 1 to 1 mapping. A value of 2.0 will double the size of the image in the specified direction by duplicating each pixel value. Real equivalents of integer scaling values will create a true scaled up image but it is possible to scale up by non-integer values or reduce the size of the image. Increasing the size of the image by non-integer scale values will create an image with added rows or columns at even intervals. If reducing the image, rows or column will be removed at even intervals and consequently information will be lost (see below).

C code

```
int pixbuf[10000];
:
gSetPixelTransform(0,0.5,0.5);
gDrawPixelArea(0,100,100,100,1,1,100,100,pixbuf);
gSetPixelTransform(0,1.0,1.0);
gDrawPixelArea(100,50,100,100,1,1,100,100,pixbuf);
gSetPixelTransform(0,1.5,1.5);
gDrawPixelArea(250,0,100,100,1,1,100,100,pixbuf);
gSetPixelTransform(0,1.0,1.5);
gDrawPixelArea(450,0,100,100,1,1,100,100,pixbuf);
gSetPixelTransform(0,11.0,1.0);
gDrawPixelArea(0,250,100,100,1,1,100,100,pixbuf);
```

F90 code

```
int pixbuf(10000)
:
call gSetPixelTransform(0,0.5,0.5)
call gDrawPixelArea(0,100,100,100,1,1,100,100,pixbuf)
call gSetPixelTransform(0,1.0,1.0)
call gDrawPixelArea(100,50,100,100,1,1,100,100,pixbuf)
call gSetPixelTransform(0,1.5,1.5)
call gDrawPixelArea(250,0,100,100,1,1,100,100,pixbuf)
call gSetPixelTransform(0,1.0,1.5)
call gDrawPixelArea(450,0,100,100,1,1,100,100,pixbuf)
call gSetPixelTransform(0,11.0,1.0)
call gDrawPixelArea(0,250,100,100,1,1,100,100,pixbuf)
```

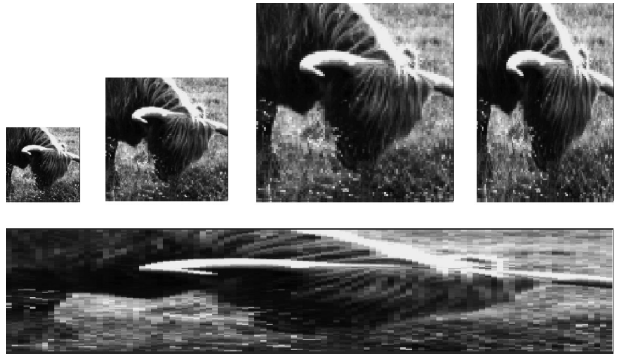


Image Scaling

Pixel Replication

An area can be defined to be filled with pixel data made up of multiple rectangular images. If the defined area is larger than the image rectangle output by gDrawPixelArea(), the image will be repeated to fill the space. If the defined area is smaller, the image will be appropriately clipped.

gSetPixelReplication(xrep, yrep)

where **xrep** and **yrep** are values in pixel coordinates which set the area that the image is to take up. If either of these values is zero then replication is switched off. This command will also change the direction of the image drawing relative to the anchor point depending on whether the values are positive or negative. Negative values result in the image display characteristics remaining unchanged, but the image is replicated in the opposite direction. Note that the images are not mirrored in any way.

C code

```
int pixbuf[10000];
:
gSetPixelReplication(250,250);
gDrawPixelArea(0,0,100,100,1,1,100,100,pixbuf);
gSetPixelReplication(100,-250);
gDrawPixelArea(300,250,100,100,1,1,100,100,pixbuf);
gSetPixelReplication(100,-200);
gDrawPixelArea(450,250,100,100,1,1,100,100,pixbuf);
gSetPixelReplication(100,-150);
gDrawPixelArea(600,250,100,100,1,1,100,100,pixbuf);
:
```

F90 code

```
integer pixbuf(10000)
:
call gSetPixelReplication(250,250)
call gDrawPixelArea(0,0,100,100,1,1,100,100,pixbuf)
call gSetPixelReplication(100,-250)
call gDrawPixelArea(300,250,100,100,1,1,100,100,pixbuf)
call gSetPixelReplication(100,-200)
call gDrawPixelArea(450,250,100,100,1,1,100,100,pixbuf)
call gSetPixelReplication(100,-150)
call gDrawPixelArea(600,250,100,100,1,1,100,100,pixbuf)
:
```

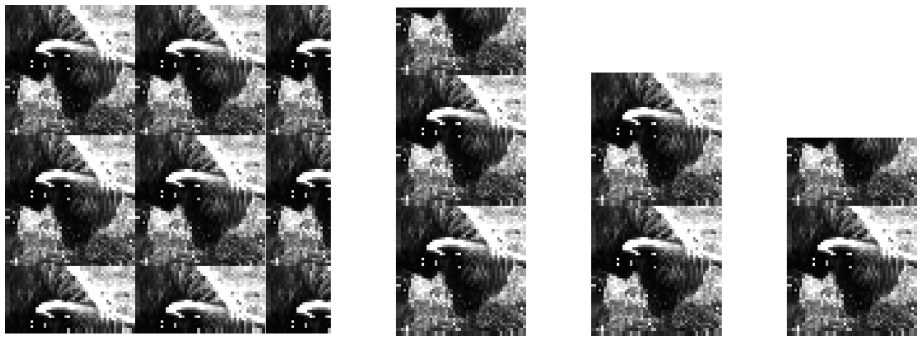


Image Replication

Pixel Enquiry Routines

The current pixel scaling attributes (as set by gSetPixelTransform() and gSetPixelReplication()) can be enquired through the following routine:

gEnqPixelAttribs(ori, xsca, ysca , xrep, yrep)

and the current pixel packing definition (as set by gDefinePixelPacking()) can be enquired through the following routine:

gEnqPixelPacking(nbp, nrp, npw, ndir, dir)

Reading Pixel Data

On some screen devices (see Appendix B), there is the facility to read images that are currently displayed on the screen. This is done with a command that is similar to gDrawPixelArea() but operates in reverse.

gGetPixelArea(ix, iy, npixx, npixy, isx, isy, idx, idy, pixbuf)

The variables **ix** and **iy** define the anchor point of a screen image that has a size represented by **npixx** and **npixy**. This is the image size that will be represented within the array **pixbuf**. The actual range of the image that is read is defined by **isx** and **isy** which is the start position of the subset within the image, with **idx**, **idy** being the size of the subset. This method will allow the selective overwriting of part of **pixbuf** which contains the full image.

The pixel rectangle must be defined within the device limits of the device or an error will occur and no reading will take place. The pixel transformation routines have no effect on the values passed to gGetPixelArea() (ie. replication and rotation settings do not apply).

Copying Pixel Images

Many devices that provide pixel facilities also include a facility to copy one pixel area to another. This facility can be accessed in GINO through the following routine:

gCopyPixelArea(source, dest, ix, iy, width, height, ixd, iyd)

where **source** and **dest** are the source and destination display identifiers. For copying areas on the primary display surface these should be set to 1, but in association with displays that provide multiple drawing areas (i.e. backing stores), pixel areas may be copied from any one to any other (see page 49).

The arguments **ix**,**iy** and **width**, **height** define the origin and dimensions of the area to be copied and **ixd**, **iyd** supply the final position of the origin. Pixel areas may be clipped if the copied region extends beyond the display limits.

Chapter 9

COLOUR DEFINITION

Colour Definition Introduction

The specification and use of colour in a GINO application varies significantly depending on the colour capabilities of the current device (i.e. whether the device is monochrome, has a static or dynamic colour table or operates in true colour mode (see page 46)). In order to provide a common interface across all these devices, GINO maintains a colour table which is initialized with a standard set of colours whenever a device is nominated. The colour of all graphical primitives may then be specified using indices into this colour table on any type of device.

Information in this section describes how GINO's colour table may be modified and the effect this has on the device as well as facilities for setting colour directly without going through the colour table (on devices that support this).

Colour Table

When a device is nominated a default colour table is initialized in GINO and the device itself (if possible). This table contains at least the following 11 entries:

Colour Table Index	Colour Constant	Colour
0	GBACKGROUND	Background (device dependent)
1	GBLACK	Black
2	GRED	Red
3	GORANGE	Orange
4	GYELLOW	Yellow

5	GGREEN	Green
6	GCYAN	Cyan
7	GBLUE	Blue
8	GMAGENTA	Magenta
9	GBROWN	Brown
10	GWHITE	White

Where a device has very few colours (i.e. some plotters and monochrome displays), the number of entries may be less than those defined above. Where a device has many more colours, these may or may not be initialized depending on the device driver (see Appendix B for information). Where a device does not have a colour table (i.e. true colour devices), GINO still maintains a colour table (of 1024 entries) which can be used in a GINO application.

In all cases the number of colour entries available on the currently nominated device can be obtained through the following routine:

gEnqColourInfo(ndc, ndt)

where **ndc** is the number of colour table entries available.

Display Types

The value of **ndt** returned by gEnqColourInfo() gives the type of colour device being used (see page 46) which also determines the effect of changing the contents of a colour table entry.

On fixed colour devices, (**ndt**=1) changes to the colour table are ignored as only a predefined number of colours are available through their colour number.

On static (**ndt**=2) and direct (**ndt**=4) colour devices, a colour table entry may be modified using the routines described below. When the particular colour index is re-used, graphical items will be drawn in the redefined colour.

On dynamic colour devices (**ndt**=3), changes to the colour table affect all graphical primitives that had been and will be drawn using the relevant colour index.

Colour Resolution

When entries in the colour table are modified (using the routines described below), both the GINO colour table and the devices' colour table (if present) are modified. However it should be noted that the values in both tables may not be exactly those requested depending on the colour resolution of the device.

The colour resolution is the range of colour values that can be stored on a particular device. For example, a colour device may only provide 4, 8 or 16 bits for storing colour values (whereas GINO uses real values in the range 0.0 to 1.0). This will restrict the range of colours that can be stored (and displayed).

When a change of colour is made to any device the actual colour used is returned to the GINO colour table so that these values may be enquired.

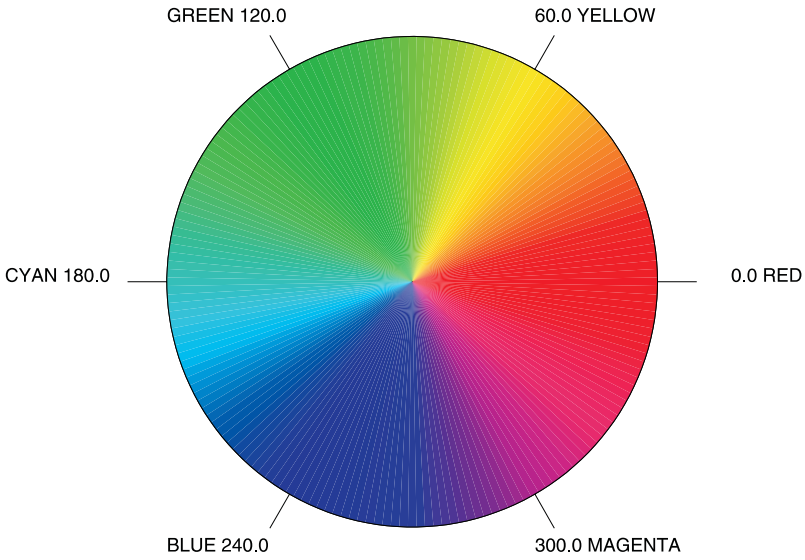
Colour Coordinate Systems

When a change is required in the colour table, generally three values are needed. These values are called the colour coordinates.

GINO supports three colour coordinate systems. These are RGB (red, green, blue), HSV (hue, saturation, value), and HLS (hue, lightness, saturation). GINO uses RGB as the standard system for communicating colour values to an output device. All colour values are converted to RGB. Any user-defined colour systems should be converted to RGB coordinates and call gDefineRGB().

The RGB system defines colour by the relative intensities of the red, green and blue primaries. However, people tend to perceive colour more in terms of intensity, strength and position in the colour spectrum, rather than as a mixture of primary colours.

Hue defines a colour's position in the spectrum. The spectrum is mapped onto an angular scale, 0.0 to 360.0 degrees, where red is at 0.0, green at 120.0 and blue at 240.0 degrees (see figure below). Angles outside the range 0.0 to 360.0 are treated modulo 360.0.



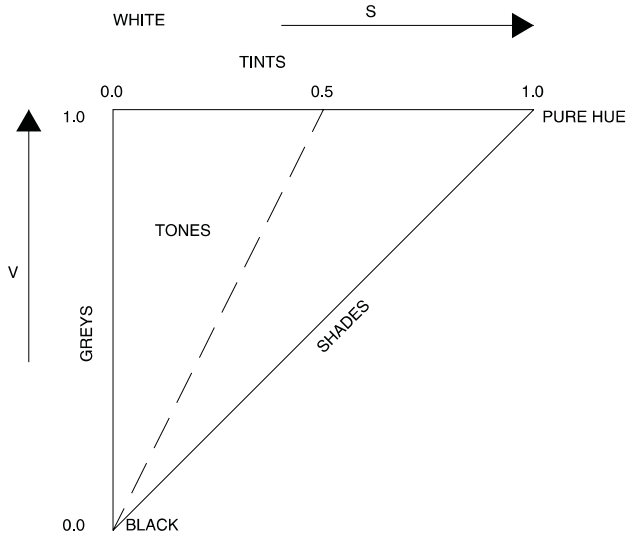
The Hue Spectrum

Saturation defines the strength or brightness of a colour (i.e. bright or pale tints). Saturation is expressed on a linear scale of 0.0 to 1.0 where 0.0 gives the palest hue and 1.0 the strongest.

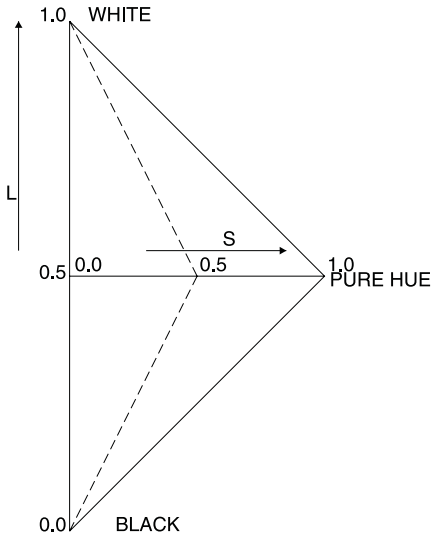
Value and lightness describe the intensity or lightness of a colour (i.e. light or dark shades). Value and lightness are both expressed on a linear scale of 0.0 to 1.0 but work in slightly different ways (see below). An attempt to define the coordinates saturation, lightness and value outside the range 0.0 to 1.0 causes the coordinates to be clipped so that values less than 0.0 become 0.0, and those greater than 1.0 become 1.0. GINO issues a warning message if this happens. The same applies for R,G, and B intensities. These are also defined on a linear scale of 0.0 to 1.0.

The differences between value (V) and lightness (L) is that with the HSV system, value gives the most intense hue when equal to 1.0. whereas with HLS, lightness gives the most intense hue when equal to 0.5 and gives white when equal to 1.0 regardless of any hue or saturation values (compare the figures below).

Each colour coordinate system also defines a greyscale.



Tints, Tones and Shades



The Lightness Scale

Conversion Between Coordinate Systems

The way colour values are transmitted, makes it possible to convert values from one coordinate system to another. Essentially this is a matter of defining a colour value in one system, and enquiring about it in another.

The conversion need not involve the colour values stored in the device driver (i.e. the values which drive the output device). By setting **icol** less than zero during definition, the conversion to RGB is made but it only gets stored internally and not passed to the device. If an enquiry is now made with the same (negative) **icol**, the enquiry routine will look at the internal storage and return the values it finds there.

For example:

C code

```
GHSVSTY hsv;
   gDefineRGB(-7,0.6,0.6,0.3);
   gEnqHSV(-7,&hsv);
   gDisplayStr("Coordinates in HSV:");
   gDisplayRealFloat(hsv.hue,10);
   gDisplayRealFloat(hsv.sat,10);
   gDisplayRealFloat(hsv.value,10);
```

F90 code

```
type (GHSVSTY) hsv
   gDefineRGB(-7,0.6,0.6,0.3)
   gEnqHSV(-7,hsv)
   gDisplayStr('Coordinates in HSV:')
   gDisplayRealFloat(hsv%hue,10)
   gDisplayRealFloat(hsv%sat,10)
   gDisplayRealFloat(hsv%value,10)
```

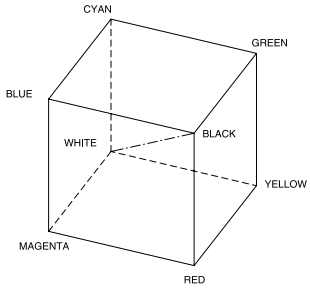
Colour 7 is defined in the RGB system and examined in the HSV system. The values returned by gEnqHSV() are output to the device. In this example, hsv.hue=60.0, hsv.sat=0.5, hsv.value=0.6.

If an enquiry is made with a negative **col** after device nomination but before any user redefinition, all coordinates return zero because the internal values have not yet been set to anything.

RGB Colour Coordinate System

The RGB (red, green, blue) colour coordinate system defines colours by specifying the relative intensities of the red, green and blue primaries. The resulting colour is therefore a mixture of the three primaries.

RGB colour space takes the form of a cube (see below).



Key: Represents the achromatic (grey) scale where R=G=B. For white, R=G=B=1.0. for black, R=G=B=0.0.

RGB Colour Cube

Using the RGB System

gDefineRGB(col, red, green, blue)

gEnqRGB(col, rgb)

Routine gDefineRGB() defines colours by specifying their red, green and blue intensities. An intensity of 0.0 means that none of the primary is present, and an intensity of 1.0 gives saturation. For example:

```
gDefineRGB(13,0.0,0.0,1.0);          call gDefineRGB(13,0.0,0.0,1.0)
```

would define colour 13 (**col**=13) as pure primary blue.

```
gDefineRGB(14,0.5,0.0,0.8);          call gDefineRGB(14,0.5,0.0,0.8)
```

would define colour 14 (**col**=14) as a light purple.

Whether or not these particular colours will actually be output (when selected by gSetLineColour() or gSetLineStyle()) depends upon the capabilities of the output device.

The routine gEnqRGB() returns the colour definition for the particular colour identified by **col** in the GRGBSTY structure.

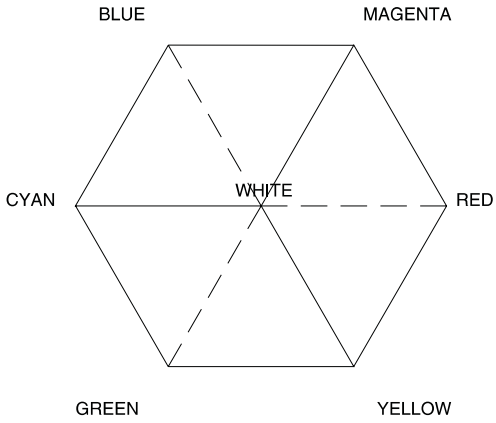
HSV Colour Coordinate System

The HSV (hue, saturation, value) system defines a colour by specifying its hue, saturation and value.

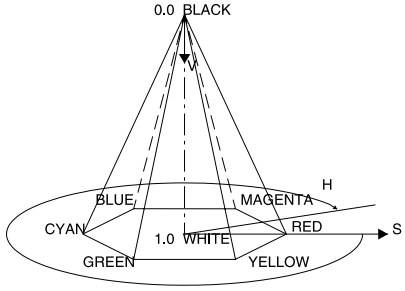
Value is a measure of the intensity or lightness of a colour. The brightest most intense hue occurs when saturation and value both equal 1.0.

HSV colour space may be considered as a single hexcone (see figures below).

It is instructive to compare the HSV (and the HLS) system with the RGB colour cube when this cube is viewed along its principal diagonal (compare the figures below).



RGB Colour Cube viewed along principal diagonal



Key: — · — Represents the achromatic scale where S=0.0.

Single Hexcone HSV Colour Model

Using the HSV System

gDefineHSV(col, hue, sat, value)

gEnqHSV(col, hsv)

The routine gDefineHSV() defines a colour by specifying its hue, saturation and value. The series of statements:

```
gDefineHSV(1,120.0,1.0,1.0);
gDefineHSV(2,140.0,1.0,1.0);
gDefineHSV(3,160.0,1.0,1.0);
gDefineHSV(4,180.0,1.0,1.0);
```

```
call gDefineHSV(1,120.0,1.0,1.0)
call gDefineHSV(2,140.0,1.0,1.0)
call gDefineHSV(3,160.0,1.0,1.0)
call gDefineHSV(4,180.0,1.0,1.0)
```

give a set of colour definitions which range from primary green through to cyan (i.e. a sequence of greens which become increasingly blue).

Saturation and value are both normalized onto a scale of 0.0 to 1.0. Maximum saturation occurs when **sat**=1.0. Thus the statement:

```
gDefineHSV(5,120.0,1.0,1.0);
```

```
call gDefineHSV(5,120.0,1.0,1.0)
```

would define colour 5 (**col**=5) as the brightest, most saturated primary green. The statement:

```
gDefineHSV(6,120.0,0.0,1.0);
```

```
call gDefineHSV(6,120.0,0.0,1.0)
```

would define colour 6 as white. Notice that when saturation is zero, hue is ignored.

The statements:

```
gDefineHSV(7,60.0,1.0,1.0);
gDefineHSV(8,60.0,0.8,1.0);
gDefineHSV(9,60.0,0.6,1.0);
gDefineHSV(10,60.0,0.4,1.0);
gDefineHSV(11,60.0,0.2,1.0);
```

```
call gDefineHSV(7,60.0,1.0,1.0)
call gDefineHSV(8,60.0,0.8,1.0)
call gDefineHSV(9,60.0,0.6,1.0)
call gDefineHSV(10,60.0,0.4,1.0)
call gDefineHSV(11,60.0,0.2,1.0)
```

give a series of colour definitions ranging from the brightest yellow through increasingly pale yellow tints.

The parameter **value** changes colour's position on the achromatic scale. Thus

```
gDefineHSV(12,120.0,1.0,0.8);
```

```
call gDefineHSV(12,120.0,1.0,0.8)
```

would give a dark primary green. If **value** is set at 0.0, the colour defined is black, regardless of the other parameter values.

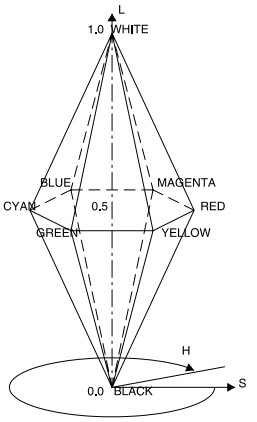
The stored colour definition may be examined by gEnqHSV() to return colour values converted to the HSV coordinate system.

HLS Colour Coordinate System

The HLS (hue, lightness, saturation) system defines colour in terms of its hue, lightness and saturation. Hue and saturation are the same as for the HSV system and lightness describes the intensity of a hue.

The lightness scale gives pure, most saturated hue when **light**=0.5 and **sat**=1.0. **light**=1.0 results in white, regardless of the settings of **hue** or **sat**. This should be contrasted with **value** in the HSV system.

HLS colour space may be represented as a double hexcone (see figure below).



HLS Double Hexcone

Using the HLS System

gDefineHLS(col, hue, light, sat)

gEnqHLS(col, hls)

The routine gDefineHLS() allows the user to define a colour in terms of its hue, lightness and saturation. At **light**=0.0 all colours merge into black and at **light**=1.0 all colours merge into white. With zero saturation, lightness describes an achromatic (grey) scale irrespective of any angle of hue which may be given.

In the following examples, lightness only is varied to give differing lightness of primary blue.

The statements

```
gDefineHLS(7,240.0,0.5,1.0);
gSetLineColour(7);
gMoveTo2D(200.0,0.0);
gDrawLineTo2D(200.0,250.0);
```

```
call gDefineHLS(7,240.0,0.5,1.0)
call gSetLineColour(7)
call gMoveTo2D(200.0,0.0)
call gDrawLineTo2D(200.0,250.0)
```

would define colour 7 as a pure, saturated primary blue, select it and draw a visible line to absolute coordinate position (200.0,250.0).

```
gDefineHLS(8,240.0,0.2,1.0);
```

```
call gDefineHLS(8,240.0,0.2,1.0)
```

would give a dark, saturated primary blue (analogous to mixing black and primary blue). The statement:

```
gDefineHLS(9,240.0,0.8,1.0);
```

```
call gDefineHLS(9,240.0,0.8,1.0)
```

would give a light saturated primary blue (analogous to mixing white with primary blue).

Direct Colour Control

On Direct Colour devices (which do not have a colour table) it is possible to define the colour of graphical primitives directly using a 24bit RGB triplet. This value contains the required quantities of Red Green and Blue in the one integer which is passed to the device through the gSetLineColour() routine. The function which generates the required value is:

col=gTrueCol(red, green, blue)

where **red**, **green** and **blue** are the required quantities of these colours. The function returns the 24bit RGB triplet containing 8 bits of data for each primary colour.

The following code shows how these two routines are used:

```
gSetLineColour(
   gTrueCol(0.8,0.2,0.5));
```

```
call gSetLineColour( &
   gTrueCol(0.8,0.2,0.5))
```

Here the current drawing colour is set to a bright shade of pink with 80% red, 20% green and 50% blue.

Chapter 10

MAPPING, WINDOWING AND MASKING

Mapping, Windowing and Masking Introduction

By default, all drawing is performed in the current device's paper coordinate system. This will usually be in millimetres (unless this has been altered by using gDefinePictureUnits() - see page 39). In the majority of cases, the user will want to specify a different coordinate system for two reasons. Firstly, because an application needs to be written to run on a number of different output devices (with different drawing limits) and secondly, the items or models in any application will rarely be limited to the physical limits of any one device. The method by which this mismatch is overcome is to set up an application dependent mapping from the users own coordinate system to that of the physical device on which the application is running.

Whether a mapping is required or not, an application will often require to restrict the drawing to within a particular subsection of the complete drawing area, or even mask the output from another area that is used by a menu or key.

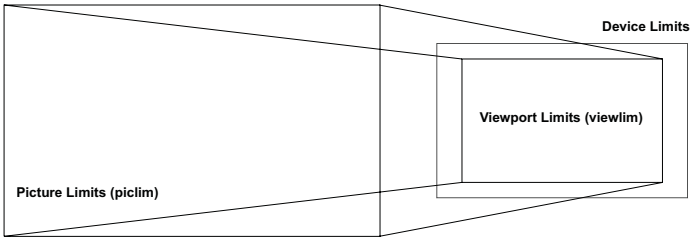
This section of the document describes the specification and use of simple rectangular clipping areas or windows and rectangular masks. GINO also provides for the definition of polygonal windows and masks (see page 245).

Viewport Mapping

A viewport mapping can be specified between the users picture coordinates and the viewport limits in paper coordinates. This mapping can be altered at any time throughout a GINO program using the routine:

gSetViewport2D(piclim, viewlim)

where **piclim** and **viewlim** are structures of type GLIMIT, the first of which specifies the users own picture coordinate limits within which all drawing dimensions should be contained. These will be mapped to the viewport limits specified by the second structure, which should lie within the current paper limits. Any viewport limits outside of the paper limits will be clipped accordingly.



Viewport Mapping

The aspect ratio of a viewport can be controlled by the routine:

gSetViewportMode(sw)

If the aspect ratio of the picture coordinates is retained the drawing can be centred in the viewport (**sw**=GCENTRAL), or placed at the bottom left of the viewport (**sw**=GBOTTOMLEFT). For **sw**= DEFORMED the aspect ratio is altered and the drawing will be scaled to fit the viewport.

The routine gEnqViewportMode() returns the current viewport mode.

gEnqViewportMode(sw)

By default, the setting of a viewport defines a clipping rectangle outside which no drawing can take place. In this default mode, the viewport limits effectively re-define the device limits of the current device. However, it is possible to use the viewport setting simply to define a convenient mapping without affecting the clipping in any way. The routine to switch between these two modes is:

gSetViewportClipSwitch(clp)

Both gSetViewportMode() and gSetViewportClipSwitch() must be called before gSetViewport2D() for either to take effect.

Viewport Enquiry

The current viewport state can be enquired using the following routines:

gEnqViewport2D(piclim, viewlim)

gEnqViewportState(sw, clp, limit)

where gEnqViewport2D() returns the requested viewport limits and gEnqViewportState() returns all viewport switches and actual viewport limits in paper units. This may vary from the requested paper limits depending on the current viewport mapping switch.

Clearing the Viewport

The current viewport area, as defined by gSetViewport2D() and gSetViewportMode(), can be cleared by filling with the background colour with the routine:

gClearViewport()

The area cleared by gClearViewport() represents the complete viewport limits in picture units as specified by gSetViewport2D() (excluding parts outside the device limits). Depending on the viewport mapping switch this may not represent the complete viewport in paper units set by gSetViewport2D(), but rather, those limits as returned by gEnqViewportState(). If the larger area needs to be cleared, this should be done prior to setting the viewport.

Clipping

The default clipping limits are defined to be the current drawing limits as defined by a call to gSetDrawingLimits() or the current viewport limits as set by gSetViewport2D() as long as viewport clipping is not switched off with gSetViewportClipSwitch() (see above).

The clipping is carried out by the device driver where possible as this is usually the most efficient and accurate method possible. It is possible to specify that the clipping is to be performed by GINO itself or that clipping be disabled completely. These options are set using the routine:

gSetClippingMode(mode)

where **mode** may be GHARD (the default) for hardware clipping, GSOFT for software clipping or GNOCLIP to disable clipping completely. The last mode should only be used where output is known to be limited to the physical limits of the device as unpredictable results may occur otherwise.

The routine gEnqClippingMode() returns the current clipping mode.

gEnqClippingMode(mode)

Note that the windowing and masking limits are in picture coordinates. Any calls to scaling or transformation routines have no effect on the resulting window size.

Window Mode

The routine for specifying the current clipping/windowing mode is:

gSetWindowMode(swi)

This routine switches the windowing mechanism off (**swi**=GOFF) or on (**swi**=GON or GON2D).

Switching windowing off effectively sets a window to the default clipping limits but GINO will generate warning messages if an attempt is made to draw outside the device limits while in this state. When **swi**=GON, the previous window limits prior to switching windowing off are restored. When **swi** = GON2D, the function switches 2-D windowing on and sets the window to the default 2D clipping limits.

To switch on basic 2-D windowing and set the window to the viewport limits:

```
gSetWindowMode(GON2D);
```

```
call gSetWindowMode(GON2D)
```

Rectangular Window

The routine for defining a 2-D rectangular window is:

gSetWindow2D(window)

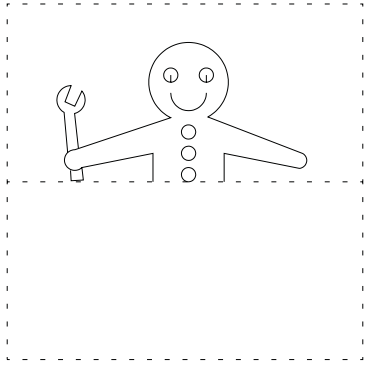
The structure **window** contains four elements representing the limits of a rectangular window: **window.xmin**, **window.xmax**, **window.ymin**, **window.ymax**. If the user specifies a window larger than the default clipping limits, GINO clips it to those limits, but combines and retains the window.

For example:

Routine man() defines a gingerbread man in a 50mm square box. To draw the top half of the picture only (as in the figure below):

```
static GLIMIT window =
    {0.0,50.0,25.0,50.0};

gSetWindow2D(&window);
man();
```

```
type (GLIMIT) :: window = &
    GLIMIT{0.0,50.0,25.0,50.0}

call gSetWindow2D(window)
call man
```



2-D Windowing

Enquiring Window Limits

The current state of windowing may be obtained by using the routine:

gEnqWindowState(swi, bounds)

The current setting of the window switch is returned in **swi** and the complete 3D limits are returned in the structure **bounds**. Note that if no Z limits have been set using gSetWindow3D(), these are returned as arbitrary large numbers.

For example:

```
GLIMIT3 window;

gEnqWindowState(&swi,&window);
```

```
type (GLIMIT3) window

call gEnqWindowState(swi,window)
```

Rectangular Masks

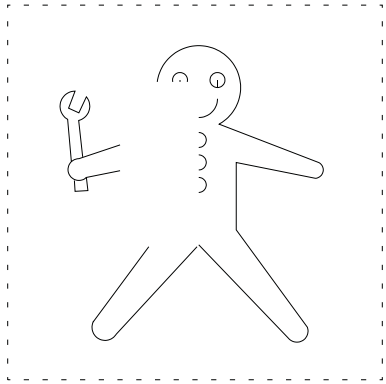
The routine to define a rectangular mask and switch masking on is:

gSetMask2D(limit)

The argument of type GLIMIT defines a rectangular area, inside which no drawing will occur until the mask is switched off or redefined. For example:

```
static GLIMIT mask =
       {150.0,25.5,17.75,39.75};

gSetMask2D(&mask);
man();
```

```
type (GLIMIT) :: mask = &
   GLIMIT(150.0,25.5,17.75,39.75)

call gSetMask2D(mask)
call man
```



Example of rectangular mask

The routine to switch the current mask on or off is:

gSetMaskMode(swi)

where **swi**=GOFF switches masking off and **swi**=GON restores the previous mask as defined by gSetMask2D(). The routine gSetMaskMode() has no effect if a mask has not previously been defined.

Mask Enquiry

To enquire the current mask limits and its switch, the routine gEnqMaskState() is used:

gEnqMaskState(swi, bounds)

swi returns the current setting of gSetMaskMode() and the remaining arguments are the same as gEnqWindowState() except that a maximum of four limits are available.

Chapter 11

2D TRANSFORMATIONS

2D Transformations Introduction

Routines are provided in GINO to enable the geometric transformations: shift, rotate, scale and shear to be applied to definitions of 2D objects. A simple definition may be transformed to produce objects of differing shape, size, orientation or position.

When a transformations routine is called, a new axis system (termed ‘space axes’) is created and subsequent drawing and positioning coordinates are considered in relation to this new axis system. When transformations are being combined, each transformation is relative to the axis system set up by the previous transformation.

To apply transformation to an object, the transformation routines must be called before the drawing routines. Once a transformation routine is called, the transformation that has been set up affects all subsequent drawing. Transforming can be switched on or off at any stage in a program or can be reset or modified.

To illustrate transformations a routine man(), which defines a gingerbread man, is used. The original axes are drawn as solid lines and denotes X and Y. The space axes set up as a result of the transformations are denoted in the diagrams by X1 and Y1 and are shown as dashed lines.

Simple 2D Transformations

2D Shifting

Shifting specifies the vector increments through which the origin is shifted from the origin of the previous axis system.

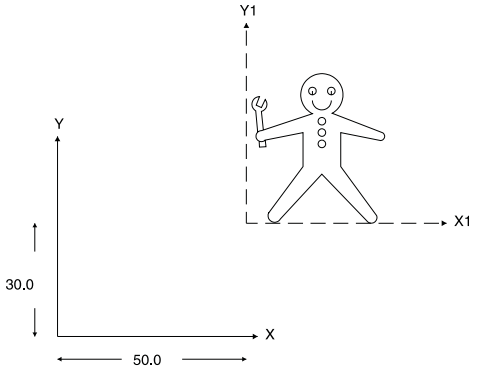
Shifting enables objects to be repositioned anywhere in the drawing area. The routines for shifting are:

gShift2D(dx, dy)

For example - to draw a gingerbread man shifted by 50.0mm in the X direction and 30.0mm in the Y direction:

```
gShift2D(50.0,30.0);
man();
```

```
call gShift2D(50.0,30.0)
call man
```



Shifting

2D Rotation

The routine for 2-D rotation is:

gRotate2D(angle)

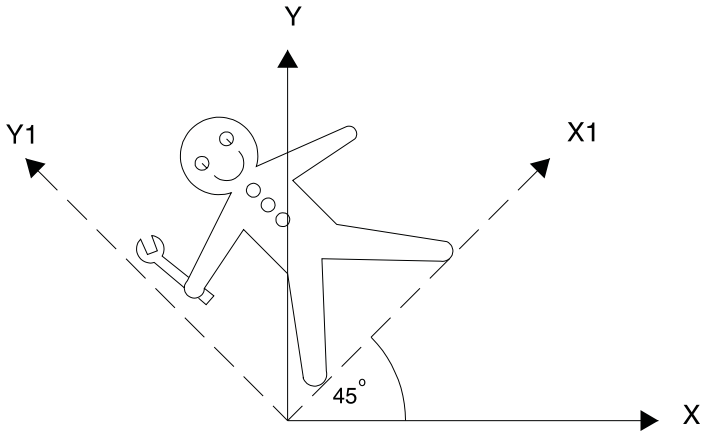
Where **angle** specifies the angle in degrees through which the X and Y axes are rotated about the Z axis. Positive rotation is anticlockwise. Note that rotation always takes place about the origin of the current axis system.

Example:

To draw a gingerbread man rotated through 45°:

```
gRotate2D(45.0);
man();
```

```
call gRotate2D(45.0)
call man
```



2-D Rotation

2D Scaling

The routine for scaling is:

gScale2D(sx, sy)

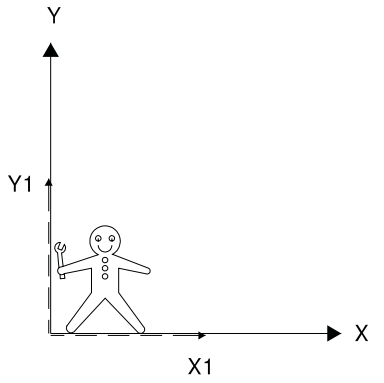
The arguments specify the amount by which the axes are to be scaled. Values of greater than 1.0 give magnification and values between 0.0 and 1.0 give reduction. If one or more arguments are negative, then a mirror image is produced.

Examples:

- To draw a gingerbread man uniformly scaled by 0.5 in both directions:

```
/* Scale all axes */
gScale2D(0.5,0.5);
man();
```

```
! Scale all axes
call gScale2D(0.5,0.5)
call man
```

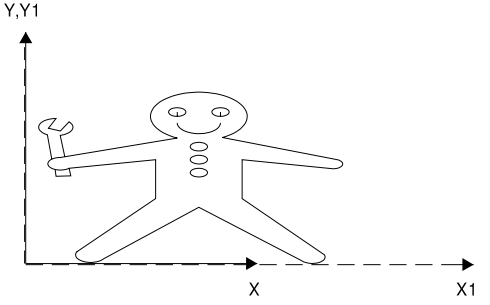


Uniform Scaling

- To draw a fat gingerbread man scaled in X by 2:

```
gScale2D(2.0,1.0);
man();
```

```
call gScale2D(2.0,1.0)
call man
```



Differential Scaling

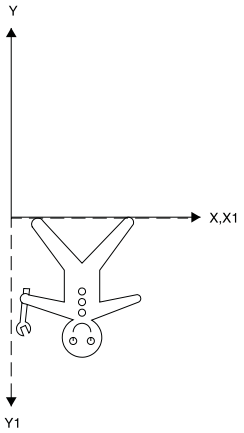
Mirror Images

Mirror images can be produced by using the scaling routines with negative arguments. For example:

To draw a gingerbread man upside down:

```
gScale2D(1.0,-1.0);
man();
```

```
call gScale2D(1.0,-1.0)
call man
```



Mirror Image

2D Shearing

The routines for shearing are:

gShear2D(dep, a)

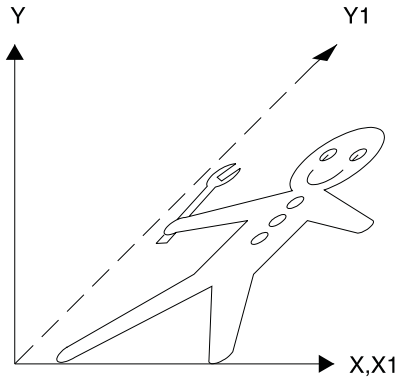
In 2-D the value of **dep** can be GXAXIS or GYAXIS and indicates which of the X or Y axes is to be sheared (the dependent axis).

The argument **a** gives the tangent of the angle through which the axis **dep** is sheared. For example:

- To draw a sheared gingerbread man such that the shear factor is 1.0:

```
gShear2D(GYAXIS,1.0);
man();
```

```
call gShear2D(GYAXIS,1.0)
call man
```



2-D Shearing

Combining Transformations

Using the Same Transformation Type

When combining transformations of the same type, the general result is not dependent on the order in which the routines are called.

Example:

```
gScale2D(10.0,10.0);
gScale2D(3.0,3.0);
```

```
call gScale2D(10.0,10.0)
call gScale2D(3.0,3.0)
```

has the same effect as:

```
gScale2D(3.0,3.0);
gScale2D(10.0,10.0);
```

```
call gScale2D(3.0,3.0)
call gScale2D(10.0,10.0)
```

The above sequence of routines is equivalent to a single call to the routine with an arguments of 30.0, i.e. the combined effect is obtained by multiplying the arguments. In the case of transformation routines other than the scale routines, the cumulative effect is obtained by adding the arguments.

Example:

```
gRotate2D(alpha);
gRotate2D(beta);
```

```
call gRotate2D(alpha)
call gRotate2D(beta)
```

is equivalent to:

```
gRotate2D(alpha+beta);
```

```
call gRotate2D(alpha+beta)
```

Using Different Transformation Types

When combining transformations of different types, the effect obtained depends on the order in which the routines are called. The following examples show the effect of applying combinations of transformations in different orders.

In general the order in which transformations should be used to set up straightforward effects is:

Shift

Rotate

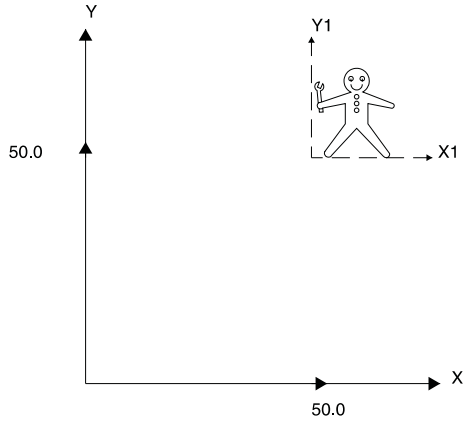
Scale

Combining Shift with Scale

The figure below illustrates the following sequence:

```
gShift2D(50.0,50.0);
gScale2D(0.5,0.5);
man();
```

```
call gShift2D(50.0,50.0)
call gScale2D(0.5,0.5)
call man
```

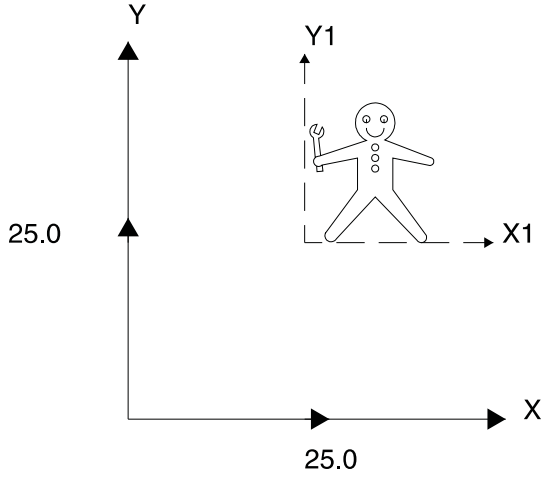


With the order reversed, i.e.:

```
gScale2D(0.5,0.5);
gShift2D(50.0,50.0);
man();
```

```
call gScale2D(0.5,0.5)
call gShift2D(50.0,50.0)
call man
```

The effect is as shown in the figure below:

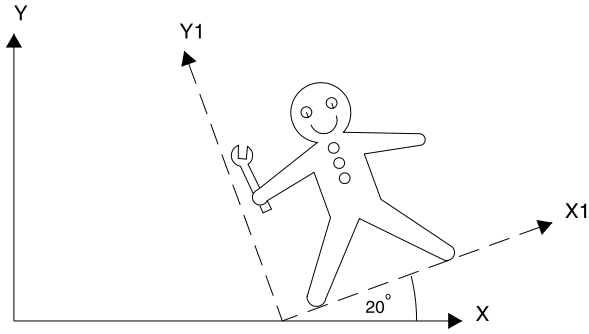


Combining Shift and Rotation

The figure below illustrates the following sequence:

```
gShift2D(50.0,0.0);
gRotate2D(20.0);
man();
```

```
call gShift2D(50.0,0.0)
call gRotate2D(20.0)
call man
```

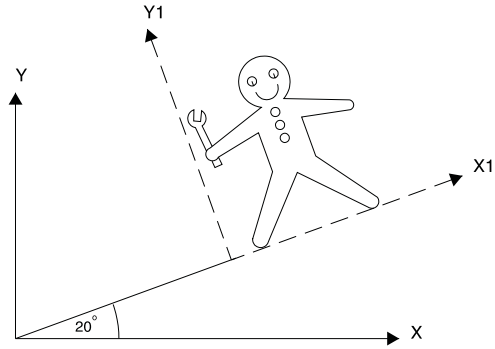


With the order reversed i.e.:

```
gRotate2D(20.0);
gShift2D(50.0,0.0);
man();
```

```
call gRotate2D(20.0)
call gShift2D(50.0,0.0)
call man
```

The result is as shown below:



Combining Rotation with Uniform Scaling

Where rotation is combined with non-differential scaling, the order of calling the routines does not effect the result.

Example, the sequence:

```
gScale2D(2.0,2.0);
gRotate2D(-30.0);
man();
```

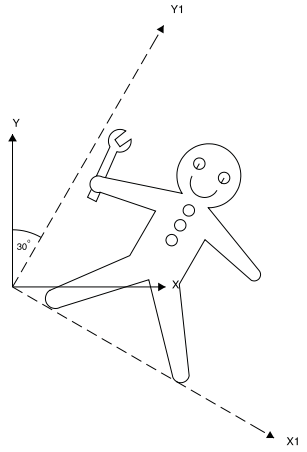
```
call gScale2D(2.0,2.0)
call gRotate2D(-30.0)
call man
```

gives the same result as:

```
gRotate2D(-30.0);
gScale2(2.0,2.0);
man();
```

```
call gRotate2D(-30.0)
call gScale2D(2.0,2.0)
call man
```

The result is shown below:



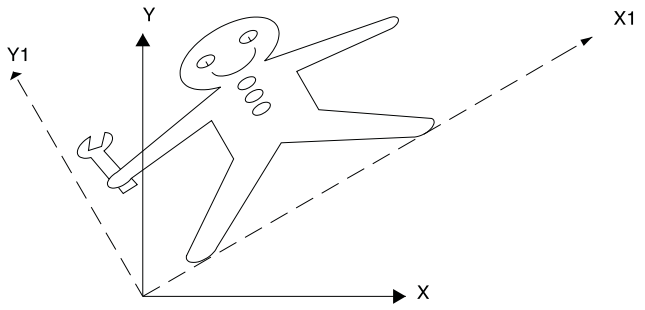
Combining Rotation with Differential Scaling

The effect of combining rotation with differential scaling is dependent on the order in which routines are called. For example:

```
gRotate2D(30.0);
gScale2D(2.0,1.0);
man();
```

```
call gRotate2D(30.0)
call gScale2D(2.0,1.0)
call man
```

gives the result shown below:

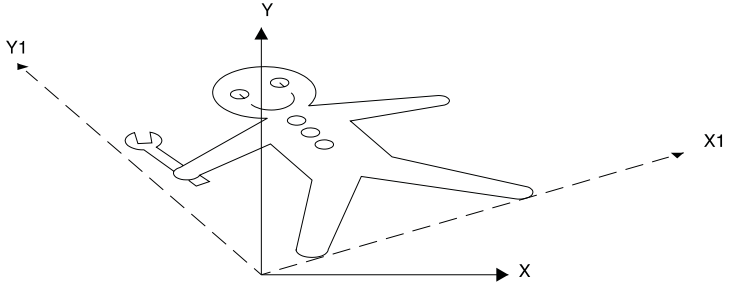


However, the sequence:

```
gScale2D(2.0,1.0);
gRotate2D(30.0);
man();
```

```
call gScale2D(2.0,1.0)
call gRotate2D(30.0)
call man()
```

gives a different result as shown below:



Note: The axes are no longer at right-angles.

2D Transformation Enquiry

Current Drawing Position

The position of the 'pen' at any given time can be described in terms of its coordinates relative to the origin of the drawing area; these are termed 'picture coordinates'. The pen position can also be given in terms of its coordinates relative to the current local axis system; these are the space coordinates. In general, objects are specified in terms of space coordinates and are drawn in terms of picture coordinates; that is space coordinates are transformed into picture coordinates.

At any stage in a program, the pen position can be obtained (in either picture or space coordinates) by using one of the following routines:

gEnqPicturePos(point)

gEnqSpacePos(point)

Each of which returns a structure of type GPOINT3 structure, the elements of which are set to the current X,Y,Z coordinates expressed in current units.

2D Untransforming

The space coordinates of any point of which the picture coordinates are known, can be obtained using one of the routines:

gUntransformPoint2D(xp, yp, point)

The routine gUntransformPoint2D() sets **point.x** and **point.y** to zero if the current transformation contains 3-D terms or perspective and a warning message is output.

Point Testing of Current 2D Transformation

The routine gTransformPoint2D() enable the user to see what would happen to a point if it were subject to the current transformation.

gTransformPoint2D(xs, ys, point)

transforms the space coordinate position (**xs**, **ys**) into picture coordinates and returns the value of the picture coordinates in (**point.x, point.y**).

In the case where no current transformation exists, (**xs**, **ys**) and (**point.x**, **point.y**) will have the same value.

2D Transformation Control

There are many addition facilities to switch on/off, save, combine or reinitialize the current 2D transformation state. These are described along with their 3D counterparts later in this document (see page 371).

The most useful routine for basic 2D transformation work is:

gSetTransformMode(swi)

which can be used to switch the current transformation on or off or reinitialize it to its initial null (or unit) state.

Transforming Characters and Symbols

By default, GINO starts up in an untransformed character mode. If hardware characters are required, these cannot be affected by GINO transformations, but the transforming of software characters can be switched on by calling:

gSetCharTransformMode(GON)

Characters and symbols will be generated in software by GINO using straight lines which are transformed and windowed and then output using the current line style (Broken, Thickness and End type). The size, orientation and italic angle with respect to the current space axes are exactly as specified, but the characters are also affected by the current transformation set by gRotate2D(), gShift2D(), gShear2D() or any of the transformation matrix routines.

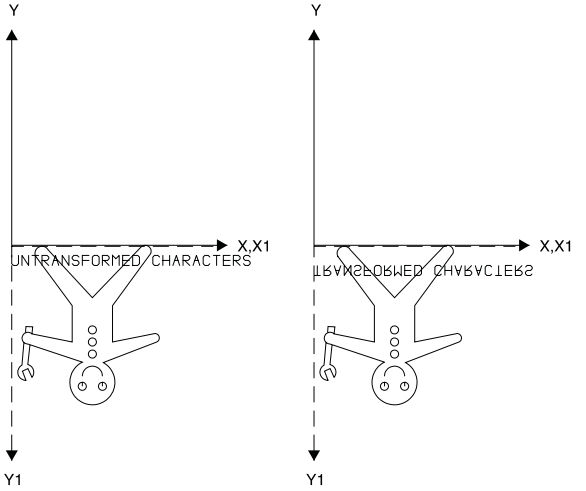
For example - If a mirrored image is required, a call to gShift2D(1.0,-1.0) will only mirror the characters if gSetTransformMode(GON) has been called.

C code

```
/* Set up mirror image transformation */
      gScale2D(1.0,-1.0);
      gmanandtext();
/* Select transformable software characters */
      gSetCharTransformMode(GON);
      gmanandtext();
```

F90 code

```
! Set up mirror image transformation
      call gScale2D(1.0,-1.0)
      call gmanandtext
! Select transformable software characters
      call gSetCharTransformMode(GON)
      call gmanandtext
```



Software Transformable Characters

The transforming of characters and symbols can be switched off by calling gSetCharTransformMode(GOFF). In this case, the character mode reverts to the one set by the last call to gSetHardChars(), gSetMixedChars() or gSetSoftChars(), or failing that it reverts to the default character mode gSetMixedChars().

Chapter 12

BASIC INTERACTION

Basic Interaction Introduction

GINO provides a simple facility to input information from a graphics device. This is the cursor or mouse input facility and is supported by the majority of graphics terminals and workstations. Cursor input provides a way of communicating or interacting with a GINO program. There are, of course, other ways to input information, e.g. standard C or Fortran I/O or using a GUI interface such as GINOMENU.

Advanced interaction ‘events’ are described later in this document (see page 447).

Cursor Input

A graphics device that supports cursor input will be able to indicate a position on the drawing surface by means of a graphics cursor or pointer. It will also provide some means for the user to move the cursor around either with a mouse, arrow keys, joystick or other device. When the cursor has been appropriately positioned, the user presses a key or button to cause the cursor’s position to be returned to GINO. An ASCII character code to identify which key was pressed is also returned.

The routine to call for cursor input is:

gGetCursorEvent(key, point)

When gGetCursorEvent() is called, the graphics cursor or pointer is switched on or changed to a graphics cursor shape. As soon as input has been triggered by the user, gGetCursorEvent() returns in **point** the cursor's position in picture coordinates. Cursor input is usually triggered by pressing a key on a keyboard or a button on a mouse. **key** returns the key's ASCII code as an integer number. If something other than a keyboard key was pressed, **key** returns an ASCII code which identifies the trigger. (See page 447 for more details on the returned ASCII values.)

On non-windowing graphics devices, a routine is provided to enable the user to set the cursor position:

gSetCursorPos(x, y)

The position (**x,y**) is specified in picture coordinates. The default for the cursor start position is usually the centre of the device limits and it is reset to this after a call to gNewDrawing(). When gGetCursorEvent() is called, the cursor should appear at or move to the specified position. If the call to gGetCursorEvent() is successful, the cursor start position is automatically updated to the position returned by gGetCursorEvent(). Therefore the call to gSetCursorPos() can only affect the next call to gGetCursorEvent().

On windowing devices the pointer is always present but its position can be enquired or set using the mouse position routines (See page 456)

Defining Cursor Shapes

For the devices that offer a cursor or mouse input device it is often possible to alter the shape of the cursor. The routine gSetCursorType() changes the cursor type and gEnqCursorType() returns the current setting.

gSetCursorType(type, forcol, bakcol)

gEnqCursorType(type, forcol, bakcol)

Where **type** is a positive integer that defines the type of cursor. A choice of a small or large cross is often available on raster terminals, whereas a larger selection is available on window machines where the cursor represents the current pointer position and the shape is defined as a small bit pattern. The number of cursor types available on the current device can be obtained through the routine gEnqDeviceState() (see page 39) and users are referred to the relevant Appendix B document for further details.

The arguments **forcol** and **bakcol** define the foreground and background colours of the cursor if these can be set.

Defining Cursor Action Types

As well as setting the cursor shape, some devices can provide additional functionality in the form of 'rubber' shapes that are continually updated while the cursor or pointer device is moved. Typical shapes are rubber bands, rubber boxes and rubber ellipses. Where such functionality is provided, the routine gSetCursorAction() will set the desired cursor action type and gEnqCursorAction() returns the current setting.

gSetCursorAction(action, lverts, points)

gEnqCursorAction(action, lverts, points)

where **action** defines a number of 'rubber' shapes that indicate both the current pen position when gGetCursorEvent() is called and the current pointer position. For example, when **action** = GRUBBERBOX, a 'rubber' box is drawn with one static corner at the current pen position and a variable corner at the current pointer position.

It is also possible to define fixed polyline shapes which can be used as a cursor or pointer. To use this facility **action** is set to GPOLYLINE and the vertices of the polyline in picture coordinates are placed in the array **points**. The number of vertices is set in **lverts**. A maximum of 200 vertices is permitted in this facility. The coordinates are absolute coordinates and may be positive and/or negative such that the position (0.0,0.0) is located at the cursor or pointer position as it is moved around the screen.

Application

Although the input facility provided by gGetCursorEvent() is apparently very basic, it can be used in a variety of powerful ways. For example, a light pen mechanism can be simulated by dividing part of the screen into areas so that a cursor 'hit' inside an area indicates a particular function. It is also possible to identify items on the screen by performing a similar area check. In this case, however, it is necessary to have some sort of data structure to record where each item is drawn. This is achieved by using segment handling software described later (see page 423).

Chapter 13

ADVANCED USE OF 2D POLYGONS

Advanced Use of 2D Polygons Introduction

Polygonal boundaries may be defined and stored for subsequent use, e.g. filling, and the definitions are stored in a workspace which is part of the gSetWorkspaceLimit() workspace area maintained by GINO (see page 33). The boundaries of a polygon are lines which join its vertices and the vertices are stored as the polygon definition. The vertices are generated by GINO drawing routines and the first and last vertices are connected, meaning that polygons are always closed.

Allocating Workspace for the Storage of Polygons

Polygon definitions need a workspace within which they may be stored. The routine gDefinePolygonWorkspace() is used to define such a workspace within the gSetWorkspaceLimit() workspace (see page 33).

gDefinePolygonWorkspace(nw)

Polygon workspace size depends on the size and number of the polygons to be stored. Each polygon has a number of vertices which define its edges. Each vertex needs two real words of storage space. Each polygon also requires a header of eight real words. So, if NP polygons with a combined total of NV vertices are to be stored, the amount of space required, NW (number of real words) is:

NW = 2*NV + 8*NP

For example, if a 25 sided and a 12 sided polygon are stored,

NV = 25 + 12

NP = 2

So,

NW = (2*37) + (8*2)

= 90

The call:

```
gDefinePolygonWorkspace(90);
```

```
call gDefinePolygonWorkspace(90)
```

would allocate just sufficient space within the workspace area to store the two polygons.

The call to gDefinePolygonWorkspace() must be preceded by a call to gSetWorkspaceLimit() to define the global workspace. This must include at least sufficient space for gDefinePolygonWorkspace's needs; i.e. **nw** must not be larger than the space delimited by gSetWorkspaceLimit() .

Example:

```
#include <gino-c.h>
main()
{
  gOpenGino();
  gSetWorkspaceLimit(3000);
  gDefinePolygonWorkspace(1200);
  .
  .
  gCloseGino();
}
```

```
program polygon
use gino_f90

  call gOpenGino()
  call gSetWorkspaceLimit(1,3000)
  call gDefinePolygonWorkspace &
                              (1200)
  .
  .
  call gCloseGino
stop
```

A polygon definition which uses arcs or curves can generate many vertices. The user must either allow sufficient space in the polygon workspace, or modify the resolution of the currently nominated output device. The greater the device resolution, the more vertices are used to describe a curve. The default tolerance set for each device generally gives a reasonable result. However, this may be controlled by GINO software by using the gSetArcTolerance() routine (see page 77).

Polygon Definition

gStartPolygon()

gEndPolygon()

gSetPolygonMode(sw)

Polygons are defined by a series of lines, moves, etc. The coordinates supplied by the user are transformed, if there is a current transformation, and the resulting X and Y picture coordinates can be stored as polygon vertices.

The storage of the vertices is controlled by gStartPolygon() and gEndPolygon(). gStartPolygon() starts a new polygon and gEndPolygon() closes the current polygon. The sequence of calls would be as follows:

```
   gMoveTo2D(20.0,50.0);
   gStartPolygon();
/* define a triangle */
   gDrawLineBy2D(20.0,30.0);
   gDrawLineBy2D(20.0,-30.0);
   gMoveTo2D(20.0,50.0);
   gEndPolygon();
```

```
  call gMoveTo2D(20.0,50.0)
  call gStartPolygon
! define a triangle
  call gDrawLineBy2D(20.0,30.0)
  call gDrawLineBy2D(20.0,-30.0)
  call gMoveTo2D(20.0,50.0)
  call gEndPolygon
```

Once gStartPolygon() is called, vertices defined by subsequent drawing routines are stored in the polygon workspace. gEndPolygon() stops this storage of vertices and closes the polygon definition. A polygon cannot be changed in any way once it is closed. The gStartPolygon/gEndPolygon() pair delimit a polygon definition. gStartPolygon() contains an implicit call to gEndPolygon() so in a sequence of polygon definitions it is possible to omit the calls to gEndPolygon(). However, remember not to leave a gEndPolygon() outstanding on completion of such a sequence.

gSetPolygonMode() operates two independent switches one of which functions outside gStartPolygon/gEndPolygon() and the other inside gStartPolygon/gEndPolygon().

Outside gStartPolygon/gEndPolygon():

`gSetPolygonMode(GOFF)`	Disables all polygon storage and reinitializes the gDefinePolygonWorkspace() workspace (see gClearPolygonWorkspace()).

`gSetPolygonMode(GON)`	Enables polygon storage and gClearPolygonWorkspace(). This is the default state.

Inside gStartPolygon()/gEndPolygon():

`gSetPolygonMode(GOFF)`	Suppresses the vertex storage
`gSetPolygonMode(GON)`	Enables vertex storage (set to on by gStartPolygon())

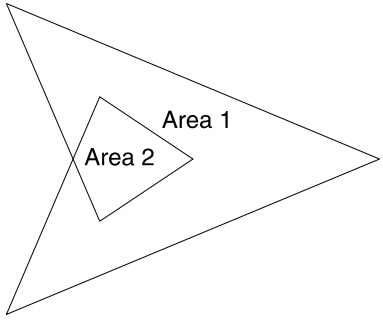
The DART shape in the figure below is produced by the following code:

C Code

```
/* ***** DART ***** */
float x,y,z,x1,y1,z1;

/*  Move to start point */
   gMoveTo2D(180.0,250.0);
/*  Switch on polygon storage */
   gStartPolygon();
/*  Start defining polygon */
   qDrawLineBy2D(120.0,-50.0);
/*  Switch off vertex storage */
   gSetPolygonMode(GOFF);
/*  Record where you are */
   gEnqSpacePos(&x,&y,&z);
/*  Move to start of write position */
   qMoveBy2D(-60.0,10.0);
/*  and output text */
   qDisplayStr("Area 1");
/*  Move back to recorded position */
   gMoveTo2D(x,y);
   gSetPolygonMode(GON);
   gDrawLineBy2D(-120.0,-50.0);
   gDrawLineBy2D(30.0,70.0);
   gDrawLineBy2D(30.0,-20.0);
   gDrawLineBy2D(-30.0,-20.0);
   gSetPolygonMode(GOFF);
   gEnqSpacePos(&x1,&y1,&z1);
   gMoveBy2D(-5.0,18.0);
   gDisplayStr("Area 2");
   gMoveTo2D(y1,y1);
   gSetPolygonMode(GON);
   gDrawLineTo2D(180.0,250.0);
/* Close polygon */
   gEndPolygon();
```

F90 code

```
! ***** DART ***** */
real x,y,z,x1,y1,z1

!  Move to start point
   call gMoveTo2D(180.0,250.0)
!  Switch on polygon storage
   call gStartPolygon
!  Start defining polygon
   call gDrawLineBy2D(120.0,-50.0)
!  Switch off vertex storage
   call gSetPolygonMode(GOFF)
!  Record where you are
   call gEnqSpacePos(x,y,z)
!  Move to start of write position
   call gMoveBy2D(-60.0,10.0)
!  and output text
   call gDisplayStr('Area 1')
!  Move back to recorded position
   call gMoveTo2D(x,y)
   call gSetPolygonMode(GON)
   call gDrawLineBy2D(-120.0,-50.0)
   call gDrawLineBy2D(30.0,70.0)
   call gDrawLineBy2D(30.0,-20.0)
   call gDrawLineBy2D(-30.0,-20.0)
   call gSetPolygonMode(GOFF)
   call gEnqSpacePos(x1,y1,z1)
   call gMoveBy2D(-5.0,18.0)
   call gDisplayStr('Area 2')
   call gMoveTo2D(y1,y1)
   call gSetPolygonMode(GON)
   call gDrawLineTo2D(180.0,250.0)
!  Close polygon
   call gEndPolygon
```



DART: An Annotated Polygon

Polygon Identity

gSetPolygonIdent(ident)

Each polygon in the polygon workspace may be given an identifier. When a polygon is closed (gEndPolygon()), it is given the current identifier, which must be a positive integer number. The current polygon identifier is the last identifier defined by gSetPolygonIdent(). The default identifier is zero and if no calls are made to gSetPolygonIdent() all polygons will have an identifier of zero.

Polygons sharing a common identifier will be grouped together and the whole group will be selected by the one identifier. Identifiers cannot be changed once the polygon is closed.

For example:

```
    gStartPolygon();
/*  Define a polygon */
    .
     .
/*  Define polygon identifier */
    gSetPolygonIdent(ident);
    gEndPolygon();
```

```
    call gStartPolygon
!   Define a polygon
    .
    .
!   Define polygon identifier
    call gSetPolygonIdent(ident)
    call gEndPolygon
```

A polygon must be given a unique identifier if it is to be distinguished from all other polygons.

Clearing Polygon Workspace

By calling gClearPolygonWorkspace() the gDefinePolygonWorkspace() storage space is reinitialized.

gClearPolygonWorkspace()

This deletes all polygons in the gDefinePolygonWorkspace() workspace. The routine gClearPolygonWorkspace() is ignored if gSetPolygonMode(GOFF) is called outside gStartPolygon()/gEndPolygon(); that is for gClearPolygonWorkspace() to re-initialize the workspace, gSetPolygonMode() must be set to GON (default).

Status of Polygon Workspace

The routine gEnqPolygonWorkspace() returns information about the state of gDefinePolygonWorkspace() workspace, and the value of the current polygon identifier.

gEnqPolygonWorkspace(npoly, nvert, nfree, ident)

npoly	Returns the total number of completed polygons with one or more vertices.
nvert	Returns the total number of vertices defined so far.
nfree	Returns the amount of remaining free space (real words).
ident	Returns the current polygon identifier.

If gEnqPolygonWorkspace() is called after the polygon DART has been defined, and assuming gDefinePolygonWorkspace() was set up as in the example in the 'Allocating Workspace for the Storage of Polygons' section, the following results will be returned:

npoly=1

nvert=7

nfree=1178

These three quantities are related to gDefinePolygonWorkspace() size in this way

(8 * npoly) + (2 * nvert) + nfree = nw

where nw is the number of words allocated in gDefinePolygonWorkspace().

If gEnqPolygonWorkspace() is called before any polygons are defined, **npoly** and **nvert** return zero and **nfree** returns **nw**, the number of words declared in gDefinePolygonWorkspace(). If no gDefinePolygonWorkspace() storage space has been assigned, **nfree** will also return zero.

Drawing Polygon Boundaries

Polygon boundaries may be drawn by a call to the routine:

gDrawPolygonBound(line)

where **line** selects the line style for the boundary from the line definitions table. If **line** is set to GCURRENT the edges are drawn in the current line style.

For example, if the drawing area is cleared by a call to gNewDrawing(), a polygon may be redrawn by a call to gDrawPolygonBound(). The statements:

```
gNewDrawing();
gDrawPolygonBound(GCURRENT);
```

```
call gNewDrawing
call gDrawPolygonBound(GCURRENT)
```

would have this effect if appended to the code for DART. The routine gDrawPolygonBound() draws all edges of all selected polygons, including those edges that were originally drawn with invisible lines. In the DART example the annotations ‘Area 1’, ‘Area 2’ will not be drawn by gDrawPolygonBound() since they were not stored as part of the definition.

Polygon Filling Workspace Requirements

Before complex polygons can be filled, a temporary workspace must also be provided. The space this workspace needs is in addition to the space needed for polygon storage (see page 33). The following sections detail how to calculate the size of the temporary workspace depending whether hardware or software fill is in operation.

If software filling is forced using gSetFillMode(GSOFT) then only the space required for software filling needs to be calculated. Clearly if hardware and software fill occur in the same program through user selection or the device capabilities the maximum size must be catered for, however, allowances for windowed and masked filled areas are described under ‘Polygon Windowing and Masking’ later in the section.

Hardware Fill Workspace Requirements

GINO ensures that all polygonal boundaries presented to the output device remain within device limits. A temporary workspace is required to hold this data.

No specific rule can be given for the size of this temporary workspace. In the extreme case, the clipping process can generate up to twice the original number of vertices in any polygon. In normal circumstances, clipping reduces the amount of data that needs to be stored.

The following formula should provide sufficient additional space NWHF, for hardware area fill:

NWHF = NP*8 + NV*2 (words)

Where,

NP = number of polygons selected for fill

NV = total number of vertices in all selected polygons

This is equivalent to NW as defined previously if all defined polygons are selected for fill simultaneously.

Software Fill Workspace Requirements

The area-filling software in GINO needs temporary workspaces to hold extra information that is necessary for generating hatch lines.

The size of the workspace depends on the total number of vertices, NV, describing the polygons selected for area fill.

Size = 5*NV (words)

Example Calculations of Workspace Requirements

Suppose it were necessary to store and fill a maximum of 21 polygons containing, at maximum, 200 vertices. All polygons are capable of being filled at one time; there is no section of polygons involved; hardware fill is always available and used.

What is the total workspace requirement?

Step 1. Calculate the size of the polygon storage area

Size = 2*200 + 8*21

= 568

Step 2. Calculate size of the selection list workspace

Size = 0

Step 3. Calculate size of the hardware fill workspace

Size = that found in step 1 (in general)

= 568

Total = 568 + 0 + 568 = 1136 (words)

This total is only approximate. In practice the sequence of program calls would look like this:

```
gSetWorkspaceLimit(1200);
gDefinePolygonWorkspace(568);
```

```
call gSetWorkspaceLimit(1,1200)
call gDefinePolygonWorkspace(568)
```

Supposing that only 11 polygons out of the 21 are to be selected for filling and it has been estimated that these polygons contain, at most, 100 vertices and software area fill is to be used. What would be the new workspace size?

Step 1. As before

Size = 2*200 + 8*21

= 568

Step 2. The selection list workspace size is given by

Size = 11

Step 3. Calculate size of the software fill workspace

Size = 5*100

= 500

Total = 568 + 11 + 500 = 1079 (words)

In practice the sequence of program calls would look like this:

```
int list[1];
.
gSetWorkspaceLimit(1100);
gDefinePolygonWorkspace(568);
.
.
gSetFillMode(GSOFT);
gSelectPolygons(list,11);
.
.
```

```
integer list(1)
.
call gSetWorkspaceLimit(1,1100)
call gDefinePolygonWorkspace(568)
.
.
call gSetFillMode(GSOFT)
call gSelectPolygons(list,11)
.
.
```

Polygon Selection

By default GINO will use all polygons currently defined in the polygon workspace. However, a subset of the currently defined polygons may be selected by a call to the routine:

gSelectPolygons(list, n)

Polygons are selected by their identifiers. The user integer array **list** should contain the list of identifiers to be selected. The statement:

```
int list[N];
.
gSelectPolygons(list, n);
```

```
integer list(N)
.
call gSelectPolygons(list, n)
```

then causes **n** identifiers to be copied from array **list** into the workspace area. This becomes the current list of polygon identifiers - For example:

C code

```
#include <gino-c.h>
main()
{
   int list[2];
   float x,y;
   .
   gOpenGino();
   .
   .
   gSetWorkspaceLimit(3000);
   gDefinePolygonWorkspace(1200);
   .
   .
   x=20.0;
   y=230.0;
   gSetArcIncrement(6);
/* Define and store polygons */
   for (i=1; i<=4; i++) {
      gStartPolygon();
      gMoveTo2D(x, y);
      gDrawArcBy2D(20.0,0.0,0.0,0.0,GCLOCKWISE);
/* Give polygon an identification */
      gSetPolygonIdent(i);
      gEndPolygon();
      y -= 50.0;
   }
   qNewDrawing();
/* Write identifiers 2 and 3 into list */
   list[0]=2;
   list[1]=3;
/* Copy LIST to workspace */
   gSelectPolygons(list,2);
/* Draw boundaries of selected polygons   **** */
   gDrawPolygonBound(GCURRENT);
      .
   gCloseGino();
}
```

F90 code

```
program poly
use gino_f90
   integer list(2)
   real x,y
   .
   call gOpenGino
   .
   .
```

```
      call gSetWorkspaceLimit(1,3000)
      call gDefinePolygonWorkspace(1200)
      .
      .
      x=20.0
      y=230.0
      call gSetArcIncrement(6)
!     Define and store polygons
      do i=1,4
         call gStartPolygon
         call gMoveTo2D(x, y)
         call qDrawArcBy2D(20.0,0.0,0.0,0.0,GCLOCKWISE)
!     Give polygon an identification
         call gSetPolygonIdent(i)
         call gEndPolygon
         y=y-50.0
      end do
      call qNewDrawing
!     Write identifiers 2 and 3 into list
      list(1)=2
      list(2)=3
!     Copy LIST to workspace
      call qSelectPolygons(list,2)
!     Draw boundaries of selected polygons   ****
      call gDrawPolygonBound(GCURRENT)
         .
      call gCloseGino
}
```

Only those polygons identified in the current list are considered by gDrawPolygonBound().

The routine gSelectPolygons() may be called repeatedly. Each time the list is redefined, the old one is first deleted.

The routine gSelectPolygons() uses a small amount of space within the workspace area and this should be included in the calculation of gSetWorkspaceLimit()'s size. The space needed is equal to the maximum number of identifiers to be selected at any one time. For example, seven identifiers gives a space requirement of seven words.

To cancel the gSelectPolygons() list, set n to 0:

gSelectPolygons(list,0);

This deletes the list from the workspace area and GINO reverts to using all currently defined polygons.

Polygon Selection Enquiry

An enquiry may be made to establish which polygons are currently selected for filling using:

gEnqPolygonList(list, n, count)

This routine returns the number of polygon identifiers in the last call to gSelectPolygons() as **count** and the polygon identifiers in the array **list**. The actual number of returned identifiers is between zero and **count** up to a maximum of **n**. If **n** is less than zero then a warning is given.

Filling a Polygon

GINO allows general polygonal areas to be filled. Polygonal boundaries must be defined prior to filling. These definitions should be stored in the gDefinePolygonWorkspace() workspace unless the output device's hardware is capable of storing polygons.

The following points should be considered for area filling:

1. Boundaries may be defined anywhere in picture space and are independent of any window or clipping limits.

2. Polygonal boundaries are by definition closed. There is an edge between the first and last vertex.

3. Boundaries may be of any shape and may self-intersect.

4. An area's boundary may be formed from several polygons which may intersect each other.

5. Polygons stored in gDefinePolygonWorkspace() workspace each have an identifier and therefore a subset of the currently defined polygons may be selected (gSelectPolygons()) for filling.

The routine for filling a general polygon area is:

gFillSelectedPolygons(fill, line, inv)

This is a global filling routine which can fill any general polygonal area.

The routine gFillSelectedPolygons() allows the user to choose the hatch and line styles with which the fill is to be drawn. For example:

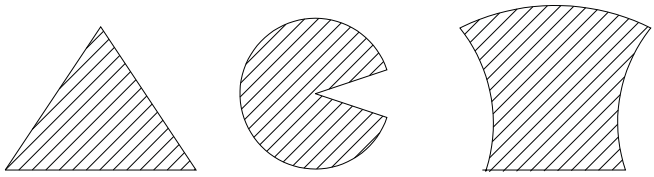
```
gFillSelectedPolygons(2,3,GAREA);
```

```
call gFillSelectedPolygons(2,3, &
                           GAREA)
```

will implement hatch style 2 (**fill**=2) and line style 3 (**line**=3). The actual appearance of the style depends on whether they are implemented by hardware or software. For **inv**=GAREA causes areas to be filled with an odd number of boundaries between them and the background area.

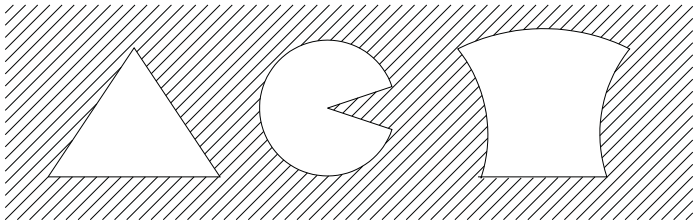
The arguments **fill** and **line** behave in the same way as for gFillRect() (see AREA FILLING), namely, they identify the hatch and line styles to be used for the fill. If **fill**=0 or out of range, it gives a solid fill and **line**=0 or out of range gives the current line style. The routine gSetFillMode() (see AREA FILLING) may be used to select between hardware and software fill styles.

The argument **inv** specifies which areas are filled. With simple polygons, that is those that do not intersect or enclose each other, when **inv** = GAREA the polygon interiors are filled as in the figure below:



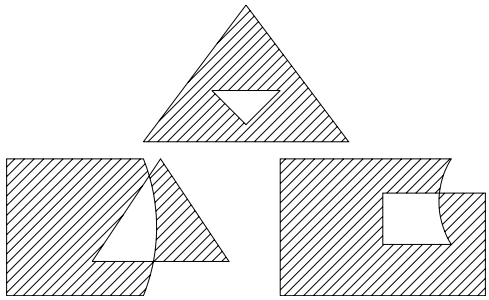
Simple Polygons, Area fill

When **inv** = GINVERSE the background area is filled as shown in the figure below:



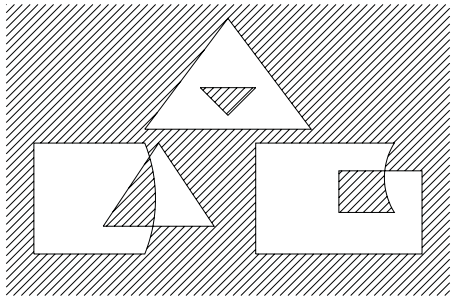
Simple Polygons Inverse fill

With complex polygons, that is those that intersect themselves or intersect or enclose each other, when **inv** = GAREA, all areas which have an odd number of edges between them and infinity are filled, as in the figure below:



Complex polygons Area fill

When **inv** = GINVERSE all areas with an even number of edges between them and infinity are filled, as in the figure below:



Complex Polygons, Inverse fill

Notice that the whole of picture space is considered when deciding which areas are to be filled. However, the actual fill is always clipped to the current window or device limits.

The polygons that are filled by gFillSelectedPolygons() are those stored in polygon workspace and selected by gSelectPolygons(). If no polygon selection has occurred, all polygons whose definitions are in the polygon workspace will be filled.

An example of this mechanism can be shown by modifying the previous program example. If the call to gDrawPolygonBound() (marked by four asterisks) is replaced by a call to gFillSelectedPolygons(), that is:

```
GFillSelectedPolygons(8,
                      GCURRENT,GAREA);
```

```
call gFillSelectedPolygons( &
                    8,GCURRENT,GAREA)
```

the two polygons identified as 2 and 3 will be filled according to hatch style 8, drawn with the current line style.

Note that it is not necessary to use the line definition table to specify the style of the filling line. By setting **line** = GCURRENT, the current line style is selected. Current line style may be modified by redefining the current line attributes. For example, adding the following code to the previous program example:

C code

```
int list[1];
      .
      .
      .
/* Reselect polygons */
      list[0]=4;
      gSelectPolygons(list,1);
/* Change current line colour and fill */
      gSetLineColour(GYELLOW);
      gFillSelectedPolygons(5,GCURRENT,GAREA);
      .
      .
      .
```

F90 code

```
integer list(1)
      .
      .
      .
! Reselect polygons
      list(1)=4
      call gSelectPolygons(list,1)
! Change current line colour and fill
      call gSetLineColour(GYELLOW)
      call gFillSelectedPolygons(5,GCURRENT,GAREA)
      .
      .
      .
```

will cause polygon 4 to be filled in yellow with hatch style 5.

The routine gSetFillMode() may be used to select between hardware and software fill styles (see page 165).

Interaction with Polygons

The routine gPolygonHit() allows the user to specify and search an area of picture space for polygons which overlap the area.

gPolygonHit(ident, x, y, r)

If any are found, the identifiers of the polygon whose edge comes closest to the centre of the area is returned in **ident**.

The search area (or 'hit area') is bounded by a circle of radius **r** whose centre, the 'hit centre', is **x**,**y** in picture coordinates.

The routine gPolygonHit() is useful in interactive applications and the coordinates for the hit centre would typically derive from a call to the routine gGetCursorEvent() (see page 241). This routine only examines those polygons which are currently selected. If no polygon overlaps the hit area, -1 is returned in **ident**.

Polygon Windowing and Masking

Polygons Suitable for Windowing and Masking

GINO allows general polygonal areas to be used as windows or masks. Polygonal boundaries must be defined prior to use. These definitions should be stored in the polygon workspace.

The following points should be considered for polygonal windowing and masking:

1. Boundaries may be defined anywhere in picture space and are independent of any window or clipping limits.

2. Polygonal boundaries are by definition closed. There is an edge between the first and last vertex.

3. Multi-polygonal windows cannot be used, only the first polygon in a set of polygons may be used.

Workspace Requirements for Windowing and Masking of Filled Areas

In order to utilize polygonal windowing and masking of filled areas enough space must be allocated. The space for windowing and masking is in addition to the space needed for polygon storage (see page 33). The following notes give guidelines on the amounts of space required for combinations of windowing and masking. The actual amounts required ultimately depend on the window, mask and object definitions.

Windowing Requirements

The following formula should provide sufficient temporary space (NWHW) to create the new clipped polygon which is sufficient for software and hardware filling, and should, therefore, be used instead.

NWHW = 7*(NV + IW + NW) + 2*NVW

where,

IW= number of intersections between the object and window

NV= number of vertices in the object being clipped

NW= number of vertices in the window

NVW= number of vertices in the windowed object

Masking Requirements

The following formula should provide sufficient temporary space (NWHM) to create the new masked polygon which is sufficient for software and hardware filling, and, therefore, should be used instead.

NWHM = 7*(NV + IM + NM) + 2*NVM

where,

IM= number of intersections between the object and mask

NV= number of vertices in the object being masked

NM= number of vertices in the mask

NVM= number of vertices in the masked object

Requirements for Simultaneous Windowing and Masking

The following formula should provide sufficient space (NWM) for simultaneously windowed and masked hardware filling:

NWM = 7*MAX((NV + IW + NW),(NV + IM + NM)) + 2*NVW + 2*NVM

where,

IW= number of intersections between the object and window

IM= number of intersections between the object and mask

NV= number of vertices in the object being clipped and masked

NW= number of vertices in the window

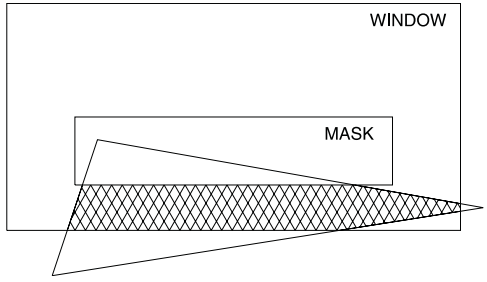
NM= number of vertices in the mask

NVW= number of vertices in the windowed object

NVM= number of vertices in the masked object

Example - Calculation of Fill Workspace Requirements

The following calculations are for the simultaneous windowing and masking of a triangle as shown below:



A Simultaneously Masked and Windowed Triangle

NV= 3

NM = 4

NW= 4 NWM = 7*MAX ((3+4+4) , (3+2+4)) + 2*5 + 2*4

IW= 4

IM= 2=7*MAX (11,9) + 10 + 8

NVW= 5

NVM=4=95

The total workspace required for storing and filling the triangle above is calculated in the following steps.

Step 1. Polygon workspace

Size = 2*3 + 8*1

= 14

Step 2. Fill workspace

Size = 95

Total = 95 + 14 = 109 (words)

The sequence of program calls would look like this:

```
int list1[1],lins2[1];
.
.
gSetWorkspaceLimit(109);
gDefinePolygonWorkspace(14);
.
.
gSetFillMode(GSOFT);
gSetPolygonWindow(list1,1);
gSetPolygonMask(list2,1);
.
.
```

```
integer list(1),list2(1)
.
.
call gSetWorkspaceLimit(1,109)
call gDefinePolygonWorkspace(14)
.
.
call gSetFillMode(GSOFT)
call gSetPolygonWindow(list1,1)
call gSetPolygonMask(list2,1)
.
.
```

Polygonal Windowing

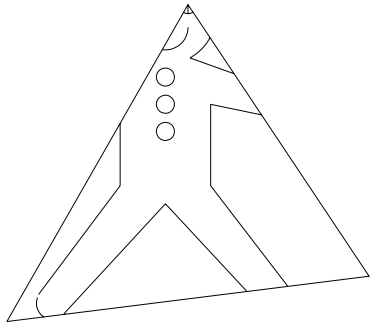
gSetPolygonWindow(list, n)

The routine gSetPolygonWindow() selects **n** polygon identifiers from the array of identifiers in **list**. The boundaries of these polygons are then used as the current window until a different set of polygons is selected or the windowing is turned off.

The following code generates a three sided window, and then draws a gingerbread man using the routine man(), as shown in the figure below.

```
int list[1];

/* Store polygon */
   gMoveTo2D(5.,0.);
   gStartPolygon();
     gSetPolygonIdent(101);
     gDrawLineTo2D(85.,10.);
     gDrawLineTo2D(45.,70.);
     gDrawLineTo2D( 5., 0.);
   gEndPolygon();
/* Store polygon identity */
   list[0] = 101;
/* Set window */
   gSetPolygonWindow(list,1);

   man();
```

```
integer list(1)

!  Store polygon
   call gMoveTo2D(5.,0.)
   call gStartPolygon
     call gSetPolygonIdent(101)
     call gDrawLineTo2D(85.,10.)
     call gDrawLineTo2D(45.,70.)
     call gDrawLineTo2D( 5., 0.)
   call gEndPolygon
!  Store polygon identity
   list(1) = 101
!  Set window
   call gSetPolygonWindow(list,1)

   call man
```



Three Sided Polygonal Windowing

Polygonal Masking

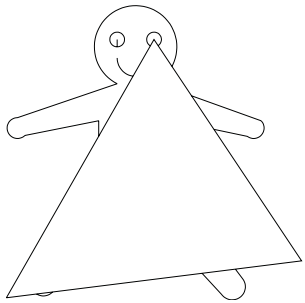
gSetPolygonMask(list, n)

The routine gSetPolygonMask() selects **n** polygon identifiers from the array of identifiers in **list**. The boundaries of these polygons are then used as the current mask until a different set of polygons is selected or the masking is turned off.

The following code generates a three sided mask, and then draws a gingerbread man using the routine man, as shown in the figure below.

```
int list[1];

/* Store polygon */
   gMoveTo2D(5.,0.);
   gStartPolygon();
     gSetPolygonIdent(101);
     gDrawLineTo2D(85.,10.);
     gDrawLineTo2D(45.,70.);
     gDrawLineTo2D( 5., 0.);
   gEndPolygon();
/* Store polygon identity */
   list[0] = 101;
/* Set mask */
   gSetPolygonMask(list,1);

   man();
```

```
integer list(1)

!  Store polygon
   call gMoveTo2D(5.,0.)
   call gStartPolygon
     call gSetPolygonIdent(101)
     call gDrawLineTo2D(85.,10.)
     call gDrawLineTo2D(45.,70.)
     call gDrawLineTo2D( 5., 0.)
   call gEndPolygon
!  Store polygon identity
   list(1) = 101
!  Set mask
   call gSetPolygonMask(list,1)

   call man
```



Three Sided Polygonal Masking

Windowing and Masking Polygon List Enquiry

The routines gEnqPolygonMaskList() and gEnqPolygonWindowList() each return a list of currently selected polygon identifiers for masking and windowing respectively.

gEnqPolygonMaskList(list, n, count)

gEnqPolygonWindowList(list, n, count)

Each routine returns up to **n** polygon identifiers. The actual number of currently selected polygons is returned as **count**. If **n** is greater than **count** then only **count** identifiers are returned; if **n** is less than **count** then only **n** identifiers are returned. The identifiers are returned in the array **list** which should be defined to hold **n** integers. If **n** is equal to or less than zero then no identifiers are returned; if **n** is less than zero then a warning message is output.

Windowing and Masking Control

Basic windowing and masking provides rectangular windows and masks only (see page 219). However, the following basic control features are also applicable to polygonal windowing and masking.

Windows and masks may be switched on and off using the routines gSetWindowMode() and gSetMaskMode() respectively.

gSetWindowMode(GOFF)	Switches current windowing off
gSetWindowMode(GON)	Switches most recently defined windowing on
gSetMaskMode(GOFF)	Switches current masking off
gSetMaskMode(GON)	Switches most recently defined masking on

Calls to gEnqWindowState() and gEnqMaskState(), return the bounding boxes of currently defined windows and masks. For example, the call

```
GLIMIT bounds;

gEnqMaskState(&swi, &bounds);
```

```
type (GLIMIT) bounds

call gEnqMaskState(swi, bounds)
```

to enquire the X and Y limits of the currently defined mask applied to the polygonal mask shown in the previous figure will return the following limits (100.0, 190.0, 100.0, 170.0)

Chapter 14

3D GRAPHICS

3D Graphics Introduction

The move from 2D graphics, covered in the earlier sections of this document to 3D graphics, is essentially the addition of the 3rd coordinate and the ability of defining and displaying objects in 3D space. However, the reality persists that at the end of the day, the objects are still being displayed on a flat 2D screen and the applications' desire is to create an illusion of a 3D scene. This can be done using a number of techniques available to the GINO user:

1) Perspective – Using a transformation of the 3D coordinates, objects further away from the viewer appear smaller than those closer to them.

2) Hidden surface removal – Sorting objects into depth order, irrespective of drawing order so that those hidden by closer objects are not displayed.

3) Shading – Calculating the effect of lights shining on objects to alter their perceived colour.

4) Depth cueing – Reducing the colour intensity of objects further away from the viewer.

All these affects are observed naturally in the real world and so give the impression of reality.

Shaded Objects

Whilst 3D scenes can be built up from lines, curves and symbols in 3D space, the primary component of realistic objects is the facet primitive. This simple polygonal shape has a number of additional attributes that enable the effect of complex lighting conditions to be displayed on the surface. The material attributes which indicate the facet's absorptive, reflective and translucent properties, together with the angle the facet makes with the viewer provide information needed to calculate the visual cues for this primitive. When the effects of multiple lights are added into the calculations, the final colour of the facet can be displayed. These calculations can be performed for each facet (flat shading) or for each vertex within a facet (smooth shading) where more realistic images are required.

The Scene

As well as the basic facet primitive, GINO also provides a set of 3D objects, such as boxes, cones, cylinders and Bezier surfaces, each of which are composed of separate facets. These provide a simple ways to start building much more complex scenes which have the same lighting and shading principles as the basic primitive.

Additional object complexity or realism can also be performed by adding texture to an object. This is achieved by mapping the contents of a 2D pixel array to the surface of an object which may consist of a single facet or a collection, such as provided in the 3D objects. The texture can also be made subject to the lighting conditions (see page 345).

The process of building up and manipulating 3D models is performed by translating and/or rotating individual components and possibly storing them in hierarchical segment structures for efficient retrieval (see page 423).

GINO also provides the ability to animate 3D objects or complete scenes in 3D space using event based interaction methods (see page 447).

3D Device Drivers

It should be pointed out, that whilst it is possible to display 3D graphics on all GINO output devices, the lighting and shading facilities are only available on devices driving 3D 'hardware'. At the present time , there are two graphics drivers that provide these facilities, namely gWogl (for use under Microsoft Windows and Windows printers) and gGlx (for use under X-Windows). On all other devices, objects will be displayed in the appropriate solid colour, in the order they are drawn, and irrespective of any lighting or special effects that have been defined.
BMP files can also be created from 3D shaded pictures using the gWoglpp() nomination routine.

Performance

There are an increasing number of graphics cards that offer performance acceleration for 3D graphics, either generally or specifically for OpenGL, Direct3D or Glide. It should be noted, however, that unless certain programming guidelines are adopted in a 3D application, a graphics card operating at full acceleration can in fact perform slower than operating without acceleration.

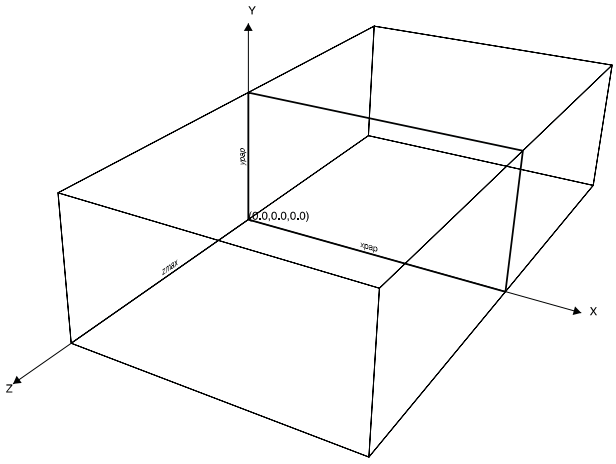
The following guidelines should therefore be adopted in order to achieve maximum performance on any graphics card.

- Use as few light sources as possible
- Store objects in segments and redraw using gDrawSeg()
- In animated objects, only use the facet primitive - most cards are tuned to draw only triangular facets at high speed
- Use as few changes to material properties within an object as possible - i.e. group together facets with the same material
- Cull back facing facets if possible - see gSetShadingMode()
- Switch off back surface lighting - see gSetMaterialIndex()
- Try changing to a lower resolution (some graphics cards do not perform well at very high resolutions such as 1600 x 1200)
- Check the Depth Buffer capabilities of the graphics card and if not 32-bit (GINO's default), change GINO's value by setting the config or environment variable WOGLDEPTH

These points are repeated in the relevant sections in the following chapters.

The 3D World

When a 3D device is initialised, a default 3D coordinate system or 'world' is set up. As has been said, 3D drawing can be performed on any device, whether it drives a simple pen plotter, a monochrome printer or sophisticated 3D graphics card. The initial 3D world that is set up represents a cube with the following layout.



Initial 3D World

Note that the default 3D origin (0.0,0.0,0.0) lies on the surface of the screen or paper, in the bottom left corner.

The limits of these physical dimensions in the X-Y direction can be enquired at any time using the routine gEnqDrawingLimits() which by default will return its measurements in millimetres:

gEnqDrawingLimits(dim,type)

It is possible to both change the default units, and in some instances, the physical drawing limits of the current device (see page 39).

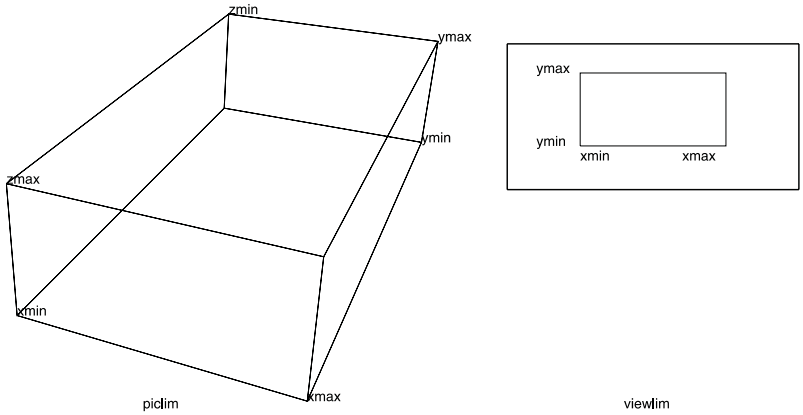
Note that the Z coordinate range is purely notional and in most cases extends from the least to the greatest numerical value possible on the particular hardware/implementation being used. On devices that drive 3D graphics hardware, a range of twice the screen or window width is used so as to define a fitting 3D drawing volume.

3D Viewport Mapping

Whilst drawing to these physical limits may be satisfactory for basic programs, it should be noted that the physical limits will vary from device to device. For example, the limits of a window on a 15" screen will obviously be different on a larger screen and on an A4 piece of paper. Developing a program to cater for these varying limits can be cumbersome and it is therefore useful to be able to map a predefined, application dependent range onto all or part of the physical limits that are available on the current device. This is called viewport mapping and is achieved using the following routine:

gSetViewport3D(piclim,viewlim)

The first argument **piclim** is a structure of type GLIMIT3 containing limits in all three directions (X,Y and Z) that define the users' 3D picture coordinate area or volume. Whereas **viewlim** is a structure of type GLIMIT which defines the area on the device onto which the user limits are to be mapped. These are measured in the current drawing limits as returned by gEnqDrawingLimits(). Any viewport limits outside the device limits will be clipped accordingly.



3D Viewport

Example: The following code maps a 3D volume defined in picture units, on to the available drawing area of the nominated device:

C code

```
#Include <gino-c.h>
GDIM paper;
GLIMIT3 picture = {0.0,1000.0,0.0,750.0,-2000.0,2000.0};
GLIMIT  viewport = {0.0,1.0,0.0,1.0};

      gOpenGino();
      xxxxx();
      gEnqDrawingLimits(paper,ipapty);
/*  Define viewport */
      viewport.xmax=paper.xpap;
      viewport.ymax=paper.ypap;
      gSetViewport3D(picture,viewport);
```

F90 code

```
use gino_f90
type (GDIM) paper
type (GLIMIT3) :: picture = &
GLIMIT3(0.0,1000.0,0.0,750.0,-2000.0,2000.0)
type (GLIMIT)  :: viewport = GLIMIT(0.0,1.0,0.0,1.0)
!
      call gOpenGino
      call xxxxx
      call gEnqDrawingLimits(paper,ipapty)
!  Define viewport
      viewport%xmax=paper%xpap
      viewport%ymax=paper%ypap
      call gSetViewport3D(picture,viewport)
```

Note that, by default, the aspect ratio in X-Y is maintained, and the viewport is centrally placed in the area defined, possibly leaving gaps on the left and right, or top and bottom in which no drawing will take place. This viewport setting can be controlled by the routine:

gSetViewportMode(sw)

In addition to the default setting described above (**sw**=GCENTRAL), the viewport can be placed at the bottom left of the viewport limits (**sw**=GBOTTOMLEFT), or the aspect ratio may be deformed (**sw**= DEFORMED) so that the picture limits are scaled to fit the viewport limits.

The current 3D viewport can be enquired using the following routine:

gEnqViewport3D(piclim,viewlim)

where **piclim** and **viewlim** are of the same type as the setting routine.

When setting up a 3D viewport, the depth (Z) range can affect the accuracy of any depth sorting or clipping so it is advisable not to set this to an arbitrary depth range, but one that suits the application and the objects being drawn appropriately (see page 327).

3D Clipping

The action of defining a viewport will, by default also define a clipping volume, outside which no drawing will take place (unless the routine gSetViewportClipSwitch() has been called). It may be necessary to set up a different or separate clipping volume within the picture limits to control the visible part of the drawing or model. This is achieved using the following routine:

gSetWindow3D(window)

where **window** is a structure of type GLIMIT3 containing limits in all three axes measured in picture units. Note that the limits of the clipping volume will always lie parallel to untransformed X,Y and Z axes.

As in 2-D, user-defined 3-D windows may be switched off/on by using gSetWindowMode().

Enquiring 3D Window Limits

The current state of windowing may be obtained by using the routine:

gEnqWindowState(swi, bounds)

The current setting of the window switch is returned in **swi** and the complete 3D limits are returned in the structure **bounds**.

For example:

```
GLIMIT3 window;
.
gEnqWindowState(&swi,&window);
```

```
type (GLIMIT3) window
.
call gEnqWindowState(swi,window)
```

Chapter 15

3D DRAWING

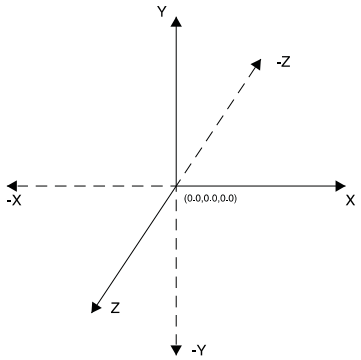
3D Drawing Introduction

GINO provides 3D drawing facilities for:

- Positioning
- Single straight lines
- Polylines
- Polyline sets
- Circular arcs
- B-spline curves
- Bezier curves

3D Axes

The 3D coordinate system used is right-handed as shown below, with the X-axis horizontal, the Y-axis vertical and the Z-axis coming out of the page.



Right Handed Coordinate System

Three-dimensional drawing can be anywhere within X,Y,Z space, with the initial origin being the bottom back left-hand corner of the drawing cube.

3D Start and End Pen Position

All drawing starts from the position at which the pen was left by the previous drawing instruction - this is termed the start pen position. Initially, the position of the pen is at (X,Y,Z) = (0.0,0.0,0.0). The arguments for all 3D drawing routines define the point at which the pen will be left after executing the routine. This is termed the “end pen position”. The end position of one routine becomes the start position for the next. The arguments can specify the absolute coordinates of the end pen position, or the end pen position relative to the start position.

3D Naming Conventions

The naming convention for 3D drawing routines is as follows:

(a) The initial part indicates the routine:

gMove*	- positioning
gDrawLine*	- drawing straight lines
gDrawArc*	- drawing circular arcs
gDrawPolyline*	- drawing series of straight lines
gDrawPolylineSet*	- drawing a set of polylines
gDrawSpline*	- drawing a cubic spline curve
gFillPolygon*	- fill a polygon

(a) The latter part indicates the type of coordinates:

To	- absolute
By	- relative

(c) The last part indicates dimension:

**2D	- two dimensions (see page 77)
**3D	- three dimensions

3D Positioning

The routines for “straight line movement” are:

gMoveTo3D(x, y, z)

gMoveBy3D(dx, dy, dz)

Examples:

- To position the pen at point (1.5,2.5,3.5) the following statement could be used:

```
gMoveTo3D(1.5,2.5,3.5);
```

```
call gMoveTo3D(1.5,2.5,3.5)
```

- To increment the start pen position by **xa** in the X-direction, **ya** in the Y-direction and **za** in the Z direction the following statement could be used:

```
gMoveBy3D(xa,ya,za);
```

```
call gMoveBy3D(xa,ya,za)
```

3D Straight Lines

The routines for drawing straight lines are:

gDrawLineTo3D(x, y, z)

gDrawLineBy3D(dx, dy, dz)

For example - to draw a straight line from the point (50.0,20.0,-10.0) to the point (60.0,80.0,200.0) the following statements can be used:

```
gMoveTo3D(50.0,20.0,-10.0);
gDrawLineTo3D(60.0,80.0,200.0);
```

```
call gMoveTo3D(50.0,20.0,-10.0)
call gDrawLineTo3D(60.,80.,200.)
```

3D Polylines

The routines for drawing 3D multiple straight lines from the current pen position are:

gDrawPolylineTo3D(npts, points3)

gDrawPolylineBy3D(npts, points3)

where **points3** is an array of structures of type GPOINT3 each containing three real elements representing the x, y and z coordinates in either absolute or relative terms.

For example - to draw the 3D arrow head shown below, an array **points3** of type GPOINT3 is initialized with four coordinate as shown below:

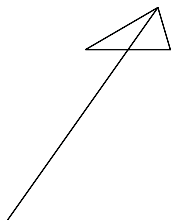
C Code

```
GPOINT3 arrow[4] = {7.07,10.0,7.07, 7.65,8.0,3.65,
                    3.65,8.0,7.65, 7.07,10.0,7.07};

   gMoveTo3D(0.0,0.0,0.0);
```

```
   gDrawPolylineTo3D(4,arrow);
```

F90 Code

```
type (GPOINT3) :: arrow(4) = (/ &
      GPOINT3(7.07,10.0,7.07), GPOINT3(7.65,8.0,3.65), &
      GPOINT3(3.65,8.0,7.65),  GPOINT3(7.07,10.0,7.07) /)

   call qMoveTo3D(0.0,0.0,0.0)
   call gDrawPolylineTo3D(4,arrow)
```



3D Polyline

The same figure could have been produced using the routine gDrawPolylineBy3D() as follows:

C Code

```
GPOINT3 arrow[4] = {7.07,10.0,7.07, 0.58,-2.0,-3.42,
                    -4.0,0.0,4.0, 3.42,2.0,-0.58};

   qMoveTo3D(0.0,0.0,0.0);
   gDrawPolylineBy3D(4,arrow);
```

F90 Code

```
type (GPOINT3) :: arrow(4) = (/ &
      GPOINT3(7.07,10.0,7.07), GPOINT3(0.58,-2.0,-3.42), &
      GPOINT3(-4.0,0.0,4.0),  GPOINT3(3.42,2.0,-0.58) /)

   call qMoveTo3D(0.0,0.0,0.0)
   call gDrawPolylineBy3D(4,arrow)
```

Shaded Polylines

To draw a polyline that is affected by lighting and shading users are referred to the section on 3D objects (see page 305).

3D Polyline Sets

3D Polyline Set Definition

A polyline set consists of an array of polylines each of which consists of an integer number of vertices and a pointer to an array of 3D vertices.

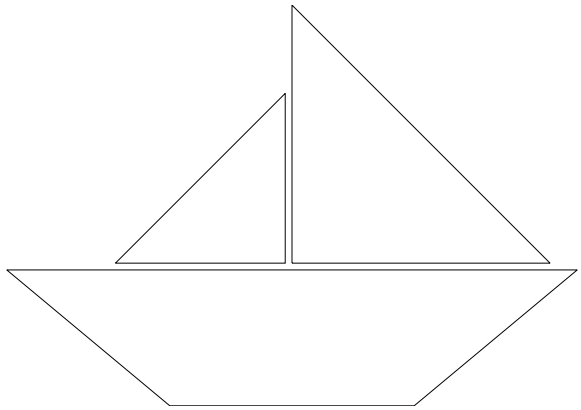
Each polyline is complete within itself and does not make use of the current pen position. For this reason polygon sets can only use absolute coordinates.

An example of a 3-D polyline set consisting of a trapezium and two triangles is represented by the following coordinates and shown in the diagram below:

	1	2	3	4	5	6	7	8	9	10	11	12	13
x:	40.	160.	340.	460.	40.	120.	245.	245.	120.	250.	440.	250.	250.
y:	140.	40.	40.	140.	140.	145.	270.	145.	145.	145.	145.	335.	145.
z:	0.0	0.0	0.0	0.0	0.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0

Polyline sizes

5 4 4



3D Polyline Set

3D Polyline Usage

Three dimensional polyline sets are filled using the following routine.

gDrawPolylineSet3D(npol, polylines3)

where **npol** is the number of polylines contained in the GPOLYGON3 array **polylines3**.

The example polyline sets described previously can be implemented as follows.

C code

```
static GPOLYGON3 poly[3] = {5, 0, 4, 0, 4, 0};
static GPOINT3 points[13] = {
      40.0,140.0,0.0, 160.0,40.0,0.0,
      340.0,40.0,0.0, 460.0,140.0,0.0, 40.0,140.0,0.0,

      120.0,145.0,10.0, 245.0,270.0,10.0,
      245.0,145.0,10.0, 120.0,145.0,10.0,
      250.0,145.0,10.0, 440.0,145.0,10.0,
      250.0,335.0,10.0, 250.0,145.0,10.0};
main()
{
   poly[0].verts=&points[0];
   poly[1].verts=&points[5];
   poly[2].verts=&points[9];

   gDrawPolylineSet3D(3,poly);
}
```

F90 code

```
type (GPOLYGON3) :: poly(3)
type (GPOINT3)   :: points(13) = {/ &
         GPOINT3(40.0,140.0,0.0),  GPOINT3(160.0,40.0,0.0), &
         GPOINT3(340.0,40.0,0.0),  GPOINT3(460.0,140.0,0.0), &
         GPOINT(40.0,140.0,0.0), &
         GPOINT3(120.0,145.0,10.0),GPOINT3(245.0,270.0,10.0), &
         GPOINT3(245.0,145.0,10.0),GPOINT(120.0,145.0,10.0), &
         GPOINT3(250.0,145.0,10.0),GPOINT3(440.0,145.0,10.0), &
         GPOINT3(250.0,335.0,10.0),GPOINT(250.0,145.0,10.0) /)

   poly(1)%nvert=5
   poly(1)%verts=>points(1:5)
   poly(2)%nvert=4
   poly(2)%verts=>points(6:9)
   poly(3)%nvert=4
   poly(3)%verts=>points(10:13)
   .
   call gDrawPolylineSet3D(3,poly)
```

3D Arcs

The routines for drawing 3D circular arcs are:

gDrawArcTo3D(xc, yc, zc, xe, ye, dze, dxt, dyt, dzt)

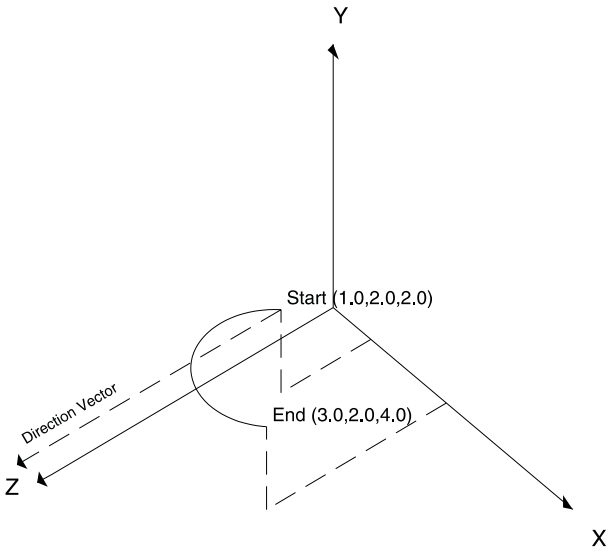
gDrawArcBy3D(dxc, dyc, dzc, dxe, dye, dze, dxt, dyt, dzt)

All arcs are drawn from the start pen position. The radius of an arc is the distance from the start point to the centre. The end pen position or any point on the straight line from the centre through the end point of the arc may be specified. The end pen position will then be calculated.

In 3-D the direction of the arc is indicated by specifying a “direction vector”. If a circle or semicircle is being drawn this vector is used to specify the plane in which the arc lies and the direction in which it has to be drawn.

Example:

- To draw a horizontal semicircle from point (1.0,2.0,2.0), centre (2.0,2.0,3.0) and end point (3.0,2.0,4.0):



3-D Semicircle

```
/* Move to start */
   gMoveTo3D(1.0,2.0,2.0);
/* Direction vector */
   dtx =  0.0;
   dty =  0.0;
   dtz = 10.0;
   gDrawArcTo3D(2.0,2.0,3.0,
      3.0,2.0,4.0,dtx,dty,dtz);
```

```
! Move to start
  call gMoveTo3D(1.0,2.0,2.0)
! Direction vector
  dtx =  0.0
  dty =  0.0
  dtz = 10.0
  call gDrawArcTo3D(2.0,2.0, &
    3.0,3.0,2.0,4.0,dtx,dty,dtz)
```

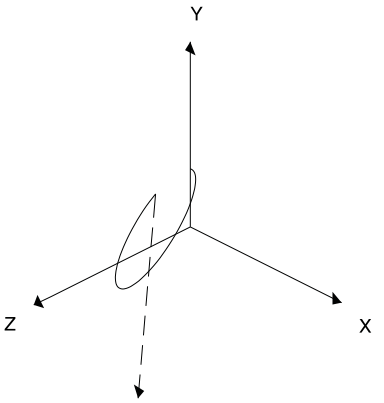
If the arc to be drawn is not a circle or semicircle, then the start and end point together with the centre point specify the plane in which the arc is to lie. In this case the direction vector merely indicates whether a major or minor arc is required.

Example:

- To draw an arc from point (1.0,2.0,2.0) with centre (2.0,2.0,3.0) and end point on the line from the centre (2.0,3.0,2.0). If the direction vector is (0.0,-1.0,0.0) then the major arc is drawn and if the direction vector is (0.0,1.0,0.0) the minor arc is drawn (see below).

```
gMoveTo3D(1.0,2.0,2.0);
gDrawArcTo3D(2.0,2.0,3.0,
2.0,3.0,2.0, 0.0,-1.0,0.0);
```

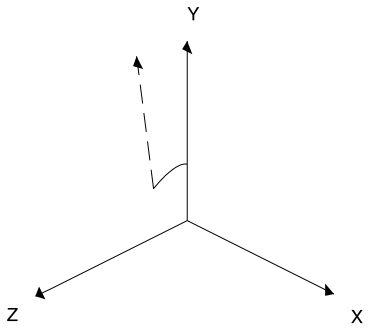
```
call gMoveTo3D(1.0,2.0,2.0)
call gDrawArcTo3D(2.0,2.0,3.0, &
2.0,3.0,2.0, 0.0,-1.0,0.0)
```



3-D Direction Vector (Major arc)

```
gMoveTo3D(1.0,2.0,2.0);
gDrawArcTo3D(2.0,2.0,3.0,
     2.0,3.0,2.0, 0.0,1.0,0.0);
```

```
call gMoveTo3D(1.0,2.0,2.0)
call gDrawArcTo3D(2.0,2.0,3.0, &
       2.0,3.0,2.0, 0.0,1.0,0.0)
```



3-D Direction Vector (Minor arc)

The same arcs can be drawn using the gDrawArcBy3D() routine using vector increments as shown below:

For the major arc:

```
gMoveTo3D(1.0,2.0,2.0);
gDrawArcBy3D(1.0,0.0,1.0,
   1.0,1.0,0.0, 0.0,-1.0,0.0);
```

```
call gMoveTo3D(1.0,2.0,2.0)
call gDrawArcBy3D(1.0,0.0,1.0, &
   1.0,1.0,0.0, 0.0,-1.0,0.0)
```

For the minor arc:

```
gMoveTo3D(1.0,2.0,2.0);
gDrawArcBy3D(1.0,0.0,1.0,
   1.0,1.0,0.0, 0.0,1.0,0.0);
```

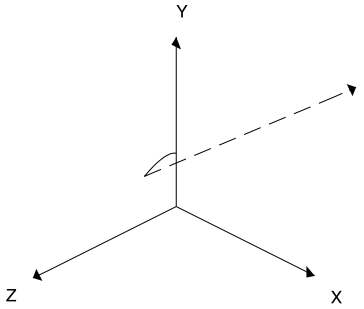
```
call gMoveTo3D(1.0,2.0,2.0)
call gDrawArcBy3D(1.0,0.0,1.0, &
   1.0,1.0,0.0,0.0,1.0,0.0)
```

Direction Vector

Identical 3D arcs can be obtained using different direction vectors.
For Example:

- The direction vector in the major-arc example above could have been specified as (0.0,-100.0,0.0) showing that the magnitude is not significant.

- The minor arc could have been produced if the direction vector had been (0.0,0.0,-10.0) showing the plane is not significant (see below).



3-D Direction Vector

3D Spline Curves

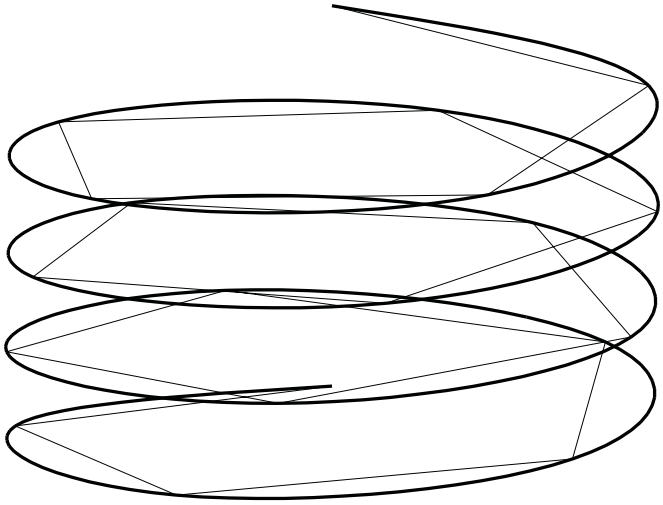
GINO provides two routines to draw a smooth curve through a series of 3D points using cubic splines:

gDrawSplineTo3D(npts, points3, beg, fin)

gDrawSplineBy3D(npts, points3, beg, fin)

Where **points3** is an array of type GPOINTS3 containing the three components of the point in space.

An example of a 3D spline curve is shown below, indicating the fitting of a curve to a 20 point helix.



Spline curve using 20 data points

3D Spline Curve Control

End conditions, increments and tension for 3D spline curves are set in the same manner as for 2D splines curves (see page 77).

The routines gSetCurveAttribs3D()/gEnqCurveAttribs3D() are used to set and enquire the end conditions for 3D spline curves. Thus:

gSetCurveAttribs3D(dxbeg, dybeg, dzbeg, dxfin, dyfin, dzfin, xbeg, ybeg, zbeg, xfin, yfin, zfin)

gEnqCurveAttribs3D(dxbeg, dybeg, dzbeg, dxfin, dyfin, dzfin, xbeg, ybeg, zbeg, xfin, yfin, zfin)

3D Bezier Curves

The Bezier curve routines offer a very different kind of curve control where the data supplied represents control points rather than points on the curve itself. Two routines are provided for this kind of curve drawing in 3D:

gDrawBezierTo3D(npts, points3)

gDrawBezierBy3D(npts, points3)

where **points3** is an array of type GPOINT3 and **npts** are the number of control points stored in the array.

3D Elevation and Reduction

As in the case of 2D Bezier curves, two routines are also provided to elevate and reduce a 3D Bezier curve definition:

gElevateBezier3D(npts, points3)

gReduceBezier3D(npts, points3)

Users are referred to the section on 2D Bezier curves for additional information on the usage of all these routines (see page 101)

3D Polygons

The routines for filling 3D polygons are:

gFillPolygonTo3D(fill, line, inv, npts, points3)

gFillPolygonBy3D(fill, line, inv, npts, points3)

gFillPolygonSet3D(fill, line, inv, npol, polygons3)

The routines gFillPolygonTo3D(), and gFillPolygonBy3D() fill a single polygon, starting at the current pen position, containing **npts** vertices of either absolute or relative points of type GPOINT3 in the array of **points3**.

The routine gFillPolygonSet3D() fills a set of **npol** polygons contained in the array **polygons3** of type GPOLYGON3. Each polygon is self contained without making reference to the current pen position. An example of filling and using an array of type GPOLYGON3 is described above with the description of gDrawPolylineSet3D().

In all three cases, an extra point is added if necessary to ensure the polygon is closed before filling. The fill style and line style are defined in the same way as gFillRect(), using **fill** and l**ine** arguments (see page 165).

The argument **inv** specifies which area is to be filled. When **inv**=GAREA the interior of the polygon is filled and when **inv**=GINVERSE the exterior area up to the current window limits is filled, leaving the interior empty. If the polygon is self intersecting, unfilled areas can be created within a polygon.

Overlapping Polygons

GINO applies all the current modelling and viewing transformations to the points in the polygon or polygon set to create a set of points in 2-D (see page 385). This polygon set is filled in the same way that a 2-D polygon would be filled. Therefore, if any parts of the 3-D polygon overlap when viewed using the current settings, unfilled areas can be created within the filling.

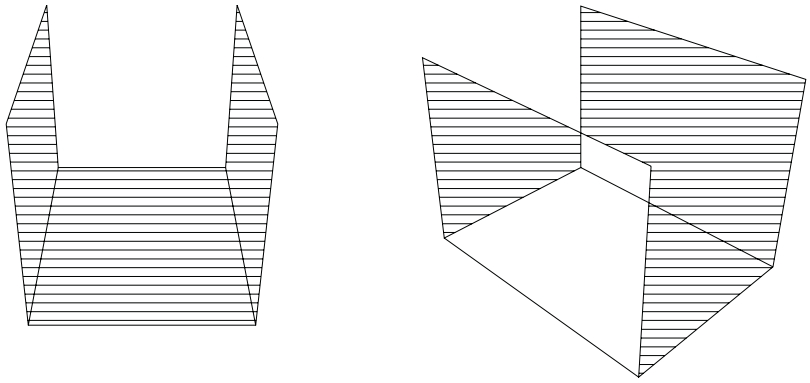
The following example shows two views of the same object displayed using 3-D polygon filling. In the first view none of the shape's faces overlap any other, in the second view overlapping takes place and causes an unfilled area to be created within the generated 2-D polygon:

C code

```
static GPOINT3 pts[8] =
            {80.0, 0.0,60.0, 80.0, 0.0, 0.0,
             80.0,80.0, 0.0,  0.0,80.0, 0.0,
              0.0, 0.0, 0.0,  0.0, 0.0,60.0,
              0.0,80.0,60.0, 80.0,08.0,60.0};
 .
 .
 gDefinePerspView(665.,405.,90.,-6.65,-4.05,0.,784.);
 gGenerateView();
 .
 .
 gMoveTo3D(pts[7].x,pts[7].y,pts[7].z);
 gFillPolygonTo3D(1,0,GAREA,8,&pts);
 .
 .
 gSetTransform(-1);
 gDefinePerspView(475.,405.,475.,-4.75,-4.05,-4.75,784.);
 gGenerateView();
 .
 .
 gMoveTo3D(pts[7].x,pts[7].y,pts[7].z);
 gFillPolygonTo3D(1,0,GAREA,8,pts);
 .
```

F90 code

```
type (GPOINT3) :: pts(8) = (/ &
      GPOINT3(80.0, 0.0,60.0), GPOINT3(80.0, 0.0, 0.0), &
      GPOINT3(80.0,80.0, 0.0), GPOINT3( 0.0,80.0, 0.0), &
      GPOINT3( 0.0, 0.0, 0.0), GPOINT3( 0.0, 0.0,60.0), &
      GPOINT3( 0.0,80.0,60.0), GPOINT3(80.0,08.0,60.0} /)
 .
 .
 call gDefinePerspView(665.,405.,90.,-6.65,-4.05,0.,784.)
 call gGenerateView
 .
 .
 call gMoveTo3D(pts(8)%x,pts(8)%y,pts(8)%z)
 call gFillPolygonTo3D(1,0,GAREA,8,pts)
 .
 .
 call gSetTransform(-1)
 call gDefinePerspView(475.,405.,475.,-4.75,-4.05,-4.75,784.)
 call gGenerateView
 .
 .
 call gMoveTo3D(pts(8)%x,pts(8)%y,pts(8)%z)
 call gFillPolygonTo3D(1,0,GAREA,8,pts)
```



Two views of the same 3-D polygon

The shapes' edges are added using gMoveTo3D() and gDrawLineTo3D().

Users are referred to the Facet primitive for true 3D polygon display (see page 295).

3D Point Storage

In a similar manner as the 2D drawing routines, all the vertices generated by the 3D drawing routines are stored when point or polygon storage is switched on and can be used for the definition of polygons or returning to the application. These facilities are described in the 2D Drawing section of the document (see page 103).

The routine to return all 3 coordinates of the stored points is:

nret=gReturnInternalPoints3D(nn, points3, np, polylines3, npts, npol)

where **points3** is an array of type GPOINT3 and **polylines3** is an array of type GPOLYGON3. The arguments **nn** and **np** should be set to the size of these arrays. The arguments **npts** and **npol** return the number of points and polylines that actually exist in the internal workspace which may be more than those returned if the supplied arrays are not sufficiently large enough. The function itself returns the actual number of complete polylines that have been placed in the user supplied arrays.

Further details can be found in the above referenced section of this document.

3D Interpolation

GINO provides a facility to interpolate user suppled 3D data or from previously drawn 3D curves, lines or arcs using the above point storage mechanism. Passing a single data value with a set of 3D data points, the function gInterpolateData3D() can return all the intersections using linear interpolation.

The function has the following form:

nint=gInterpolateData3D(nopt, ptint, npts, points3, nptout, ptout1, ptout2)

where **nopt** can be GXDATA, GYDATA or GZDATA indicating the interpretation of the argument **ptint**, the value to be interpreted. The argument **npts** specifies the number of 3D data points supplied in the array **points3** (which is of type GPOINT3) and **nptout** is the size of the output arrays **ptout1** and **ptout2**.

The function returns the number of intersection points returned in the arrays **ptout1** and **ptout2**. Where **nopt**=GXDATA these arrays will contain Y and Z values, where **nopt**=GYDATA they will contain X and Z values and where **nopt**=GZDATA they will contain X and Y values respectively. There may be zero, one or more than one depending on the form of the data, but it will never exceed **nptout** even though there may be more intersections possible from the supplied data.

Chapter 16

FACETS

Facets Introduction

A facet is a special kind of polygon that can be made subject to the current lighting and shading environment (see page 325). Without lighting and shading, the facet is displayed in the current colour as either a filled polygon or its boundary depending on its fill style. The facets reaction to light is defined in terms of its material properties which are used instead of its colour when lighting and shading is enabled (see page 339).

In the same way as a normal polygon, a facet is defined by a set of vertices in 3D space, but because of its association with lighting and shading the facet should be defined as either triangular (3 vertices) or quadrilateral (4 vertices). The primitive is not actually limited to this number or vertices, but in order to be able to correctly calculate the true effect of light shining on the facet, the vertices should be planar (i.e. All the vertices lie in one plane), hence the preference for triangular and quadrilateral facets. It is also often true that graphics hardware is also tuned to the display of these simpler facets, giving a much improved performance when objects are limited to them.

In addition to the facet vertices, this primitive can have additional information associated with the vertices which determine the actual appearance of the facet in the current lighting and shading environment. These include non-planar normals, texture coordinates or vertex colours. These are described in the following sections.

Facet Definition

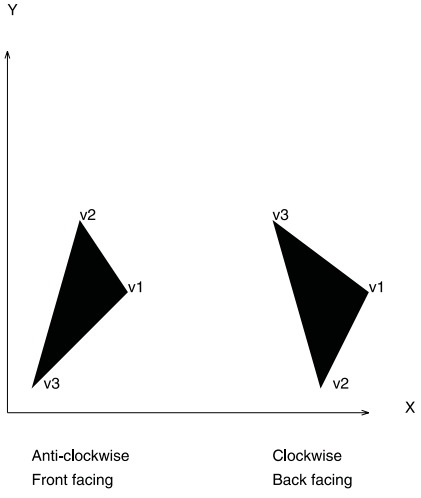
In its simplest form, a facet is defined as an array of 3D vertices stored in an array of structures of type GPOINT3 using the following routine:

gDrawFacet(npts, points, [gNormals, gTextCoords, gColours].)

where **points** is an array of 3D vertices and **npts** is the number of vertices in the array. The arguments **gNormals**, **gTextCoords** and **gColours** are all optional and are described below.

Facet Faces

Before describing the additional facet forms, it is important to note that unlike normal polygons, facets have two faces, a front and back face, each of which can be given different material attributes and therefore appear differently when viewed from either direction. The order of the vertices as viewed by the viewer is used to determine which face is which. By default, vertices lying in an anti-clockwise order are deemed to be showing the front face, and therefore if you move round to the other side, the vertices will lie in a clockwise order, and so you will be looking at the back of the facet. This is known as the winding rule. Therefore the order in which the vertices are passed to the facet drawing routine are important in determining the 'structure' of a single or a group of facets.



Facet Faces

Note the default winding rule can be reversed when setting up the lighting and shading environment (see page 325).

Normals

The method used to calculate the correct shade of a facet in any lighting environment uses the angle that the facet lies in relation to the direction the light is shining. In actual fact, the underlying mathematics uses a vector which is perpendicular to the plane of the facet, known as its *normal*. GINO will automatically calculate this vector (unless alternative vectors are supplied) based on the position of the first 2 and last vertices in each facet. This single normal is called a *planar normal* because it applies to the whole plane of the facet.

This single vector is sufficient when flat shading is being used or where the facet represents a flat surface. If, however, curved surfaces are being constructed, and smooth shading is required, a more accurate definition of the facet is required to give an accurate visual appearance. Under these circumstances normals need to be calculated for each vertex by averaging planar normals of adjacent facets and then supplied to the gDrawFacet() routine in the **gNormals** optional array argument.

GINO provides a routine to return the *planar normal* of a set of vertices that may then be manipulated as required by the application.

gReturnPlanarNormal(npts, vertices, normal)

where **vertices** is an array of **npts** points of type GPOINT3 and the planar **normal** is returned similarly in a structure of type GPOINT3. The following example shows the smoothing of two adjoining facets by averaging their planar normals:

C Code

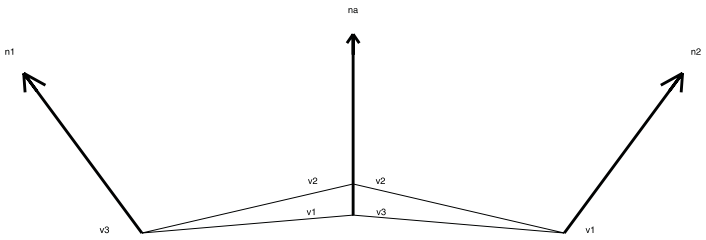
```
/* Averages planar normals for smooth surface */
include <gino-c.h>

GPOINT3 facet1[] = {60.0,30.0,20.0 ,60.0,30.0,10.0 ,20.0,10.0,5.0 },
        facet2[] = {110.0,10.0,5.0 ,60.0,30.0,10.0 ,60.0,30.0,20.0};
GPOINT3 normal1(3),normal2(3),n1,n2,na;

main()
{
   gOpenGino();
   XXXXX();
```

```
/* Get planar normals */
   gReturnPlanarNormal(3,facet1,n1);
   gReturnPlanarNormal(3,facet2,n2);

/* Average normal */
   na.x=(n1.x + n2.x)/2.0;
   na%y=(n1.y + n2.y)/2.0;
   na%z=(n1.z + n2.z)/2.0;

/* Generate facet with user supplied normals */
   normal1[0]=na;
   normal1[1]=na;
   normal1[2]=n1;

   normal2[0]=n2;
   normal2[2]=na;
   normal2[3]=na;

/* Draw facets */
   gDrawFacet(3,facet1,gNormals,normal1,0);
   gDrawFacet(3,facet2,gNormals,normal2,0);

   gSuspendDevice();
   gCloseGino();
}
```

F90 Code

```
! Averages planar normals for smooth surface
use gino_f90
!
type (GPOINT3) :: facet1(3) = &
      (/ GPOINT3(60.0,30.0,20.0), &
         GPOINT3(60.0,30.0,10.0), &
         GPOINT3(20.0,10.0,5.0) /)
type (GPOINT3) :: facet2(3) = &
      (/ GPOINT3(110.0,10.0,5.0), &
         GPOINT3(60.0,30.0,10.0), &
         GPOINT3(60.0,30.0,20.0) /)
type (GPOINT3) :: normal1(3),normal2(3),n1,n2,na
!
   call gOpenGino
   call XXXXX
!
!  Get planar normals
   call gReturnPlanarNormal(3,facet1,n1)
   call gReturnPlanarNormal(3,facet2,n2)
!
!  Average normal
   na%x=(n1%x + n2%x)/2.0
   na%y=(n1%y + n2%y)/2.0
   na%z=(n1%z + n2%z)/2.0
!
!  Generate facet with user supplied normals
   normal1(1)=na
   normal1(2)=na
   normal1(3)=n1
!
   normal2(1)=n2
   normal2(2)=na
   normal2(3)=na
```

```
!
!  Draw facets
   call gDrawFacet(3,facet1,gNormals=normal1)
   call gDrawFacet(3,facet2,gNormals=normal2)
!
   call gSuspendDevice
   call gCloseGino
   stop
   end
```



Averaging Normals

Textured Facet

In addition, or as an alternative, to the lighting effects on a facet, one or more facets may be ‘covered’ with a texture. This powerful technique can be used to ‘drape’ a predefined image over a series of facets to add a textured appearance to an object. The image may represent some abstract pattern, some additional 4D data or a photo realistic image, but in all cases the data is supplied in the same form as for a pixel image described earlier in this document (see page 189).

In these circumstances, the facet definition may need to define its location in relation to the image and this is achieved by supplying texture coordinates in the optional **gTextCoord** array argument to the gDrawFacet() routine. The texture mapping facility is fully described later in this document (see page 345).

Coloured Facet

A third type of facet is provided by the GINO library, that is not affected by lighting or texture mapping. If the gDrawFacet() routine is supplied with specific colours for each vertex in the optional **gColours** array, the facet will be drawn with its interior (or boundary) with graduated colours between those specified.

Therefore, along one edge, if one vertex is specified as red, and the other as yellow, the edge will be drawn with each displayable point along the edge changing from red through orange to yellow. The same applies to the interior of the facet where three or more vertices are defined.

Colours may be supplied as indices in the current colour table or as 24bit RGB triplets using the gTrueCol() function.

An example of drawing a coloured facet is shown below:

C Code

```
/* draws coloured facets */
include <gino-c.h>

GPOINT3 facet1[] = {60.0,30.0,20.0 ,60.0,30.0,10.0 ,20.0,10.0,5.0 },
        facet2[] = {110.0,10.0,5.0 ,60.0,30.0,10.0 ,60.0,30.0,20.0};
int     cols1[] = { 2,4,5 };
int     cols2[] = { 7,4,2 };

main()
{
   gOpenGino();
   XXXXX();

/* Set up view */
   xeye=60.0;
   yeye=300.0;
   zeye=100.0;
   gDefinePerspView(xeye,yeye,zeye,0.0,-yeye,-zeye,800.0);
   gGenerateView();

/* Define smooth shading */
   gSetShadingMode(GGOURAUD,0);

/* Draw two coloured facets */
   gDrawFacet(3,facet1,gColours,cols1,0);
   gDrawFacet(3,facet2,gColours,cols2,0);

   gSuspendDevice();
   gCloseGino();
}
```

F90 Code

```
! draws coloured facets
use gino_f90
!
type (GPOINT3) :: facet1(3) = &
      (/ GPOINT3(60.0,30.0,20.0), &
         GPOINT3(60.0,30.0,10.0), &
         GPOINT3(20.0,10.0,5.0) /)
type (GPOINT3) :: facet2(3) = &
      (/ GPOINT3(110.0,10.0,5.0), &
         GPOINT3(60.0,30.0,10.0), &
         GPOINT3(60.0,30.0,20.0) /)
integer        :: cols1(3) = (/ 2,4,5 /)
integer        :: cols2(3) = (/ 7,4,2 /)
```

```
!
   call gOpenGino
   call XXXXX
!
!  Set up view
   xeye=60.0
   yeye=300.0
   zeye=100.0
   call gDefinePerspView(xeye,yeye,zeye,0.0,-yeye,-zeye,800.0)
   call gGenerateView
!
!  Define smooth shading
   call gSetShadingMode(GGOURAUD)
!
!  Draw two coloured facets
   call gDrawFacet(3,facet1,gColours=cols1)
   call gDrawFacet(3,facet2,gColours=cols2)
!
   call gSuspendDevice
   call gCloseGino
   stop
   end
```

Coloured facets

Note that smooth colour graduations between vertices will only occur if GGOURAUD shading is set in gSetShadingMode().

Facet Attributes

Facet Fill Style

Facets can be drawn in two different styles, either solid filled or a series of lines linking the vertices (the boundary). This attribute is set using the routine:

gSetFacetFillStyle(fill)

where **fill** can be either GSOLID (default) or GHOLLOW. In both cases, the facet is drawn using the current colour or material properties.

The current facet fill style can be enquired using the following routine:

gEnqFacetFillStyle(fill)

As it is not possible to define a different coloured boundary to that of the centre, if two such colours are required then the facet needs to be drawn twice with different colour attributes (but see below).

Facet Offset

Where a facet boundary needs to be displayed as well as its interior, or some additional detail needs to be added to the 'surface' of an object, care needs to be taken as to the hidden surface mechanism used in GINO. Full details of the depth buffering technique used to display 3D objects is given later in this document (see page 327), suffice to say that the default is to only display information that is nearer to the viewer. This has the effect of removing detail at the same distance from the viewer (unless the default depth buffering mode is changed) subject to the accuracy of the Z buffer.

A useful alternative to the modification of the depth buffer, is to specify a nominal offset to either the facet's interior or its boundary as necessary. This can be achieved using the following routine:

gSetFacetOffsetMode(mode)

where **mode** can be any of the following:

GOFF	Switch off all offsets
GBOUNDARYAWAY	Shift boundary away from viewer
GINTERIORAWAY	Shift interior away from viewer
GINTERIORNEAR	Shift interior nearer viewer
GBOUNDARYNEAR	Shift boundary nearer viewer

In the simple case of requiring the visibility of both the facet and its boundary, the boundary can be 'shifted' nearer the viewer by setting the offset mode to GBOUNDARYNEAR. Where additional surface detail is required, possibly drawn using gDrawPolylineTo3D(), shifting the interior away from the viewer would be the preferred option (mode = GINTERIORAWAY).

The current facet fill style can be enquired using the following routine:

gEnqFacetOffsetMode(mode)

Chapter 17

3D OBJECTS

3D Objects Introduction

GINO provides facilities to draw a set of 3D objects which (apart from the shaded polyline) are generated from facets positioned in 3D space. Such objects may be used to construct complex models according to the current viewing and modelling transformations.

Shaded Polyline:

Solid 3D primitives:

- Box
- Wedge
- Cone
- Cylinder
- Sphere
- Volume

Surface primitives:

- Spline Surface
- Bezier surface
- Ruled Bezier surface
- Tabulated Bezier surfaces
- Swept Bezier surface
- Bezier sphere
- Bezier volume

Local Axes System

All objects and surfaces (except those mentioned below) are defined in a local axes system U,V and W. For 3D primitives these are aligned along the current X,Y and Z axes, whereas for Bezier surfaces, the U axis is aligned along the major curve and the V axis is aligned along the extrusion.

All the solid object routines require a set of obligatory arguments to define their position in 3D space, together with certain dimensions (some of which default to 1.0). Alternative dimensions and orientations are specified using optional arguments for width, height, depth or radii together with local rotations about the U, V and or W axes, or as absolute or relative vectors in the local axis system.

Shaded polylines, Spline and Bezier surfaces are fully defined in 3D space by their data/control points, but the Bezier sphere and volumes of rotation can have optional local rotations.

Object Complexity

Objects of rotation and surfaces are built up from a number of quadrilateral facets in each of the U and V axes (representing the circumferential and vertical dimensions for volumes of rotation). This can be altered using optional arguments in the appropriate routines to increase or decrease the smoothness of the object as required. The exception to this is for normal volumes of rotation where the vertical complexity is determined by the number of points in the supplied outline.

Object Shading

All 3D objects are shaded according to the current lighting/shading environment with automatic generation of planar or averaged normals as required by the current shading mode (see page 325). Objects are drawn using the current facet attributes (see page 301) and colour or material properties (see page 339).

Object Texture Mapping

Individual objects are assigned texture mapping coordinates in the range 0.0 to 1.0 where texture mapping has been switched on prior to the drawing of the object (see page 345). This allows the draping of complete image files over each object with ease. Where images are required to be replicated over one object or to cover several objects, it is the responsibility of the application writer to generate appropriate texture coordinates using the texture coordinate generation routines (see page 349).

Shaded Polyline

The shaded polyline is a special 3D object that is fully defined through its arguments. It is distinct from the normal polyline in that it is subject to the current lighting, material and texture mapping settings and is distinct from the other objects in this section in that it is not constructed from facets.

The routine to draw such a polyline is:

gDrawShadedPolylineTo3D(npts, points, normals, [gTextCoords])

where **npts** are the number of vertices defining the polyline and **points** and **normals** are arrays of type GPOINT3 which define the coordinates and the normals at each vertex. An optional argument **gTextCoords** can also be supplied if texture mapping is to be applied to the polyline.

The shaded polyline is useful where linear detail needs to be added to a scene with objects and/or surfaces and where the detail needs to respond to the current lighting conditions. It is important that the correct normals are supplied with this routine, usually indicating the normals of the associated facets on which the detail is being added.

Note that all other polylines, polygons and rectangles are drawn with lighting and texture mapping switched off and thus appear in their specified colour.

3D Primitives

The following sections describe the drawing of each of the solid 3D primitives.

Boxes

Three forms of a box primitive are provided:

gDrawRect3D(xmin, xmax, ymin, ymax, zmin, zmax)

gDrawCube(xc, yc, zc, dim, [gURot, gVRot, gWRot, gUComp, gVComp, gWComp])

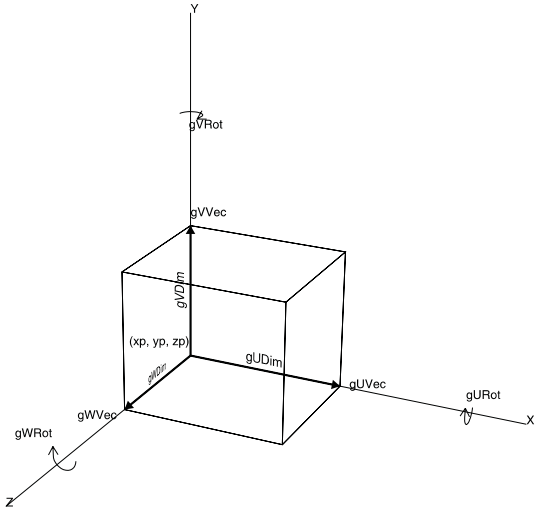
gDrawBox(xp, yp, zp, [gUDim, gVDim, gWDim, gURot, gVRot, gWRot, gUVec, gVVec, gWVec, gAbs, gUComp, gVComp, gWComp])

The gDrawRect3D() routine draws a box whose edges are aligned along the current axes and does not have a local orientation. The gDrawCube() routine draws a cube centred at the specified position, whereas gDrawBox() is a generalised box routine, the dimension and orientation of which can be specified in one of three different ways.

- Dimensions and rotations – using optional arguments **gUDim**, **gVDim**, **gWDim** together with optional local rotations set about the U, V or W axes using arguments **gURot**, **gVRot** and **gWRot**
- Absolute vectors – using optional arguments **gUVec**, **gVVec**, **gWVec** with **gAbs** set to GABSOLUTE
- Relative vectors – using optional arguments **gUVec**, **gVVec**, **gWVec** with **gAbs** set to GRELATIVE

Where absolute or relative vectors are used, these must be mutually perpendicular in order to maintain the cuboid shape.

Each of the obligatory and optional arguments are shown in the figure below.



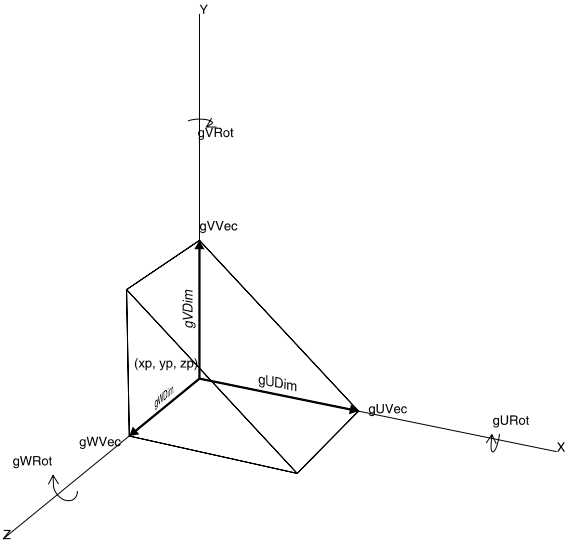
Box Primitive

The optional arguments **gUComp**, **gVComp** and **gWComp** determine the number of sub-divisions of each face in the U,V or W directions respectively. The default in each case is 1.

Wedges

A wedge is similar to a box except that one corner is cut between the upper left and lower right corners as shown below:

gDrawWedge(xp, yp, zp, [gUDim, gVDim, gWDim, gURot, gVRot, gWRot, gUVec, gVVec, gWVec, gAbs])



Wedge Primitive

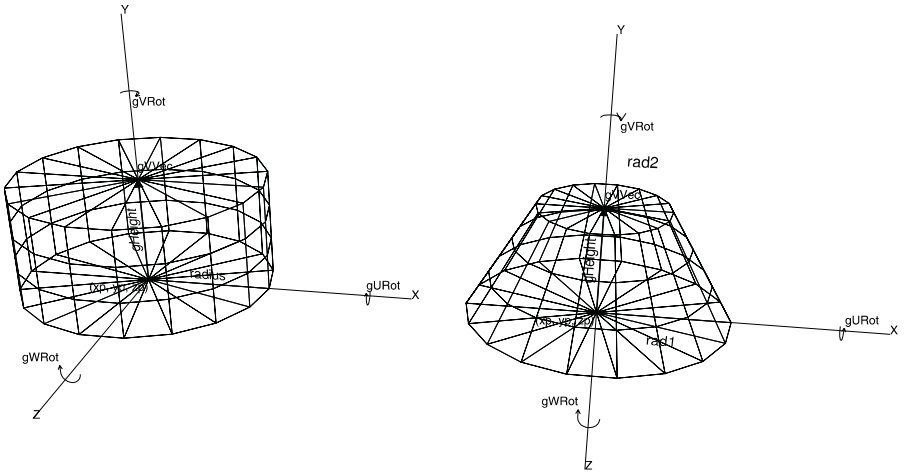
Cylinders and Cones

Cones and cylinders can be drawn using the following routines:

gDrawCylinder(xp, yp, zp, radius, [gHeight, gURot, gVRot, gWRot, gVVec, gAbs, gUComp, gVComp])

gDrawCone(xp, yp, zp, rad1, rad2, [gHeight, gURot, gVRot, gWRot, gVVec, gAbs, gUComp, gVComp])

where **xp**, **yp**, **zp** is the centre of the base of each object and the radii are specified as shown below.



Cone and Cylinder Primitives

Note that the height of either object may be specified using the optional argument **gHeight** together with optional local rotations set about the U, V or W axes using arguments **gURot**, **gVRot** and **gWRot**, or by using **gVVec** as an absolute or relative vector from the object's origin (i.e. the centre of the base).

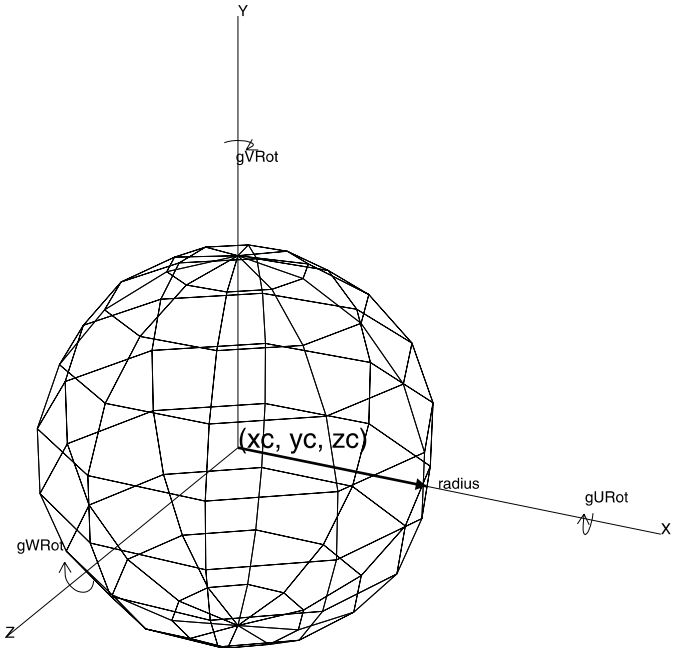
The optional arguments **gUComp** and **gVComp** determine the number of facets in the circumferential and vertical directions respectively. In the above figures, the values of 18 and 3 are used.

Spheres

A simple faceted sphere may be drawn using the following routine:

gDrawSphere(xc, yc, zc, radius, [gURot, gVRot, gWRot, gUComp, gVComp])

Where **xc**, **yc** and **zc** specify the centre position of the sphere as shown below.



Sphere Primitive

The optional angular rotations **gURot**, **gVRot**, **gWRot**, do not affect the visual appearance of the sphere, but may be used to control the alignment of a texture map that is added to the surface of the sphere.

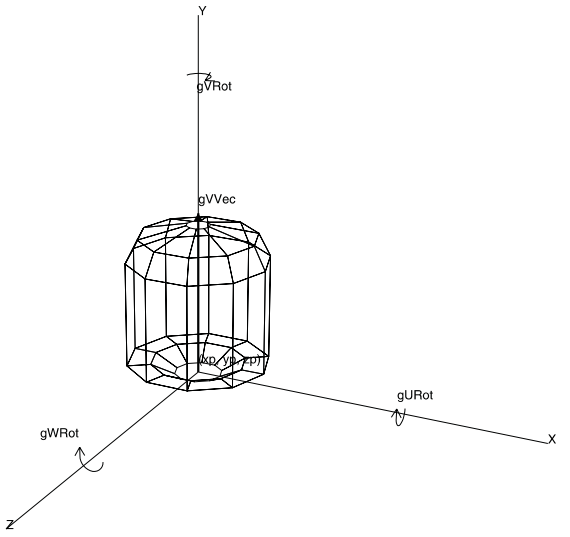
The optional arguments **gUComp** and **gVComp** determine the number of facets in the horizontal and vertical directions respectively controlling the smoothness of the sphere.

Volumes of Rotation

Volumes of rotation can be constructed using the following routine:

gDrawVolume(xp, yp, zp, npts, points, [gVVec, gAbs, gURot, gVRot, gWRot, gUComp])

where **xp**, **yp**, **zp** is the bottom centre of rotation of a set of points in the X-Y plane.



Volume of Rotation

The axis of rotation by default extends from the base centre vertically upwards in the Y direction, but this may be altered by specifying an absolute or relative vector (using **gVVec** and **gAbs**) or local axes of rotation (using **gURot**, **gVRot** and/or **gWRot**).

Surface Primitives

Two basic forms of surface primitive are provided for in the GINO library, the Spline surface and the Bezier surface. The basic difference between the two forms is the interpretation of the supplied data points. In the same way as the corresponding curve types, Spline surfaces pass through all the supplied data points, whereas the Bezier surface only aligns with the supplied data at the start and end of the surface. The supplied data can therefore be seen as control points affecting the shape of the surface only.

Both forms of surface are constructed using one or more sets of user supplied 3D data/control points from which are interpolated a smooth set of points that represent the interior/edges of the surface.

In other words, the complexity of the surface is not determined by the number of points supplied in the data arrays, but the values set in the optional arguments **gUComp** and **gVComp**.

The following sections describe the drawing of each of the surface primitives.

Spline surface

The basic Spline surface is constructed from a grid of 3D data points, with an optional complexity specification of the final surface:

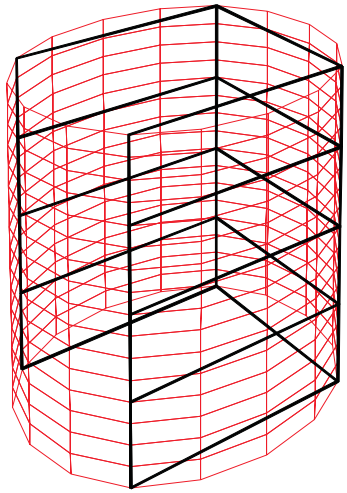
gDrawSplineSurface(nx, ny, mesh, [gUComp, gVComp])

Where **nx** and **ny** are the dimensions of the 2 dimensional array of data points of type GPOINT3 passed in **mesh**.

The setting of **gUComp** and **gVComp** define the complexity of the surface in terms of the number of facets generated across the complete surface in each direction. The values of **gUComp** and **gVComp** may however be rounded down internally as the actual number of facets must be a multiple of the number of data points so that the surface passes through each of them as supplied. The default setting for these arguments is 4*(**nx-1**) and 4*(**ny-1**) respectively

No end conditions can be set for spline surfaces but special attention is given to data that is closed in the 'U' direction such that a smooth join is automatically maintained at the junction. The routine gSetSplineTension() can also be used to control the tension of the spline surface in the same manner as for curves (see page 100).

A closed spline surface generated from 25 data points is shown in the figure below with the code following:



Spline Surface

C Code

```
#include <math.h>
#include <gino-c.h>

#define NX 5
#define NY 5
#define CLOSED 1

int main ()
{
    GDIM paper;
    GLIMIT3 viewport = {-100.0,100.0,-100.0,200.0,0.0,5000.0};
    GLIMIT limits = {0.0,0.0,0.0,0.0};

/*  Define control points */
    GPOINT3 pxyz[NX][NY];

    int i,j,ip;
    float s,x,y,z;
    float spten=0.0;

/*  Initialise Gino and Device */
    gOpenGino();
    xxxxx();

/*  Set up 3D Viewport to fit whole drawing area */
    gEnqDrawingLimits(&paper,&ip);
    limits.xmax = paper.xpap;
    limits.ymax = paper.ypap;
    gSetViewport3D(&viewport,&limits);
```

```
        /*  Define View */
            gDefinePerspView(100.0,200.0,800.0,-0.1,-0.2,-0.8,1200.0);
            gUpdateView();

        /*  Compute surface grid */
        /*  Define the elliptic cylinder - to show the difference a closed
        and */
        /*  open surface choose exact or approximate values of Pi below */
            for (i=0; i<NX; i++) {
                if (CLOSED)
                   s = 2.0*3.1415926*i/(NX-1);
                else
                   s = 2.0*3.14*i/(NY-1);

                x = sin(s);
                z = 2.0*cos(s);
                for(j=0; j<NY; j++) {
                   y = -1.0 + 2.0*j/(NY-1);
                   pxyz[i][j].x = x*50;
                   pxyz[i][j].y = y*50;
                   pxyz[i][j].z = (z+1.0)*50;
                }
            }

        /*  Set Hollow Fill Style */
            gSetFacetFillStyle(GHOLLOW);

        /*  Compute spline surface after setting the spline tension */
            gSetSplineTension(spten);
            gSetLineColour(GRED);
            gDrawSplineSurface(NX,NY,(GPOINT3 *)pxyz);

        /*  Draw control points mesh */
            gSetLineColour(GBLACK);
            gSetLineWidth(1.0);
            for(i=0; i<NX; i++) {
              gMoveTo3D(pxyz[i][0].x,pxyz[i][0].y,pxyz[i][0].z);
              gDrawPolylineTo3D(NY,pxyz[i]);
            }
            for(i=0; i<NY; i++) {
              gMoveTo3D(pxyz[0][i].x,pxyz[0][i].y,pxyz[0][i].z);
              for(j=1; j<NX-1; j++)
                gDrawLineTo3D(pxyz[j][i].x,pxyz[j][i].y,pxyz[j][i].z);
            }
            gFlushGraphics ();

        /*  Close down */
            gSuspendDevice();
            gCloseGino();
        }
```

F90 Code

```
        use gino_f90
        parameter (nx=5,ny=5)
        type (GDIM) :: paper
        type (GLIMIT3) :: viewport =
        GLIMIT3(-100.0,100.0,-100.0,200.0,0.0,5000.0)
        type (GLIMIT) :: limits = GLIMIT(0.0,0.0,0.0,0.0)
        !  Define control points
        type (GPOINT3) :: pxyz(nx,ny)
```

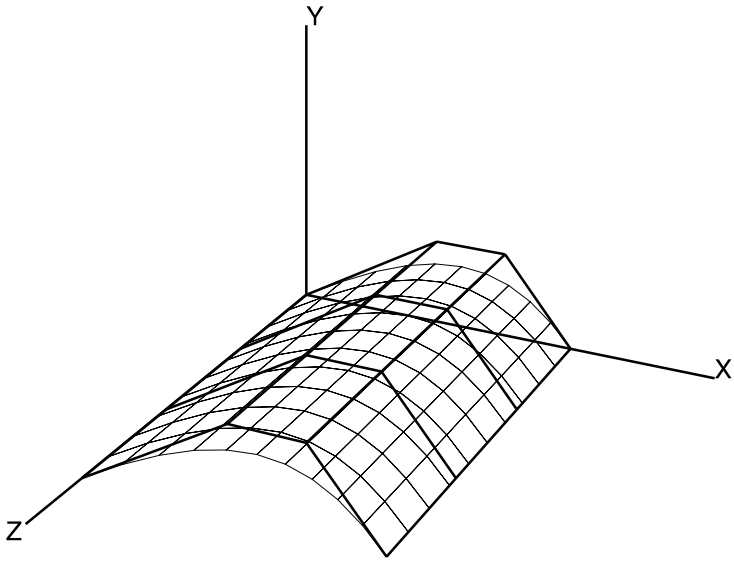
```
      !
      !  Initialise Gino and Device
            call gOpenGino
            call gWogl
      !
      !  Set up 3D Viewport to fit whole drawing area
            call gEnqDrawingLimits(paper,ip)
            limits%xmax = paper%xpap
            limits%ymax = paper%ypap
            call qSetViewport3D(viewport,limits)
      !
      !  Define View
            call gDefinePerspView(100.0,200.0,800.0,-0.1,-0.2,-0.8,1200.0)
            call qUpdateView
      !
      !  Compute surface grid
      !  Define the elliptic cylinder - to show the difference a closed
      and
      !  open surface choose exact or approximate values of Pi below
            iclose=1
            do i = 1, nx
                if (iclose .eq. 1) then
                   s = 2.0*3.1415926*(i-1)/(nx-1)
                else
                   s = 2.0*3.14*(i-1)/(nx-1)
                end if
                x = sin(s)
                z = 2.0*cos(s)
                do j = 1, ny
                   y = -1.0 + 2.0*(j-1)/(ny-1)
                   pxyz(i,j)%x = x*50
                   pxyz(i,j)%y = y*50
                   pxyz(i,j)%z = (z+1.0)*50
                end do
            end do
      !
      !  Set Hollow Fill Style
            call qSetFacetFillStyle(GHOLLOW)
      !
      !  Compute spline surface after setting the spline tension
            call gSetSplineTension(spten)
            call gSetLineColour(GRED)
            call gDrawSplineSurface(nx,ny,pxyz)
      !
      !  Draw control points mesh
            call gSetLineColour(GBLACK)
            call gSetLineWidth(1.0)
            do i = 1, ny
               call gMoveTo3D(pxyz(1,i)%x,pxyz(1,i)%y,pxyz(1,i)%z)
               call gDrawPolylineTo3D(nx-2,pxyz(2,i))
            end do
            do i = 1, nx
               call gMoveTo3D(pxyz(i,1)%x,pxyz(i,1)%y,pxyz(i,1)%z)
               do j = 2, ny
                  call gDrawLineTo3D(pxyz(i,j)%x,pxyz(i,j)%y,pxyz(i,j)%z)
               end do
            end do
            call gFlushGraphics
      !
      !  Close down
            call gSuspendDevice
            call gCloseGino
            end
```

Bezier surface

The basic Bezier surface is constructed from a grid of 3D control points, with an optional complexity specification of the final surface:

gDrawBezierSurface(nx, ny, mesh, [gUComp, gVComp])

Where **nx** and **ny** are the dimensions of the 2 dimensional array of control points of type GPOINT3 passed in **mesh**.



Bezier Surface

Note that whilst only 16 control points are passed to the gDrawBezierSurface() routine, the surface consists of 100 (10x10) facets due to the default settings of **gUComp** and **gVComp**. This example is produced using the following code:

C Code

```
#include <gino-c.h>
#define NX 4
#define NY 4
GDIM paper;
GLIMIT3 viewport = {-200.0,200.0,-200.0,200.0,0.0,1000.0};
GLIMIT  limits   = {0.0,0.0,0.0,0.0};
```

```
/* Define control points */
GPOINT3  pxyz[NX][NY] = {
       0.0, 0.0,  0.0,  0.0, 0.0, 50.0,
       0.0, 0.0,100.0,  0.0, 0.0,150.0,

      50.0,30.0,  0.0, 50.0,30.0, 50.0,
      50.0,30.0,100.0, 50.0,30.0,150.0,

      75.0,30.0,  0.0, 75.0,30.0, 50.0,
      75.0,30.0,100.0, 75.0,30.0,150.0,

     100.0, 0.0,  0.0,100.0, 0.0, 50.0,
     100.0, 0.0,100.0,100.0, 0.0,150.0};

main()
{
   int ip,i,j;

/* Initialize GINO and device */
      gOpenGino();
      xxxxx();

/* Set up 3D Viewport to fit whole drawing area */
      gEnqDrawingLimits(&paper,&ip);
      limits.xmax=paper.xpap;
      limits.ymax=paper.ypap;
      qSetViewport3D(&viewport,&limits);

/* Define Lighting and shading */
      gSetShadingMode(GFLAT,0);
      gSetLightSwitch(1,GON);

/* Define view */
      gDefinePerspView(300.0,300.0,600.0,-0.3,-0.3,-0.6,800.0);
      qUpdateView();

/* Set hollow fill style */
      gSetFacetFillStyle(GHOLLOW);

/* Set material and draw surface */
      gSetMaterialIndex(GDEFAULT,GDEFAULT);
      gSetLineColour(GBLACK);
      gDrawBezierSurface(NX,NY,&pxyz[0][0]);

/* Draw control points mesh */
      gSetLineWidth(1.0);
      for(i=0; i<NX; i++) {
        gMoveTo3D(pxyz[i][0].x,pxyz[i][0].y,pxyz[i][0].z);
        gDrawPolylineTo3D(NY,pxyz[i]);
      }
      for(i=0; i<NY; i++) {
        gMoveTo3D(pxyz[0][i].x,pxyz[0][i].y,pxyz[0][i].z);
        for(j=1; j<NX; j++)
          gDrawLineTo3D(pxyz[j][i].x,pxyz[j][i].y,pxyz[j][i].z);
      }

/* Close down */
      gSuspendDevice();
      gCloseGino();
}
```

F90 Code

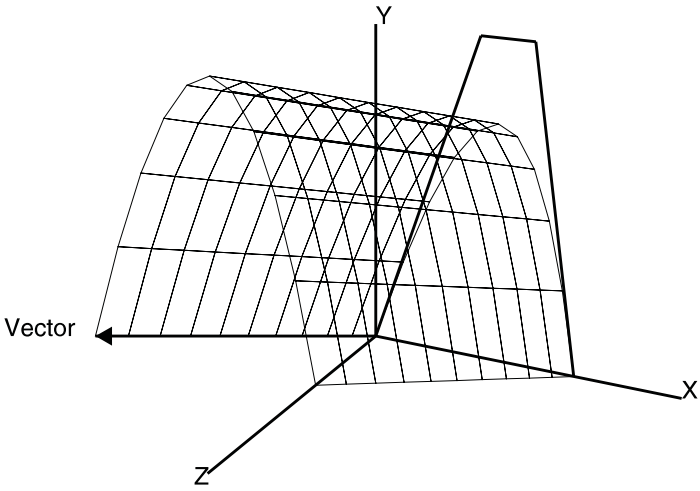
```
      use gino_f90
      parameter (nx=4,ny=4)
      type (GDIM) :: paper
      type (GLIMIT3) :: viewport = G
      LIMIT3(-200.0,200.0,-200.0,200.0,0.0,1000.0)
      type (GLIMIT) :: limits   = GLIMIT(0.0,0.0,0.0,0.0)
      ! Define control points
      type (GPOINT3) :: pxyz(nx,ny) = reshape( (/ &
            GPOINT3( 0.0, 0.0, 0.0), GPOINT3( 50.0,30.0, 0.0), &
            GPOINT3(75.0,30.0, 0.0), GPOINT3(100.0, 0.0, 0.0), &

            GPOINT3( 0.0, 0.0,50.0), GPOINT3( 50.0,30.0,50.0), &
            GPOINT3(75.0,30.0,50.0), GPOINT3(100.0, 0.0,50.0), &

            GPOINT3( 0.0, 0.0,100.0), GPOINT3( 50.0,30.0,100.0), &
            GPOINT3(75.0,30.0,100.0), GPOINT3(100.0, 0.0,100.0), &

            GPOINT3( 0.0, 0.0,150.0), GPOINT3( 50.0,30.0,150.0), &
            GPOINT3(75.0,30.0,150.0), GPOINT3(100.0, 0.0,150.0) /), &
            (/nx,ny/) )
      !
      ! Initialize GINO and device
            call gOpenGino
            call xxxxx
      !
      ! Set up 3D Viewport to fit whole drawing area
            call gEnqDrawingLimits(paper,ip)
            limits%xmax=paper%xpap
            limits%ymax=paper%ypap
            call gSetViewport3D(viewport,limits)
      !
      ! Define Lighting and shading
            call gSetShadingMode(GFLAT)
            call gSetLightSwitch(1,GON)
      !
      ! Define view
            call gDefinePerspView(300.0,300.0,600.0,-0.3,-0.3,-0.6,800.0)
            call qUpdateView
      !
      ! Set hollow fill style
            call gSetFacetFillStyle(GHOLLOW)
      !
      ! Set material and draw surface
            call gSetMaterialIndex(GDEFAULT,GDEFAULT)
            call gSetLineColour(GBLACK)
            call qDrawBezierSurface(nx,ny,pxyz)
      !
      ! Draw control points mesh
            call gSetLineWidth(1.0)
            do i = 1, ny
               call gMoveTo3D(pxyz(1,i)%x,pxyz(1,i)%y,pxyz(1,i)%z)
               call gDrawPolylineTo3D(nx-1,pxyz(2,i))
            end do
            do i = 1, nx
               call gMoveTo3D(pxyz(i,1)%x,pxyz(i,1)%y,pxyz(i,1)%z)
               do j = 2, ny
                  call gDrawLineTo3D(pxyz(i,j)%x,pxyz(i,j)%y,pxyz(i,j)%z)
               end do
            end do
```

```
!
! Close down
      call gSuspendDevice
      call gCloseGino
      stop
      end
```

Tabulated Bezier surface

The tabulated Bezier surface is generated from a set of 3D control points and a vector. An intermediate Bezier curve is computed from the control points and the surface is constructed by extending the curve along the specified vector.

gDrawTabulatedBezierSurface(np, points, vector, [gUComp, gVComp])

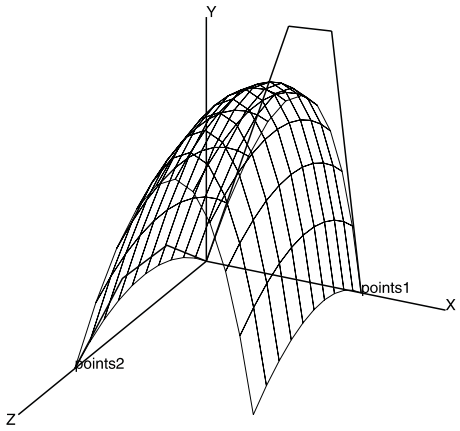


Tabulated Surface

Swept Bezier surface

A swept Bezier surface is similar to the tabulated surface except that the surface is constructed along the curve computed from a second set of control points:

gDrawSweptBezierSurface(np1, points1, np2, points2, [gUComp, gVComp])

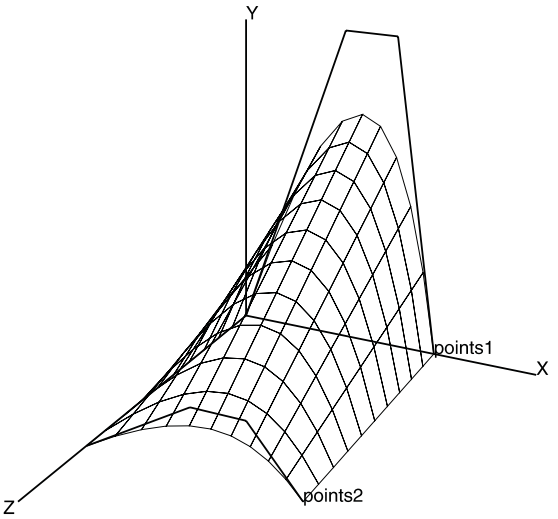


Swept Surface

Ruled Bezier surface

The ruled Bezier surface is also generated from two sets of 3D control points except that the surface is generated by constructing a grid of patches between each computed curve. The number of points in each set of control points need not be the same.

gDrawRuledBezierSurface(np1, points1, np2, points2, [gUComp, gVComp])

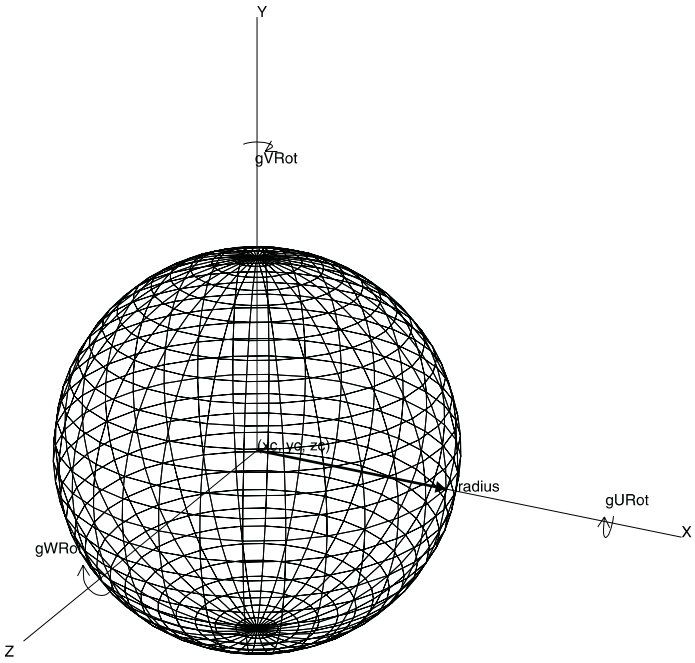


Ruled Surface

Bezier sphere

A Bezier sphere is internally constructed by rotating a semi-circular Bezier curve around a central axis. The user needs only to specify the centre, radius and optionally its complexity and local orientation.

gDrawBezierSphere(xc, yc, zc, radius, [gUComp, gVComp, gURot, gVRot, gWRot])

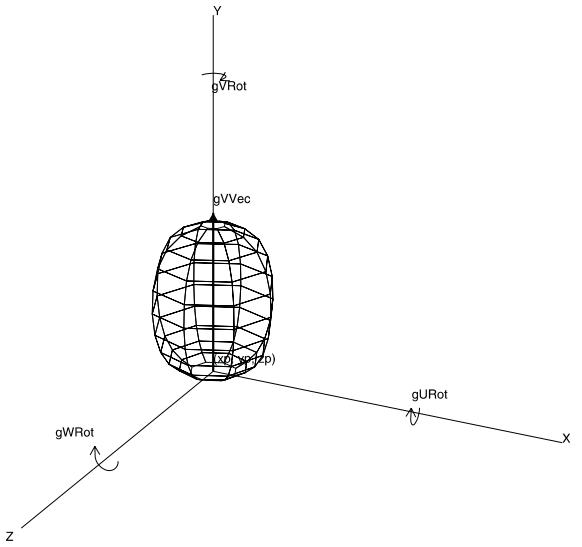


Bezier Sphere

Bezier volume

A Bezier volume of rotation is constructed from a user supplied set of 2D control points rotated round a central axis. The control points are used to generate a smooth Bezier curve which is rotated about a vector (0.0,1.0,0.0) starting at the position specified in xp,yp,zp. The optional arguments can be used to alter the complexity and local orientation.

gDrawBezierVolume(xp, yp, zp ,npts, points, [gURot, gVRot, gWRot, gUComp, gVcomp])



Bezier Volume

Chapter 18

LIGHTING AND SHADING

Lighting and Shading Introduction

The routine that sets up the lighting and shading environment is:

gSetShadingMode(mode, [gCulling, gBlending, gWinding])

which also sets up other lighting parameters as described below.

Shading

By default shading is switched off, under which circumstances objects are displayed in their defined colour, ignoring any lights and not sorted according to their distance from the viewer. Therefore new objects will hide objects already on the display if they occupy the same space.

When shading is switched on, objects further away from the viewer are automatically hidden from those closer. This is also known as depth buffering (see below) in which data is only placed on the output device if it represents an object which is closer to the view point than the data already displayed at that point. At the same time, the correct colour of the object is calculated using the current lighting and the objects material properties.

This shading can be performed in a number of different ways, requiring different amount of processing, and giving different levels of realism.

- Flat shading (**mode** = GFLAT) uses a single facet normal to calculate the lighting values of each facet.

- Smooth or Gouraud shading (**mode** = GGOURAUD) uses normals at each vertex if available and can give much smoother surfaces as long as the correct data is supplied with the facets.

- Phong shading (**mode** = GPHONG) gives the best results as this method interpolates normals at each pixel of each facet and is only available on certain systems.

To take full advantage of lighting and shading facilities, it is advisable to use the facet primitive, as these primitives contain all the necessary material attributes to calculate their correct appearance under different lighting conditions (see page 295).

Culling

One of the features of a facet is that it has a front and back face which allows different material properties to be defined for the inside and the outside of objects using a single 'skin'. However, it is obviously more expensive, in computing terms, to calculate the lighting values on both sides of every facet. It is possible therefore to save time by ignoring either all of the front or all of the back facing facets using the optional argument **gCulling** to the gSetShadingMode() routine.

This feature is useful if the scene is composed of solid objects and you are never interested in the inside or back of these objects.

Blending

Where any of the 3D objects in a scene are required to be transparent, using the appropriate material property, it is necessary to enable an additional shading function called blending. This adds another level of complexity to the lighting calculation as the colours of existing pixels need to be multiplied by the new objects colour to obtain the correct values. This feature is switched on using the optional argument **gBlending** to gSetShadingMode().

Winding Rule

By default, GINO interprets the coordinates of facets and polygons such that vertices defined in an anti-clockwise order are understood to be facing the viewer. Conversely, where vertices are defined as being in an anti-clockwise order as seen from the current viewing point, the surface is interpreted as facing away from the viewer. This is known as the facet winding and can be reversed using the optional argument **gWinding** argument where an application requires its vertices to be interpreted in the opposite way.

Shading Enquiry

The current lighting and shading settings may be enquired using the routine:

gEnqShadingMode(att)

where **att** is a structure of type GSHADING which contains the four aspects of the lighting and shading environment.

Depth Buffering

Once the depth buffer has been enabled using one of the shading modes described above, the decision to display an object is based on two things. 1) The distance of the object in relation to other objects already on the screen or window and 2) the logical depth test to be applied. By default these are calculated as follows:

The distance of an object from the viewer is mapped on to a value between 0.0 and 1.0 which represents the total range of the viewing volume as set up by gSetViewport3D(). Therefore objects at the back of the volume (i.e. furthest away from the viewer) are said to have a depth of 1.0 and objects closes to the viewer are said to have a depth of 0.0. When the screen or window is cleared the depth buffer is set to 1.0 for each pixel. Whenever a new object is displayed, its distance from the viewer is mapped onto the depth range (0.0 to 1.0) and for each pixel, if the depth is less than the depth in the depth buffer, the pixel is displayed and the buffer is updated. Otherwise if the depth is greater than or equal to the value in the depth buffer the pixel is not displayed (because it is deemed to behind an existing object).

In most systems, this depth range (0.0 to 1.0) is further mapped onto a 32bit or 16 bit integer to speed up the calculation and so there are in fact only 2**16 or 2**32 different depth values that can be tested against. This trade off of performance against accuracy enhances the importance of the correct setting of the viewing volume as set up by gSetViewport3D(), because too large a range will decrease the accuracy of the depth buffering algorithms. This can result in small objects which lie in front of existing objects not being displayed because their depth (as a 16/32bit integer) is calculated as being the same as the existing object.

It is possible to change these default settings using the following routine:

gSetDepthMode(mode,dinit)

where **mode** is the test that is applied to each pixel to be displayed, against the value in the depth buffer. The list of possible modes is given in the table below, together with its normal setting of the depth buffer initial value, **dinit**:

Mode	Initial Value
GNEVER	n/a
GLESSTHAN (default)	1.0
GLESSTHANOREQUALTO	1.0
GEQUALTO	As required
GNOTEQUALTO	As required
GGREATERTHANOREQUALTO	0.0
GGREATERTHAN	0.0
GALWAYS	n/a

As explained above, the default **mode** is GLESSTHAN and most of the alternative settings would only be required when special effects are required. Setting the mode to GLESSTHANOREQUAL can be useful for adding detail onto the surface of an existing object as this sets the logical test so that pixels less than or equal to the existing objects are still displayed. However this technique depends on the accuracy of the Z range set in the current 3D viewport (see also gSetFacetOffsetMode() for an alternative method).

Note that the depth buffer is only initialized to the value set in **dinit** when the screen is next cleared (gNewDrawing()) and **mode** is activated when gSetShadingMode() is next called.

The current depth buffer mode settings can be enquired using the following routine:

gEnqDepthMode(mode,dinit)

where the arguments return the values last set by gSetDepthMode() or the default ones.

Lighting

Light exists in three forms, ambient, diffuse and specular, and objects will absorb or reflect different amounts of these different kinds of light giving a different visual appearance.

Ambient light is background light that doesn't have a particular source and is shining in all directions.

Diffuse light has a particular source and is reflected off a surface evenly. A surface will be brighter if it lies perpendicular to the direction of the light source.

Specular light is also directional, but it is reflected sharply in a particular direction. A surface will be brighter if the angle of the surface reflects the light source towards the viewer.

Thus, in a simple example, a blue object is one which reflects blue ambient and diffuse light, and is shiny if it reflects specular light.

In the real world, things are a little more complex, with objects reflecting some kinds of light and absorbing others, letting some colours pass through all together (transparency) or even emitting light. All of these features will be dealt with later when an object's material properties are described (see page 339).

Light Sources

GINO provides a single routine that is used to define up to 8 independent light sources of 4 different types:

gDefineLightSource(light, colour, ...)

where light is the **light** number (1-8) and **colour** is the colour of the light source (which may be defined in terms of a colour index number or a 24bit RGB triplet (see page 205)). The setting of additional optional arguments to this routine define the different types of light source.

Ambient light

With no extra arguments, the type of light defined is an ambient light, ie. one that has no specific source or direction.

For example:

```
amb=gTrueCol(0.3,0.3,0.3);
gDefineLightSource(1,amb,0);
```

```
amb=gTrueCol(0.3,0.3,0.3)
call gDefineLightSource(1,amb)
```

defines light number one to be an ambient light with 30% white light.

Directional Light

Adding a direction vector using the optional argument **gDir**, specifies a directional light source. This is one where the source of light is said to be an infinite distance away in the direction of the vector specified. The rays of light are parallel to each other travelling from the source along the vector (in the opposite direction to the vector itself). A directional light source will only illuminate the side of objects in some way facing the light source, which in any one scene will always be the same side.

For example:

```
GPOINT3 vector = {100.0,0.0,0.0);

gDefineLightSource(1,5,
      gDir,vector,0);
```

```
type (GPOINT3) :: vector = &
      GPOINT3(100.0,0.0,0.0)
call gDefineLightSource(1,5, &
      gDir=vector)
```

specifies a green directional light source at an infinite distance away along the X axis which will illuminate the right hand side of all objects in a scene..

Point Light Source

Adding a 3D coordinate using the optional argument **gPos**, specifies a point light source at the specified position. Here the light is said to radiate out in all directions from the position of the light source and will therefore illuminate all objects with sides facing this position.

For example:

```
GPOINT3 pos = {10.0,10.0,-10.0);

gDefineLightSource(1,10,
      gPos,pos,0);
```

```
type (GPOINT3) :: pos = &
     GPOINT3(10.0,10.0,-10.0)
call gDefineLightSource(1,10, &
      gPos=pos)
```

specifies a white light located at (10.0,10.0,-10.0), which will illuminate all object surrounding this position.

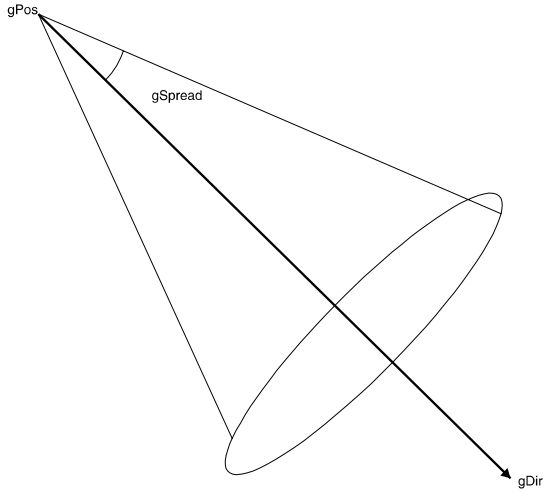
A point light may also have two additional attenuation factors **gAtten1** and **gAtten2**, which specify its constant and linear attenuation over distance from its source. These factors affect the light's strength according to the following formulae:

Attenuation factor = 1.0 / (**gAtten1** + distance * **gAtten2**)

Unless specified otherwise, **gAtten1** = 1.0 and **gAtten2** = 0.0 which implies full strength and no fade over distance.

Spot Light

A spot light is one which has a narrower focus than a point light, and so needs both the **gPos** (to specify its position) and **gDir** (to specify its direction) optional arguments. Two additional optional arguments specify the concentration and spread angle of the spot light.



Spot light definition

For example:

```
GPOINT3 pos = {0.0,0.0,10.0),
        dir = (0.0,0.0,-1.0);

gDefineLightSource(1,10,
     gPos,pos,gDir,dir,
     gConc,100.0,gSpread=40.0,0);
```

```
type (GPOINT3) :: &
   pos = GPOINT3(0.0,0.0,10.0),&
   dir = GPOINT3(0.0,0.0,-1.0)
call gDefineLightSource(1,10, &
      gPos=pos,gDir=dir, &
      gConc,100.0,gSpread,40.0)
```

specifies a point light source at (0.0,0.0,10.0) shining along the Z axis. The concentration is set at 100% which is at full strength and the spread angle of the light source is set at 40 degrees. This means that there is no light emanating from this source outside a notional cone with a 40% internal angle.

Specular Light Component

Under normal circumstances, ambient lights emit ambient light whilst directional, point and spot lights emit diffuse light. It is possible, however, to add a specular colour component to any light source using the optional argument **gSpec**. This may be necessary to define a light source which contains high levels of specular light, for instance in a spot light.

Light Switch

In order to light up a scene, it is not enough to simply specify the required light sources. They have to be switched on as well! The routine to control the state of each of the defined light sources is:

gSetLightSwitch(light,switch)

where **light** is the number and **switch** may be GON or GOFF. Note that, by default, all lights are switched off, so all objects will appear black until at least one light is switched on.

Default Lights

When GINO is initialised, two light sources are predefined. These may simply be switched on using gSetLightSwitch() or re-defined using gDefineLightSource(). The two lights are:

Light1: A white ambient light

Light2: A white directional light shining from 0.0,0.0,ZMAX

Light Source Enquiry

The complete set of attributes of any of the eight lights sources, including their state, may be enquired using the routine:

gEnqLightAttribs(light,attribs)

where **light** is the light number and **attribs** is a structure of type GLITATT.

Light Usage

An example using material and lighting is shown below:

C Code

```
#include <qino-c.h>

GDIM paper;
GLIMIT3  picture = {0.0,300.0,0.0,200.0,-1600.0,1600.0};
GLIMIT   viewport = {0.0,1.0,0.0,1.0};
GPOINT3  position = {0.0,500.0,100.0};
GPOINT3  direction = {0.0,-0.6,-1.0};
GMATATT  material2;
GPOINT3  table[4] = {400.0,-25.0,-200.0, -400.0,-25.0,-200.0,
                     -400.0,-25.0,2000.0,  400.0,-25.0,2000.0};

#if defined(MWIN) || defined(WOGL)
int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpszCmdParam, int nCmdShow)
#else
int main ()
#endif
{
   int ip;
   qOpenGino();
   qWoql(hInstance,hPrevInstance);

/* Set up viewport */

   qEnqDrawingLimits(&paper,&ip);
   picture.xmax=paper.xpap;
   picture.ymax=paper.ypap;
   viewport.xmax=paper.xpap;
   viewport.ymax=paper.ypap;
   qSetViewport3D(&picture,&viewport);

/* Define view */

   qDefineSphericalView(-50.0,50.0,100.0,350.0,0.5,-0.5,-1.0,200.0);
   qUpdateView();

/* Define lights */

   qSetShadingMode(GGOURAUD,qCulling,GBACK,0);
   gDefineLightSource(2,GWHITE,gPos,position,gDir,direction,
                        qConc,80.0,qSpread,100.0,0);
   qSetLightSwitch(1,GON);
   gSetLightSwitch(2,GON);
```

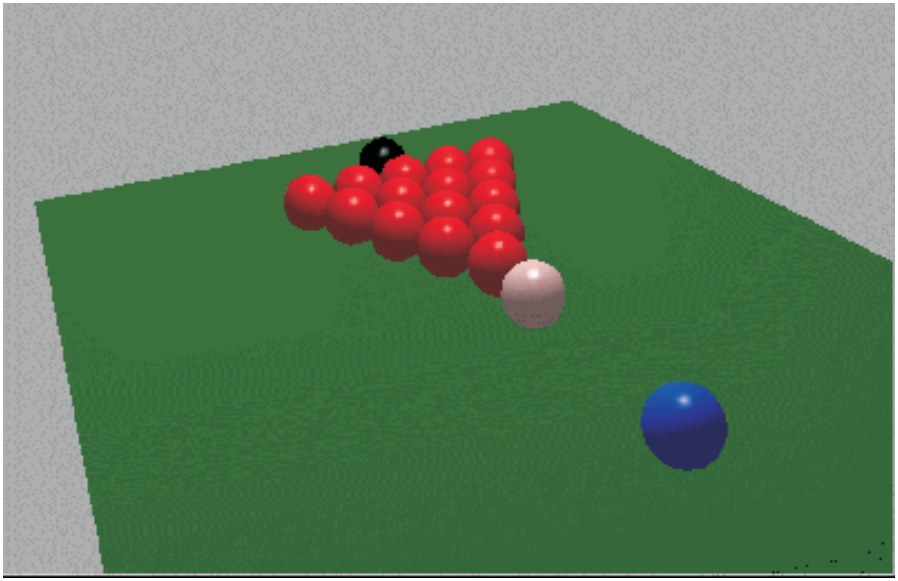
```
        /* Define material */

           material2.ambient=0.3;
           material2.diffuse=0.6;
           material2.specular=1.0;
           material2.shine=80.0;
           material2.trans=1.0;
           qDefineMaterial(2,&material2);

        /* Plot table */

           qSetMaterialIndex(1,1);
           qSetMaterialColour(qTrueCol(0.0,0.5,0.0),0);
           qDrawFacet(4,table,0);

        /* Plot reds */

           qSetMaterialIndex(2,0);
           qSetMaterialColour(GRED,0);
           qDrawSphere(0.0,0.0,0.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(-60.0,0.0,0.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(-120.0,0.0,0.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(60.0,0.0,0.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(120.0,0.0,0.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(-90.0,0.0,52.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(-30.0,0.0,52.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(30.0,0.0,52.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(90.0,0.0,52.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(-60.0,0.0,104.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(60.0,0.0,104.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(0.0,0.0,104.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(-30.0,0.0,156.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(30.0,0.0,156.0,30.0,qUComp,25,qVComp,25,0);
           qDrawSphere(0.0,0.0,208.0,30.0,qUComp,25,qVComp,25,0);

        /* Plot pink */

           qSetMaterialColour(qTrueCol(1.0,0.8,0.8),0);
           qDrawSphere(0.0,0.0,268.0,30.0,qUComp,25,qVComp,25,0);

        /* Plot black */

           qSetMaterialColour(GBLACK,0);
           qDrawSphere(0.0,0.0,-60.0,30.0,qUComp,25,qVComp,25,0);

        /* Plot blue */

           qSetMaterialColour(GBLUE,0);
           qDrawSphere(0.0,0.0,460.0,30.0,qUComp,25,qVComp,25,0);

        /* Close down */

           qSuspendDevice();
           qCloseGino();
        }
```

F90 Code

```
        Program snooker
        use gino f90
        type (GDIM) paper
```

```
      type (GLIMIT3) :: picture =
      GLIMIT3(0.0,300.0,0.0,200.0,-1600.0,1600.0)
      type (GLIMIT) :: viewport = GLIMIT(0.0,1.0,0.0,1.0)
      type (GPOINT3) :: position = GPOINT3(0.0,500.0,100.0)
      type (GPOINT3) :: direction = GPOINT3(0.0,-0.6,-1.0)
      type (GMATATT) :: material2
      type (GPOINT3) :: table(4) = (/ &
        GPOINT3(400.0,-25.0,-200.0), GPOINT3(-400.0,-25.0,-200.0), &
        GPOINT3(-400.0,-25.0,2000.0), GPOINT3(400.0,-25.0,2000.0) /)
      !
         call qOpenGino
         call xxxxx
      !
      ! Set up viewport
         call qEnqDrawingLimits(paper,ip)
         picture%xmax=paper%xpap
         picture%ymax=paper%ypap
         viewport%xmax=paper%xpap
         viewport%ymax=paper%ypap
         call qSetViewport3D(picture,viewport)
      !
      ! Define view
         call gDefineSphericalView(-50.0,50.0,100.0,350.0, &
                                   0.5,-0.5,-1.0,200.0)
         call qUpdateView
      !
      ! Define lights
         call gSetShadingMode(GGOURAUD,gCulling=GBACK)
         call gDefineLightSource(2,GWHITE,gPos=position,gDir=direction, &
                                   qConc=80.0,qSpread=100.0)
         call qSetLightSwitch(1,GON)
         call qSetLightSwitch(2,GON)
      !
      ! Define material
         material2%ambient=0.3
         material2%diffuse=0.6
         material2%specular=1.0
         material2%shine=80.0
         material2%trans=1.0
         call qDefineMaterial(2,material2)
      !
      ! Plot table
      !
         call qSetMaterialIndex(1,1)
         call qSetMaterialColour(qTrueCol(0.0,0.5,0.0),0)
         call qDrawFacet(4,table)
      !
      ! Plot reds
      !
         call qSetMaterialIndex(2,0)
         call qSetMaterialColour(GRED,0)
         call qDrawSphere(0.0,0.0,0.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(-60.0,0.0,0.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(-120.0,0.0,0.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(60.0,0.0,0.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(120.0,0.0,0.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(-90.0,0.0,52.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(-30.0,0.0,52.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(30.0,0.0,52.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(90.0,0.0,52.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(-60.0,0.0,104.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(60.0,0.0,104.0,30.0,qUComp=25,qVComp=25)
         call qDrawSphere(0.0,0.0,104.0,30.0,qUComp=25,qVComp=25)
         call gDrawSphere(-30.0,0.0,156.0,30.0,gUCcomp=25,gVComp=25)
```

```
            call qDrawSphere(30.0,0.0,156.0,30.0,qUComp=25,qVComp=25)
            call qDrawSphere(0.0,0.0,208.0,30.0,qUComp=25,qVComp=25)
         !
         ! Plot pink
         !
            call qSetMaterialColour(qTrueCol(1.0,0.8,0.8),0)
            call qDrawSphere(0.0,0.0,268.0,30.0,qUComp=25,qVComp=25)
         !
         ! Plot black
         !
            call qSetMaterialColour(GBLACK,0)
            call qDrawSphere(0.0,0.0,-60.0,30.0,qUComp=25,qVComp=25)
         !
         ! Plot blue
         !
            call qSetMaterialColour(GBLUE,0)
            call qDrawSphere(0.0,0.0,460.0,30.0,qUComp=25,qVComp=25)
         !
         ! Close down
            call qSuspendDevice
            call qCloseGino
            stop
            end
```



Snooker Balls

It should also be noted that, increasing the number of lights that are used (i.e. switched on), increases the time taken to calculate the correct colour of the objects being displayed.

Fog

Fog simulation is available on devices that provide 3D lighting and shading facilities and can be useful for added visual realism in a scene. This special effect mixes a user defined colour into the scene based on the distance from the viewer that each object is drawn. The same routine can either add the colour based linearly on the distance from the viewer (which is essentially depth-cueing), or more realistically on an exponential function of the distance.

The routine to define the required fog attributes is:

gDefineFog(mode, colour, [gStart,gEnd,gDensity])

where **mode** can be one of:

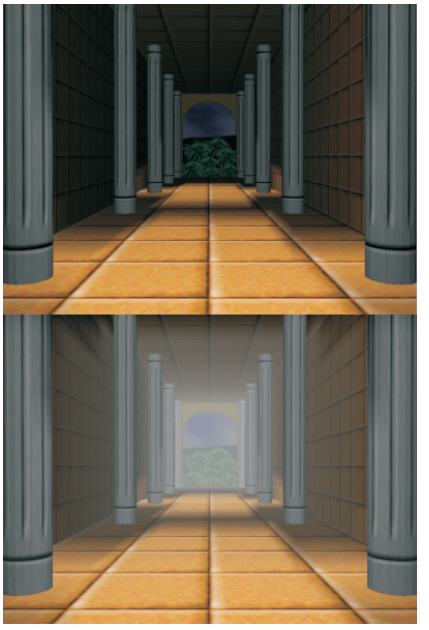
- GNONE - No fog simulation
- GLINEAR - Linear fade used for depth-cueing
- GEXP1 - Exponential fade used for simulating cloud or heavy fog
- GEXP2 - Exponential fade used for simulating smoke or weather haze

For most fog applications, the fog **colour** would be white, although any colour can be used, passed either as a colour index or a 24bit RGB triplet.

The optional arguments to the gDefineFog() routine specify the arguments to various fog modes.

The GLINEAR fog mode can be used for depth-cueing, where objects are obscured according to their distance from the viewer. In this mode the start and end distances (**gStart** and **gEnd**) specify the viewing range over which the fog should be applied. For example, in a perspective view drawn within a viewing volume with a depth of 1000.0, specifying start and end values of 500.0 and 1000.0 will cause the fog to start half way through the volume and be at full density (i.e. the fog colour) at the back of the viewing volume.

For GEXP1 and GEXP2 fog modes, the **gDensity** setting controls the exponential calculations. Density values of around 0.0025 give realistic effects.



Adding Fog to a Scene

Fog Enquiry

The current fog settings can be enquired using the routine:

gEnqFog(attribs)

where **attribs** is a structure of type GFOGATT containing values that are the default settings or those set by the last call to gDefineFog().

Chapter 19

MATERIAL PROPERTIES

Material Properties Introduction

The facet primitive and all objects built up of facets (cubes, cylinder, bezier surfaces etc.) may use the current drawing colour to define their colour, but in order to fully describe their appearance under different lighting conditions, a facet has a material property.

In its full form, a material property describes the facets reaction to the different types of light described in the previous chapter, namely ambient, diffuse and specular. In addition to this, a material may be defined as having a certain level of translucence (i.e. allowing light to pass through it), or even emit light itself. Due to the complexity of specifying material properties, GINO provides a number of levels of definition so that lighting effects can be quickly introduced as well as providing sophisticated facilities for photo realistic effects.

It should be remembered that unlike other primitives, a facet has two surfaces, a front and back and different material properties can be assigned to each.

Material Property Definition

Three different levels of material definition are provided by GINO, each with increasing sophistication and complexity. These are described in the following sections.

Colour Matching

The simplest way of defining the colour of facets (and objects) is to use the current line colour (gSetLineColour()) to automatically set both the ambient and diffuse material properties of the surface. This is in fact the default state of material definition and so no special calls are needed if this method is required.

Under these circumstances, facets (and objects) appear in a shade of the specified colour according to the angle of the light(s) shining on the surface. If no lights are switched on, the facets will appear in the colour selected without shading.

To return to this mode from a different material mode, the following use of the routine gSetMaterialIndex() is required:

```
gSetMaterialIndex(GOFF,GOFF);
```

```
call gSetMaterialIndex(GOFF,GOFF)
```

Material Table

The second alternative is to use a material table to define a set of material properties that can be used for many facets or objects independently of their actual colour. Therefore if a scene has many objects of different colours but made of the same ‘substance’, a single table entry would be defined to represent that material. Examples include matt or shiny surfaces.

The material table stores coefficients of ambient, diffuse and specular light, which are multiplied by the object colour to give the actual material properties of the facets when drawn. The material table can contain up to 256 entries defining different material types. Each entry is set using the routine:

gDefineMaterial(mat, rep)

where **mat** is the material table index and **rep** is a structure of type GMATATT containing the material table attributes.

When GINO is initialized, three entries of the table are set to the following values:

mat	ambient	diffuse	specular	shine	trans	description
1	0.3	0.6	0.0	30.0	1.0	normal
2	0.3	0.6	1.0	30.0	1.0	plastic
3	0.3	0.6	1.0	100.0	1.0	shiny

The settings for any entry in the material table may be enquired through the routine:

gEnqMaterial(mat, rep)

Material Index and Colour

The current material coefficients are set using the following routine:

gSetMaterialIndex(fmat,bmat)

where **fmat** and **bmat** are the material table indices required for all the front and back faces of subsequently drawn facets.

The current facet material colour is then set using the routine:

gSetMaterialColour(fcol, bcol)

where **fcol** and **bcol** are the required colour setting for all the front and back faces. These may be indices into GINO's colour table or 24-bit true colour values returned through the function gTrueCol()(see page 205).

Example:

Two special cases of material index values to the routine gSetMaterialIndex() should be noted. If either **fmat** or **bmat** are negative, the current index setting for the front or back face (as appropriate) is not changed. If the back face material index, **bmat** is set to GOFF, then no lighting calculations are performed on the back facing surfaces. This operation is separate from the culling of back faces performed by gSetShadingMode(), but by applying the two together can dramatically improve performance on some graphics cards if back faces are not required in a scene.

The current material index and colour settings are returned using the following routine:

gEnqMaterialAttribs(fcol,bcol,fmat,bmat)

where **fcol** and **bcol** are the current material colours for the front and back faces and **fmat** and **bmat** are the current material table indices.

This method is sufficient for the majority of lighting scenes, but does not provide for objects that reflect different colours of ambient and diffuse light, or objects that emit light.

Facet Material Properties

The final method sets the current material property in terms of the actual colours required for all the possible attributes including emission. This is achieved using the routine:

gSetFacetMaterialProps(side, amb, diff, spec, emit, shine, trans)

where **side** is either GFRONT or GBACK, representing the facet face being defined. The arguments **amb**, **diff**, **spec** and **emit** are all integer colour values which may be indices into GINO's colour table or 24-bit true colour values returned through the function gTrueCol(). The materials shininess is set as a percentage using **shine** and its translucence, in the range 0.0 to 1.0, is set in **trans**.

This method overrides the use of the material table as the setting routine gSetFacetMaterialProps() sets the material properties directly.

The current facet material properties can be enquired using the routine:

gEnqFacetMaterialProps(side, amb, diff, spec, emit, shine, trans)

Translucence

One of the material properties of facets that can be defined in conjunction with its reflectivity and shininess, is its translucence. A solid object is said to be opaque (**trans**=1.0) if you cannot see any object through it and transparent (**trans**=0.0) if you can see right through it. In reality, glass and Perspex objects have varying levels of translucence which can be emulated using this property of the material being defined. However, it is important to note that translucence is only correctly calculated if blending is switched on in the gSetShadingMode() routine.

Shadows

Simple (planar) shadows can be generated with the aid of the two following routines:

gCreatePlanarShadowMatrix(plane,light,matrix)

gModifyView(matrix)

where **plane** is a set of three (GPOINT3) points representing the plane on which the shadow is to be drawn, **light** is the position of the light source and **matrix** is a 16 element (4x4) array containing a modification to the current view.

The steps required to generate a shadow of one or more objects on a planar surface are as follows:

1) Switch the depth mode to be GLESSTHANOREQUAL. This is required because the shadow needs to be drawn at the same depth as a surface object on which it lies and must not be omitted from the depth buffer.

2) Define a suitable (black) shadow material. GMATSTY(0.0,0.0,0.0,0.0,0.8) defines one which has no colour components and a little transparency. Remember to switch blending on if you want the transparency to be realised.

3) Set up the lighting conditions and draw the required surface and objects.

4) Set up the shadow matrix and modify the current view using the above routines.

5) Redraw the objects using the shadow material for each of them.

The source of GINO Example program 10 shows the generation of shadows whilst moving the light source and/or spinning the objects.

Chapter 20

TEXTURE MAPPING

Texture Mapping Introduction

Texture mapping is the ‘draping’ of images over objects and is used to give objects texture or to map complex pictures over a multi-faceted surface. The process is achieved by mapping the pixels of a pixel array onto the pixels of either a single facet (polygon) or a set of facets. More and more graphics cards now provide hardware facilities to do texture mapping, so the process can be relatively quick although on the older or more basic hardware the addition of textures will dramatically slow down an application.

Texture mapping is not the same as ray-tracing which is another technique to map images onto a surface. Ray tracing is however limited to mapping images onto a surface after it has been transformed onto a 2D plane where as texture mapping is done at the 3D level and is therefore much more useful to 3D scenes.

Texture Mapping Modes

Texture data can be ‘draped’ over an object in two different ways, either to replace or merge with other details of the surface. The required mode is selected using the routines:

gSetTextureMappingMode(mode, ...)

where **mode** can be one of the following:

- GOFF - switch texture mapping off (the default state).
- GOVERLAY - where the texture is placed directly on the surface with no modification.
- GMODULATE - where the texture is merged with the colour of the surface. This is used where you want to merge a texture with the defined colour (and lighting effects) of an object.
- GBLEND - where the texture is merged with the object's colour (as above) and a constant blend colour.

The remaining optional arguments to gSetTextureMappingMode() are described below (see page 354).

Texture Mapping Data

GINO uses the same pixel data that is used for the display of 2D images for texture mapping, i.e. an integer array containing colour index values or 24 bit packed RGB values (see page 205). These can be generated within an application or read into an appropriately sized integer array from a BMP or JPEG files using

gGetImageFile(type, file, coldef, offset, collim, xgrid, ygrid, npix, pixbuf)

This routine should be used in conjunction with gEnqImageFile() to check the image type and dimensions of the external file being read, as well as defining any colour mapping that is required (see page 73).

It should be noted however, that it is usually not possible to use any sized pixel array for texture mapping. In order for the process to be as efficient as possible, texture mapping software requires the texture map to be dimensioned in terms of a power of 2 (up to a total limit of the equivalent of 1024x1024 pixels). The image doesn't have to be square, but it has to be a power of 2 in both directions. e.g. 8x16, 64x64, 512x256 or 2048x512.

The texture map may contain an additional optional border of 1 pixel around each edge increasing the image size by 2 pixels in each direction. Thus a 64x64 pixel image becomes 66x66. The border is used for the correct tiling of large images (see page 349) or where a separate repeated colour is used when the image is smaller than the object it is being draped over (see page 354).

Once the data has been read into memory, the pixel array is assigned to be the current texture map using the following routine:

gDefineTexture(level, xgrid, ygrid, border, nbyte, pixbuf)

where **level** is the level of a possible multi-level texture map (see page 348), **xgrid** and **ygrid** define the dimensions of the pixel array with the data itself passed in the integer array **pixbuf**. The argument **border** is set to GON of GOFF depending on whether the image has an extra border of 1 pixel around each edge.

The value of **nbyte** should be set to the number of relevant bytes in the pixel data that are to be used for the texture map. The following table illustrates the possible settings:

nbyte	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
1	1			Luminance
2	1		Alpha	Luminance
3 (default)	1	Red	Green	Blue
4	Alpha OR 1	Red	Green	Blue

Note that byte values of 1 and 2 are only relevant to texture modes GMODULATE and GBLEND and byte values of 3 and 4 are only relevant to texture modes GMODULATE and GOVERLAY. Where colour indices or data returned from gGetImageFile() are used for texture maps, **nbyte** should be set to 3. Other values can only be used where the texture map data is manually constructed or special significance is to be interpreted into data read from alternative sources. In all these cases bit 24 of the texture map data should be set to 1.

The following code shows the steps required to read in a texture image and assign a texture mapping mode:

C Code

```
include <gino-c.h>
int *image;

   int type,xgrid,ygrid,nbpp,ncols,nn;
/* Initialize GINO and device */
   gOpenGino();
   xxxxx();

/* Enquire image file dimensions */
   qEnqImageFile(”ball.ico”,&type,&xgrid,&ygrid,&nbpp,&ncols);

/* Allocate memory for image file */
   nn = xgrid*ygrid;
   image = malloc ( nn * sizeof(int));
   if(image != NULL) {

/* Read in image file */
       gGetImageFile(type,”ball.ico”,1,0,0,
                     &xgrid,&ygrid,nn,image);
```

```
/* Assign image to texture map */
      gDefineTexture(0,xgrid,ygrid,GOFF,3,image);

/* Define Texture mapping mode */
      gSetTextureMappingMode(GOVERLAY,0);

   }
```

F90 Code

```
use gino_f90
integer, dimension(:), allocatable :: image

! Initialize GINO and device
   call gOpenGino
   call xxxxx

! Enquire image file dimensions
   call gEnqImageFile('ball.ico',itype,ixgrid,iygrid,nbpp,ncols)

! Allocate memory for image file
   nn = ixgrid*iygrid
   allocate(image(1:nn),stat=ier)
   if(ier.eq.0) then

! Read in image file
      call gGetImageFile(itype,'ball.ico',1,0,0, &
                         ixgrid,iygrid,nn,image)

! Assign image to texture map
      call gDefineTexture(0,ixgrid,iygrid,GOFF,3,image)

! Define Texture mapping mode
      call gSetTextureMappingMode(GOVERLAY)

   end if
```

Note that while a texture map is currently defined and either GOVERLAY, GMODULATE or GBLEND modes are current, texture mapping is applied to every object (apart from hardware text) that is subsequently drawn. Users should therefore switch off texture mapping when objects are required to be drawn without texture mapping applied.

Multiple Texture Maps

GINO provides a facility to define multiple texture map images for an object or objects, so that when the object is viewed from a distance, a less detailed texture map may be applied. This feature can give improved image quality and save display time at the expense of greater memory usage. These are known as *mipmapped* textures.

In addition to the 'primary' texture (**level** = 0) a series of secondary images need to be supplied (**level** = 1, 2, etc.) with each one a power of 2 smaller in each dimension. Thus if the primary image was 256x128 pixels, the second image would be 128x64 and the third 64x32 and so on down to a 1x1 image to complete all the necessary levels.

The mechanism used to determine which image is used when, is controlled by the routine gSetTextureMappingMode() described below.

Tiling Images

Where a texture map, larger than that supported by GINO is required, it is necessary to divide the object and texture image into sections so that the larger object can be dealt with. Taking a simple example of a large flat surface requiring a texture map of 2048x2048 pixels. It is necessary to divide both the surface and the texture map into four quarters and deal with each one in turn.

In order to correctly blend the four images together in 3D space, it is also advisable to add a border to each quarter image, representing the adjacent row or column from the adjoining image. Where the border represents the border of the complete image, the last row/column can be repeated in the border for consistency.

Texture Mapping Coordinates

Irrespective of the dimensions of the texture map data in terms of pixels, the currently assigned array is given an arbitrary range of 1 unit by 1 unit in what are known as' texture coordinates'. Thus the width of the texture map has a range of 0.0 to 1.0 in a locally horizontal direction (referred to as S), and the height of the texture map in pixels is mapped onto a range of 0.0 to 1.0 in a locally vertical direction (referred to as T). Note that the origin (0.0,0.0) of a pixel image is in the top left corner.

In order to map this texture data onto a surface or object, the surface or object is also assigned texture coordinates (in addition to its physical, 3D space coordinates and its surface normals (if these are used)). Under normal circumstances, these surface or object texture coordinates would also lie in the range 0.0 to 1.0, but this is not obligatory (see page 354). However when the image and the surface has texture coordinates lying in the range 0.0 to 1.0 there is obviously a simple 1:1 mapping.

An important consideration in the assignment of texture coordinates to any object, is the possible distortion of the texture map. Mapping a square texture onto an object twice as high as it is wide will obviously distort the image if the default range of 0.0 to 1.0 is assigned in both directions.

If an appropriately dimensioned image is not available, texture coordinates should be assigned to maintain the aspect ratio of the image that is available. This may mean the setting of texture coordinates that do not extend to the whole range of 0.0 to 1.0 thus omitting parts of the image, or setting of coordinates that go outside the range, with the decision as to whether the image is repeated or clamped (see page 354).

The allocation of texture coordinates for an object can be done in one of two ways.

- Direct assignment with object definition
- Automatic generation using a transformation of the objects' physical coordinates or some other arbitrary system

Direct Assignment

Texture coordinates can be assigned to facets when they are defined using the optional texture coordinate data, **gTextCoords**, in the routine gDrawFacet(). The data is supplied in an array of type GPOINT3, but only the X and Y coordinates are currently used for mapping the current texture map.

The following code (added to the previous example) defines a rectangle with texture coordinates in the range 0.0 to 2.0 vertically and 0.0 to 2.5 horizontally. This means that, by default, the image will be repeated twice vertically and two and a half times across the rectangle (see Repeating and Clamping Images below).

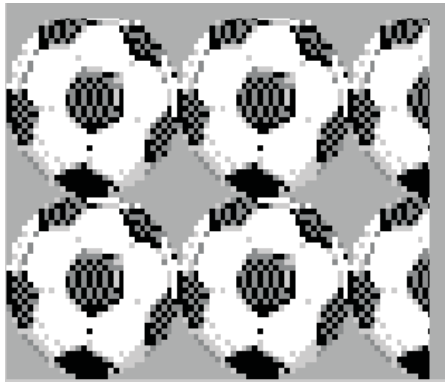
C Code

```
GPOINT3 rect[4] = { 0.0, 0.0,0.0, 75.0, 0.0,0.0,
                    75.0,60.0,0.0,  0.0,60.0,0.0};
GPOINT3 text[4] = { 0.0, 2.0,0.0,  2.5, 2.0,0.0,
                     2.5, 0.0,0.0,  0.0, 0.0,0.0};


/* Draw facet with texture coordinates */
      gDrawFacet(4,rect,gTextCoords,text,0);
```

F90 Code

```
type (GPOINT3) :: rect(4) = (/ &
      GPOINT3( 0.0, 0.0,0.0), GPOINT3(75.0, 0.0,0.0), &
      GPOINT3(75.0,60.0,0.0), GPOINT3( 0.0,60.0,0.0) /)
type (GPOINT3) :: text(4) = (/ &
      GPOINT3( 0.0, 2.0,0.0), GPOINT3( 2.5, 2.0,0.0), &
      GPOINT3( 2.5, 0.0,0.0), GPOINT3( 0.0, 0.0,0.0) /)


! Draw facet with texture coordinates
      call gDrawFacet(4,rect,gTextCoords=text)
```

Note that the vertices start at the bottom left corner, going anti-clockwise, whilst the origin of the texture coordinates are at the top left corner. This rectangle could be viewed from any angle with the texture map correctly adjusted to fit the skewed surface.



Texture Map

Automatic Generation

The alternative method is to automatically generate texture coordinates by transforming the objects physical coordinates. The current generation method is set up and enquired using the following routines:

gSetTextureCoordGeneration(mode,[gSVec, gTVec])

gEnqTextureCoordGeneration(mode,[gSVec, gTVec])

where **mode** is the coordinate generation mode and may be one of the following:

- GOFF - switch texture coordinate generation off (default)
- GOBJECT - use object coordinates
- GSPHERICAL - coordinates are generated in a sphere around the viewing position

The optional arguments **gSVec** and **gTVec** are structures of type GTEXVEC containing the required modes and multiplication factors to be applied to the objects coordinates when GOBJECT mode is used. Each argument is used to automatically calculate the horizontal (S) and vertical (T) texture coordinates respectively for every vertex of the object based on its transformed (GPICTURE) or untransformed (GSPACE) coordinates.

S = svec.xfactor * x + svec.yfactor * y + svec.zfactor * z + svec.wfactor * w

T = tvec.xfactor * x + tvec.yfactor * y + tvec.zfactor * z + tvec.wfactor * w

Therefore, to calculate texture coordinates based on a scale factor of 1/30 of the rectangles untransformed coordinates, the following code is used:

Replacing the above additions with the code below gives the same result.

C Code

```
GPOINT3 rect[4] = { 0.0, 0.0,0.0, 75.0, 0.0,0.0,
                    75.0,60.0,0.0,  0.0,60.0,0.0};
GTEXVEC svec = {GSPACE, 0.033, 0.0, 0.0, 0.0},
        tvec = {GSPACE, 0.0, -0.033, 0.0, 0.0} ;


/* Define automatic texture coordinate generation */
      gSetTextureCoordGeneration(GOBJECT,gSVec=&svec,gTVec=&tvec,0);

/* Draw facet*/
      gDrawFacet(4,rect,0);
```

F90 Code

```
type (GPOINT3) :: rect(4) = (/ &
      GPOINT3( 0.0, 0.0,0.0), GPOINT3(75.0, 0.0,0.0), &
      GPOINT3(75.0,60.0,0.0), GPOINT3( 0.0,60.0,0.0) /)
type (GTEXVEC) :: svec = GTEXVEC(GSPACE,0.033,0.0,0.0,0.0)
type (GTEXVEC) :: tvec = GTEXVEC(GSPACE,0.0,-0.033,0.0,0.0)


! Define automatic texture coordinate generation
      call gSetTextureCoordGeneration(GOBJECT,gSVec,svec,gTVec,tvec)

! Draw facet
      call gDrawFacet(4,rect)
```

There are several things to note from this example.

Firstly, the value 0.033 represents 1/30 which is the factor needed to scale the object coordinates so that the vertices map to whole integer values. Thus the bottom left vertex (at 0,0) has a texture coordinate of (0.0,0.0) and the top right vertex (at 75,60) has a texture coordinate of (2.5,-2.0). By default the image is repeated, so the image lies on the rectangle correctly.

Secondly, the value of -0.033 used to calculate the vertical (T) texture coordinate, is negative to ensure the image is the right way up. Remember that image coordinates start at the top left and Y +ve is downwards, so the texture coordinates at the bottom of the rectangle should be higher than the texture coordinates at the top of the rectangle.

A combination of transformed or untransformed generated coordinates may be used, by the appropriate setting of **gSVec** and/or **gTVec**. In addition, if either transformation vector is omitted, the corresponding coordinate is taken from those directly assigned using the **gTextCoords** argument to gDrawFacet().

Environment Mapping

The texture generation mode GSPHERICAL can be used to wrap the image of a scene over one of more objects to give the impression that they reflect the environment in which they lie. In other words, it can be used to display a series of objects which appear as is they were perfectly reflective of their surroundings.

In order to correctly use this mode, it is necessary to have an appropriate texture map which ideally should be one taken by a camera with an extremely wide angle (fish-eye) lens.

3D Objects

In order to ease the task of ‘draping’ textures over GINO’s standard 3D objects, texture coordinates in the range of 0.0 to 1.0 are automatically generated for each complete primitive if a texture mapping mode has been set prior to drawing the object. If a different set of texture coordinates is required, then these must be generated by calling gSetTextureCoordGeneration() before the object is drawn.

Texture Mapping Attributes

In most cases the default texture mapping attributes will be sufficient for the majority of users. However, some finer controls may be needed by some applications in which case the optional arguments to the routine gSetTextureMappingMode() can be used, the full description of which is:

gSetTextureMappingMode(mode, [gBlendCol, gWraps, gWrapt, gMaxfil, gMinfil, gBorderCol])

The attributes are described in the following sections:

Blending Textures

The optional argument **gBlendCol** is used in conjunction with GBLEND texture mapping mode to set a texture environment colour. This can be used for cloud textures where the constant colour would be off-white.

Repeating and Clamping Images

The arguments **gWraps** and **gWrapt** control the effects of supplying or calculating texture coordinates outside the range 0.0 to 1.0 in the horizontal (S) and vertical (T) directions respectively. By default, if the texture coordinate range exceeds these limits, then the image is automatically repeated over the object in both directions. By setting either **gWraps** and/or **gWraps** to GCLAMP the image is clamped to limits of the texture map. In this case the texture border colour (see below) is extended to the limits of the facet in the clamped direction.

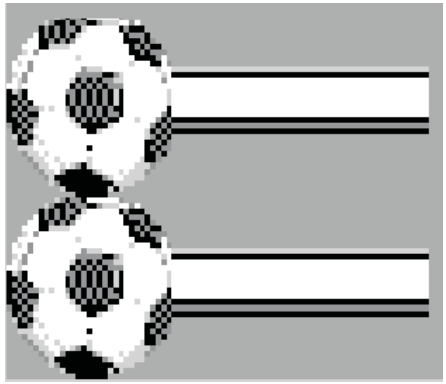
Modifying the call to gSetTextureMappingMode() in the above example shows the effect of clamping in the horizontal (S) direction:

C Code

```
gSetTextureMappingMode(GOVERLAY,gWraps,GCLAMP,0);
```

F90 Code

```
call gSetTextureMappingMode(GOVERLAY,gWraps=GCLAMP)
```



Clamping a Texture

Filtering Textures

It is fairly obvious that the process of displaying textures over any object involves a decision as to which pixel of the texture map (or texel) is placed at which pixel of the screen in relation to the position of the object. This will almost never be a one-to-one mapping as some form of enlargement or reduction will almost always be required. The process of selecting which pixel is used is called filtering.

The optional arguments **gMaxfil** and **gMinfil** control the selection of which texel is displayed when this enlargement or reduction, respectively, occurs. By default both filters are set to GNEAREST which means that the texel with its centre nearest the required pixel is displayed. This option is quick but can result in aliasing artifacts in the image.

The alternative option, GLINEAR uses a weighted average of a 2x2 square of texels that lie nearest the screen pixel, often giving much improved results. Note that the 2x2 square will include texels that lie outside the actual texture map when dealing with the edges. When the texture map is repeated (**gWraps** or **gWrapt** = GREPEAT), the 2x2 square includes texels from the opposite edge of the texture map, whereas when the texture map is clamped (**gWraps** or **gWrapt** = GCLAMP), the 2x2 square includes the border colour as described below.

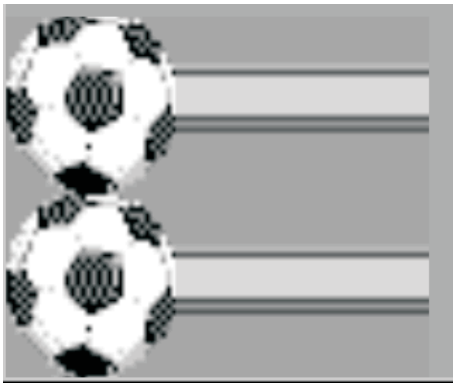
Changing the filters to GLINEAR in the previous example (without setting a border colour) causes the clamped edge to be merged with the default background grey causing the following effect:

C Code

```
gSetTextureMappingMode(GOVERLAY,gWraps,GCLAMP,
            gMaxfil,GLINEAR,gMinfil=GLINEAR,0);
```

F90 Code

```
call gSetTextureMappingMode(GOVERLAY,gWraps=GCLAMP, &
            gMaxfil=GLINEAR,gMinfil=GLINEAR)
```



Use of GLINEAR Filter

Where a texture map has additional mipmaps supplied with it (see Multiple Texture Maps above), four additional options for **gMinfil** are used to determine which mipmap and which texel is used when reducing the texture map to wrap ever smaller copies of the object. These are:

- GNEARESTNEAREST - Nearest mipmap using nearest texel filter
- GNEARESTLINEAR - Nearest mipmap using linear texel filter
- GLINEARNEAREST - Linear interpolated mipmap using nearest texel filter
- GLINEARLINEAR- Linear interpolated mipmap using linear texel filter

The smoothest results are obtained using the GLINEARLINEAR method, but at the expense of greater computational requirements.

Note that all mipmaps are ignored if **gMinfil** is set to GNEAREST or GLINEAR.

Texture Border Colour

When a texture map is clamped and the reduction or enlargement texture filter is set to GLINEAR, or linear texel filtering is used with mipmaps, the weighted linear average of 2x2 texels will include texels outside the actual texture map when calculating values at the edges. In these cases the border colour of the texture map is significant in these calculations.

The border colour is set in one of two ways; either the texture map can have an extra row/column of pixels along all its four edges (see page 346), or a fixed colour can be assigned using the optional argument **gBorderCol** to the routine gSetTextureMappingMode(). This value may be a colour index or a 24bit RGB triplet returned from the gTrueCol() function.

Setting the border colour to white in the above example gives the following effect:

C Code

```
gSetTextureMappingMode(GOVERLAY,gWraps,GCLAMP,
            gMaxfil,GLINEAR,gMinfil=GLINEAR,gBorderCol=GWHITE,0);
```

F90 Code

```
call gSetTextureMappingMode(GOVERLAY,gWraps=GCLAMP, &
            gMaxfil=GLINEAR,gMinfil=GLINEAR,gBorderCol=GWHITE)
```

Texture Border Colour

Texture Mapping Enquiry

The current setting of the texture mapping mode and all its attributes can be enquired using the routine:

gEnqTextureMappingMode(attribs)

where **attribs** is a structure of type GTEXATT containing the current texture mode and each of the elements described above.

Chapter 21

3D TRANSFORMATIONS

3D Transformations Introduction

As with 2D, routines are also provided in GINO to enable 3D geometric transformations: shift, rotate, scale and shear to be applied to definitions of lines and objects. These are commonly called modelling transformations as they are used to position and/or place various components of a complex model.

Modelling transformations are distinct from defining the overall ‘view’ of the mode which might apply perspective or non-perspective transformations (see page 385).

When a transformation routine is called, a new axis system (termed ‘space axes’) is created and subsequent drawing and positioning coordinates are considered in relation to this new axis system. When transformations are being combined, each transformation is relative to the axis system set up by the previous transformation.

To apply transformation to an object, the transformation routines must be called before the drawing routines. Once a transformation routine is called, the transformation that has been set up affects all subsequent drawing. Transforming can be switched on or off at any stage in a program or can be reset or modified (see page 371).

Great care should be exercised when using compounded transformations. Users are recommended to avoid mixing 2-D and 3-D routines, as this leaves the Z-plane undefined.

To illustrate transformations a routine man(), which defines a gingerbread man, is used. The original axes are drawn as solid lines and denoted X and Y. The space axes set up as a result of the transformations are denoted in the diagrams by X1, Y1 and Z1 and are shown as dashed lines.

Current Transformation

When using transformations, each point drawn is operated on by the total combined effect of previously called transformation routines. This effect is termed the current transformation. It is updated each time a transformation routine is called. A mathematical explanation of the transformation mechanism is given in Technical Information.

Simple 3D Transformations

3D Shifting

Shifting specifies the vector increments through which the origin is shifted from the origin of the previous axis system.

Shifting enables objects to be repositioned anywhere in the drawing area. The routine for shifting is:

gShift3D(dx, dy, dz)

For example - to draw an object shifted by 50.0mm in the X direction, 30.0mm in the Y direction and 60.0mm in the Z direction:

```
gShift3D(50.0,30.0,60.0);
object();
```

```
call gShift3D(50.0,30.0,60.0)
call object
```

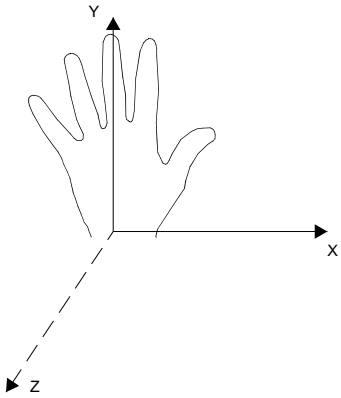
3D Rotation

The routine for 3-D rotation is:

gRotate3D(axis, angle)

The first argument specifies the axis about which the other two are rotated. The angle is specified in degrees and positive rotation is as shown below. (This is best remembered using the right-hand rule as follows. Holding out your right hand, palm towards you and the thumb extended, your thumb points in the direction of positive X, your fingers in positive Y, and your palm pushes along positive Z).

Example:



3-D Right-hand rule

- To draw a gingerbread man rotated through 120° about the X axis:

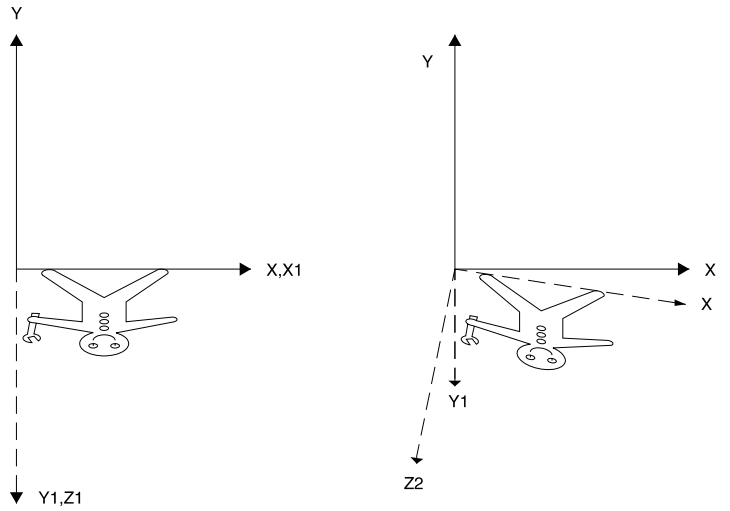
```
/* Rotate XY plane about X axis*/
   gRotate3D(GXAXIS,120.0);
   man();
```

```
! Rotate XY plane about X axis
  call gRotate3D(GXAXIS,120.0)
  call man
```

- Then negatively rotate about the Y axis by 10°. Note the new Y axis is where the original Z axis was before any transformations (compare the examples in the figure below):

```
gRotate3D(GYAXIS,-10.0);
man();
```

```
call gRotate3D(GYAXIS,-10.0)
call man
```



Rotation

Permutating the Axes

The user can specify the axes which are to be vertical and horizontal. The routine is:

gSetViewAxis(nh, nv)

The arguments specify which two axes are to be horizontal and vertical.

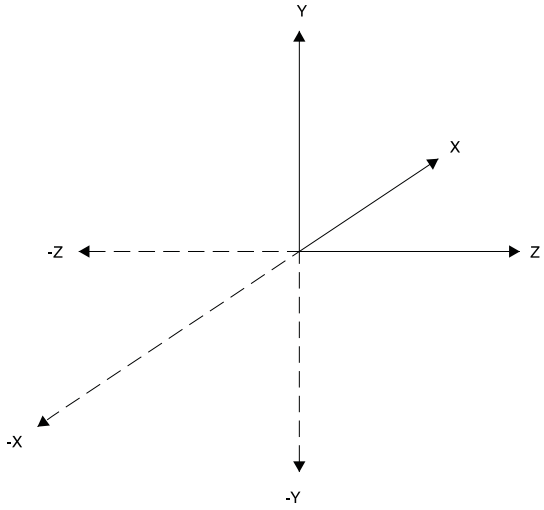
Example:

- To specify that the Z axis is to be horizontal and the Y axis is to be vertical:

```
gSetViewAxis(GZAXIS, GYAXIS);
```

```
call gSetViewAxis(GZAXIS, GYAXIS)
```

The resulting axis system is shown below:



Modified Axis system

The position of the third axis is such that a right-handed system is preserved.

3D Scaling

The routine for scaling is:

gScale3D(sx, sy, sz)

The arguments specify the amount by which the axes are to be scaled. Values of greater than 1.0 give magnification and values between 0.0 and 1.0 give reduction. If one or more arguments are negative, then a mirror image is produced.

Examples:

- To draw a gingerbread man uniformly scaled by 0.5 in all directions:

```
/* Scale all axes */
   gScale3D(0.5,0.5,0.5);
   man();
```

```
! Scale all axes
  call gScale3D(0.5,0.5,0.5)
  call man
```

- To draw a fat gingerbread man scaled in X by 2:

```
gScale3D(2.0,1.0,1.0);
man();
```

```
call gScale3D(2.0,1.0,1.0)
call man
```

3D Shearing

The routine for shearing is:

gShear3D(dep, dir, a)

The value of **dep** and **dir** can be GXAXIS, GYAXIS or GZAXIS where **dep** indicates which of the X,Y or Z axes is to be sheared (the dependent axis) parallel to the axis **dir**, the third axis being unaffected.

The argument **a** gives the tangent of the angle through which the axis **dep** is sheared.

Example:

- To draw a sheared gingerbread man such that the shear factor is 1.0 along the X axis parallel to the Y axis:

```
gShear3D(GXAXIS,GYAXIS,1.0);
man();
```

```
call gShear3D(GXAXIS,GYAXIS,1.0)
call man
```

Combining 3D Transformations

Using the Same 3D Transformation Type

When combining transformations of the same type, the general result is not dependent on the order in which the routines are called.

Example:

```
gScale3D(10.0,10.0,10.0);
gScale3D(3.0,3.0,3.0);
```

```
call gScale3D(10.0,10.0,10.0)
call gScale3D(3.0,3.0,3.0)
```

has the same effect as:

```
gScale3D(3.0,3.0,3.0);
gScale3D(10.0,10.0,10.0);
```

```
call gScale3D(3.0,3.0,3.0)
call gScale3D(10.0,10.0,10.0)
```

The above sequence of routines is equivalent to a single call to the routine with an arguments of 30.0, i.e. the combined effect is obtained by multiplying the arguments. In the case of transformation routines other than the scale routines, the cumulative effect is obtained by adding the arguments.

Example:

```
gRotate3D(GXAXIS,alpha);
gRotate3D(GXAXIS,beta);
```

```
call gRotate3D(GXAXIS,alpha)
call gRotate3D(GXAXIS,beta)
```

is equivalent to:

```
gRotate3D(GXAXIS,alpha+beta);
```

```
call gRotate3D(GXAXIS,alpha+beta)
```

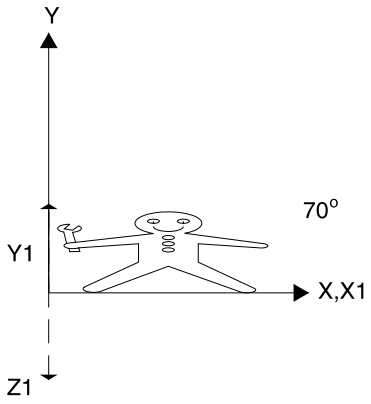
Combining 3-D Rotations

3-D rotations about different axes are the exception to the above generalization.

Example:

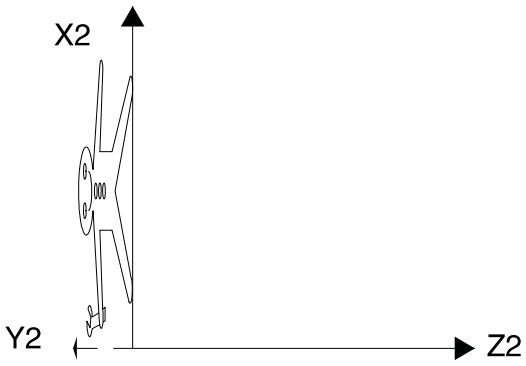
```
gRotate3D(GXAXIS,70.0);
man();
```

```
call gRotate3D(GXAXIS,70.0)
call man
```




```
/* Make man edge on */
   gRotate3D(GXAXIS,20.0);
/* Rotate through Y 90 degrees */
   gRotate3D(GYAXIS,90.0);
   gRotate3D(GXAXIS,-10.0);
   man();
```

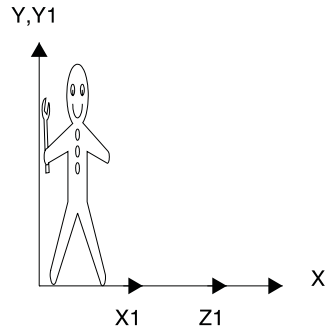
```
! Make man edge on
  call gRotate3D(GXAXIS,20.0)
! Rotate through Y 90 degrees
  call gRotate3D(GYAXIS,90.0)
  call gRotate3D(GXAXIS,-10.0)
  call man
```



whereas:

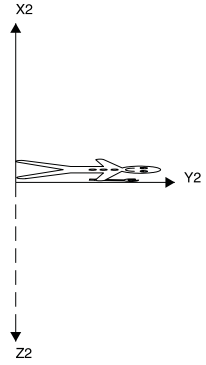
```
gRotate3D(GYAXIS,70.0);
man();
```

```
call gRotate3D(GYAXIS,70.0)
call man
```




```
/* Make man sideways on */
   gRotate3D(GYAXIS,20.0);
/* Rotate though X 90 degrees */
   gRotate3D(GXAXIS,90.0);
   gRotate3D(GYAXIS,10.0);
   man();
```

```
! Make man sideways on
  call gRotate3D(GYAXIS,20.0)
! Rotate though X 90 degrees
  call gRotate3D(GXAXIS,90.0)
  call gRotate3D(GYAXIS,10.0)
  call man
```



Using Different 3D Transformation Types

When combining transformations of different types, the effect obtained depends on the order in which the routines are called. Examples are shown in the chapter on 2D Transformations which show the principles that apply (see page 227). Again, the order in which transformations should be used to set up straightforward effects is:

Shift

Rotate

Scale

3D Transformation Enquiry

Finding the Current Drawing Position

The position of the pen at any given time can be described in terms of its coordinates relative to the origin of the drawing area; these are termed ‘picture coordinates’. The pen position can also be given in terms of its coordinates relative to the current local axis system; these are the space coordinates. In general, objects are specified in terms of space coordinates and are drawn in terms of picture coordinates; that is space coordinates are transformed into picture coordinates.

At any stage in a program, the pen position can be obtained (in either picture or space coordinates) by using one of the following routines:

gEnqPicturePos(point)

gEnqSpacePos(point)

Each of which returns a structure of type GPOINT3 structure, the elements of which are set to the current X,Y,Z coordinates expressed in current units.

3D Untransforming

The space coordinates of any point of which the picture coordinates are known, can be obtained using routine:

gUntransformPoint3D(xp, yp, zp, point)

where all elements of **point** are set to zero if the transformation contains perspective.

The routine gUntransformHomogPoint3D() is provided to convert from four-dimensional homogeneous coordinates to space coordinates:

gUntransformHomogPoint3D(xh, yh, zh, wh, point)

Homogeneous coordinates are normally obtained by calling gTransformHomogPoint3D(). They are related to picture coordinates in the following way:

xp = xh/wh

yp = yh/wh

zp = zh/wh

If the current transformation contains perspective, gUntransformHomogPoint3D() checks to see that the supplied position is consistent with the transformation and if not, outputs a warning message.

Point Testing of Current 3D Transformation

The routine gTransformPoint3D() enables the user to see what would happen to a point if it were subject to the current transformation.

gTransformPoint3D(xs, ys, zs, point)

transforms the space coordinate position (**xs**, **ys**, **zs**) into picture coordinates and returns the value of the picture coordinates in (**point.x, point.y**, **point.z**).

In the case where no current transformation exists, (**xs**, **ys**, **zs**) and (**point.x**, **point.y**, **point.z**) will have the same value.

When a perspective transformation is current, gTransformPoint3D() may not always be able to produce a valid result and in this case it would output GINO warning 15 - point lies behind the eye plane and does not transform. gTransformHomogPoint3D() is provided to cater for this sort of situation:

gTransformHomogPoint3D(xs, ys, zs, point, wh)

The transformed position is returned in four-dimensional homogeneous coordinates which are related to picture coordinates in the following way:

xp = xh/wh

yp = yh/wh

zp = zh/wh

The relationship obviously breaks down when **wh** = 0.0. This occurs when the point lies on the eye plane. When **wh** is less than zero, the point lies behind the eye plane. In both cases, the point (**xs**,**ys**,**zs**) does not project onto the view plane in any meaningful way. Note that **wh** = 1.0 when the point lies on the view plane.

Chapter 22

TRANSFORMATION CONTROL

Transformation Control Introduction

To enable transformations to be used effectively and to facilitate the organization of programs using them, routines are provided for:

- Setting transformation modes
- Switching transformations off and on
- Reinitializing
- Saving and re-using transformation sequences

View Transform Mode

GINO will, by default, use the most efficient method of defining the required transformation state to the currently nominated device. On 3D devices, this means that transformation and viewing information will be passed to the device allowing 3D Space coordinates to be handled directly through the devices 3D pipeline. On 2D devices, 3D Space coordinates are transformed through the current transformation, viewing and viewport settings to 2D Device Coordinates through the GINO pipeline (see page 36).

There may be instances where an application may wish to use a consistent method across 2D and 3D devices or switch off the 3D hardware pipeline for a particular operation. Two routines are provided to set and enquire the current view transform mode:

gSetViewTransformMode(mode)

gEnqViewTransformMode(mode)

where **mode** is either GHARDWARE or GSOFTWARE as appropriate. It is obviously not possible to switch 2D devices to use a hardware view transform mode.

Transformation State

The current transformation state can be switched off and on again by using the routine:

gSetTransform(sw)

When **sw** is GOFF or GRESET, transformations previously set up have no effect on subsequent drawing. If **sw** is GOFF, the state of the current transformation is preserved and can be restored by a call to gSetTransform(GON). The ability to switch transformations off enables items, such as the title of a drawing, to be positioned directly on the drawing area, i.e. in terms of the original (picture) axes.

Transformations are automatically switched off when a device nomination routine is called. They are switched on by the first transformation routine that is executed after a device has been nominated or when transformations have been switched off.

Reinitializing

The current transformation may be reset to its original, null state by calling gSetTransform() with **sw**=GINIT or GRESET. Until another transformation is set up, (after a call to gSetTransform(GINIT)) subsequent drawing is operated on by a null transformation. However, it is more efficient to draw with transforming switched off, although the result is the same.

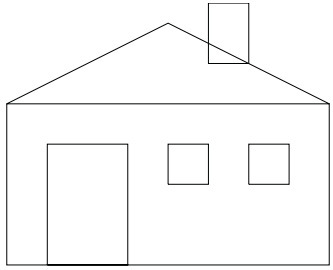
For example the following program draws the picture shown in the figure below and illustrates the use of the transformation control routines:

C code

```
#include <gino-c.h>
main()
{
void box(void);

   gOpenGino();
/* Nominate device */
   xxxxx();
   gNewDrawing();
   gShift2D(10.0,30.0);
   gScale2D(4.0,-2.0);
/* Draw walls */
   box();
/* Reinitialize */
   gSetTransform(GRESET);
/* Draw roof */
   gDrawLineTo2D(30.0,40.0);
   gDrawLineTo2D(50.0,30.0);
   gShift2D(30.0,20.0);
   gScale2D(0.5,0.5);
/* Draw LH window */
   box();
/* Shift in scaled system */
   gShift2D(20.0,0.0);
/* Draw RH window */
   box();
/* Reinitialize */
   gSetTransform(GINIT);
   gShift2D(15.0,10.0);
   gScale2D(1.0,1.5);
/* Draw door */
   box();
/* Select picture axes */
   gSetTransform(GOFF);
   gMoveTo2D(10.0,5.0);
   gDisplayStr("HOUSE");
/* Select space axes - ie.(shift (15.0,10.0),
                           scale (1.0,1.5))  */
   gSetTransform(GON);
   gScale2D(0.5,0.5);
   gShift2D(40.0,100.0/3.0);
/* Draw chimney */
   box();

   gCloseGino();
}

void box(void)
{
GLIMIT rect = {0.0,10.0,0.0,10.0)};
      gFillRect(-1,0,&rect);
}
```

F90 code

```
program house
use gino_f90

  call gOpenGino
! Nominate device
  call xxxxx
  call gNewDrawing
  call gShift2D(10.0,30.0)
  call gScale2D(4.0,-2.0)
! Draw walls
  call box
! Reinitialize
  call gSetTransform(GRESET)
! Draw roof
  call gDrawLineTo2D(30.0,40.0)
  call gDrawLineTo2D(50.0,30.0)
  call gShift2D(30.0,20.0)
  call gScale2D(0.5,0.5)
! Draw LH window
  call box
! Shift in scaled system
  call gShift2D(20.0,0.0)
! Draw RH window
  call box
! Reinitialize
  call gSetTransform(GINIT)
  call gShift2D(15.0,10.0)
  call gScale2D(1.0,1.5)
! Draw door
  call box
! Select picture axes
  call gSetTransform(GOFF)
  call gMoveTo2D(10.0,5.0)
  call gDisplayStr('HOUSE')
! Select space axes - ie.(shift (15.0,10.0),
!                         scale (1.0,1.5))
  call gSetTransform(GON)
  call gScale2D(0.5,0.5)
  call gShift2D(40.0,100.0/3.0)
! Draw chimney
  call box

  call gCloseGino
  stop
end
subroutine box
use gino_f90
type (GLIMIT) :: rect = GLIMIT(0.0,10.0,0.0,10.0)
      call gFillRect(-1,0,rect)
end
```



Transformations Matrix Control

The data representing the current transformation is held internally as an array which can be copied, reset and modified using the following sets of routines:

gPushTransform()
gPopTransform()
gSaveTransform()
gRestoreTransform()
gGetTransform2D()
gSetTransform2D()
gModifyTransform2D()
gGetTransform3D()
gSetTransform3D()
gModifyTransform3D()

Routines from different sets are not compatible with each other, e.g. gPopTransform() can only restore the effect saved by gPushTransform().

Push and Pop Transformation Matrix

These routines provide the easiest and most convenient way of saving and restoring a sequence of transformations.

gPushTransform()

gPopTransform()

The routine gPushTransform() has no effect on the current transformation itself; thus copies made in this way are stored internally. Up to ten different transformation sequences can be held simultaneously by successive calls to gPushTransform(). A sequence that has been saved using gPushTransform() is retrieved by a matching call to gPopTransform(). This causes the current transformation to be reset and the saved copy to be removed; thus each copy can only be used once.

It is only possible to restore copies in the order in which they were saved. In addition to saving and restoring transformations, gPushTransform() and gPopTransform() can be used on entry to, and exit from, a routine in which transformations are used locally. This ensures that the state of the current transformation in the calling program is unchanged by the routine.

Saving and Restoring Transformation Matrix

The routines gSaveTransform()/gRestoreTransform() make a copy (in an internal storage area) of the total state of transformation.

gSaveTransform()

gRestoreTransform()

That is the space axis system together with the indication of whether or not it is current. Since the picture axis system never changes, the information saved by gSaveTransform() is sufficient to enable gRestoreTransform() to reset the state at a subsequent point in the program. Any modifications made by calling a transformation routine or by calling gSetTransform() between the call to gSaveTransform() and the call to gRestoreTransform() will be lost when gRestoreTransform() is called. Only one copy of the axis system may be stored, and so a call to gSaveTransform() will overwrite information stored by any previous calls to the routine.

Getting and Setting Transformation Matrix

gGetTransform2D(a)

gGetTransform3D(b)

gSetTransform2D(a)

gSetTransform3D(b)

These routines perform the same routine as gPushTransform() and gPopTransform(), but give the user more control. The basic difference is that sequences of transformations are saved by gGetTransform2D() and gGetTransform3D() in user arrays, instead of in internal GINO arrays. In 2-D applications only the part of the array representing the current transformation is used. Any number of transformation sequences can be stored by giving different structures as arguments. These may be recalled any number of times and in any order using gSetTransform2D() or gSetTransform3D(), which set the current transformation (or part of it) to the state represented by the specified array.

Modify Transformation Matrix

gModifyTransform2D(a)

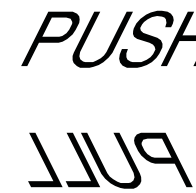
gModifyTransform3D(b)

Using these routines, transformation matrices that have been saved by gGetTransform2D() and gGetTransform3D() can be updated instead of replaced.

For example - the following statements show how gSetTransform2D() and gModifyTransform2D() can be used to reproduce an effect (saved in structure t) in different circumstances:

```
#include <gino-c.h>
   GMAT2D t;

   gSetCharTransformMode(GSOFT);
   .........
/* Set up effect */
   gScale2D(3.0,3.0);
   gShear2D(GYAXIS,0.5);
/* Save effect */
   gGetTransform2D(&t);
   .........
/* Switch transformation off */
   gSetTransform(GOFF);
   gMoveTo2D(15.0,30.0);
/* Restore */
   gSetTransform2D(&t);
   gDisplayStr("PUSH");
   .........
/* Reinitialize */
   gSetTransform(GINIT);
/* Apply transformations */
   gShift2D(45.0,10.0);
   gScale2D(-1.0,1.0);
/* Add effect */
   gModifyTransform2D(&t);
   gMoveTo2D(0.0,0.0);
   gDisplayStr("PULL");
```

```
use gino_f90
type (GMAT2D) :: t

  gSetCharTransformMode(GSOFT)
  .........
! Set up effect
  gScale2D(3.0,3.0)
  gShear2D(GYAXIS,0.5)
! Save effect
  gGetTransform2D(t)
  .........
! Switch transformation off
  gSetTransform(GOFF)
  gMoveTo2D(15.0,30.0)
! Restore
  gSetTransform2D(t)
  gDisplayStr('PUSH')
  .........
! Reinitialize
  gSetTransform(GINIT)
! Apply transformations
  gShift2D(45.0,10.0)
  gScale2D(-1.0,1.0)
! Add effect
  gModifyTransform2D(t)
  gMoveTo2D(0.0,0.0)
  gDisplayStr('PULL')
```



Transformation Matrix Building

The following routines are available for creating and storing transformation sequences without affecting the current transformation set up.

gBuildMatrix2D(x0, y0, dx, dy, angle, sx, sy, t)

gBuildMatrix3D(x0, y0, z0, dx, dy, dz, angx, angy, angz, sx, sy, sz, t)

gCombineMatrix2D(a, x0, y0, dx, dy, angle, sx, sy, t)

gCombineMatrix3D(a, x0, y0, z0, dx, dy, dz, angx, angy, angz, sx, sy, sz, t)

The routines gBuildMatrix2D() and gBuildMatrix3D() create a new transformation matrix as a combination of scaling factors, a rotation and translation in that order about a fixed point. The variables determining these values are as follows :

	2D Matrix	3D Matrix
Scaling Factors:	**sx,sy**	**sx,sy,sz**
Rotation:	**angle**	**angx,angy,angz**
Translation:	**dx,dy**	**dx**,**dy**,**dz**
Fixed Point:	**x0,y0**	**x0,y0,z0**

The routines gBuildMatrix2D() and gBuildMatrix3D() create a new transformation matrix and return it as **t**.

An existing transformation matrix array may have further transformations applied to it using the routines gCombineMatrix2D() and gCombineMatrix3D(). A combination of a shift followed by the application of scaling factors and a rotation about a fixed point may be combined with the input matrix array.

The existing transformation matrix is passed to the routines gCombineMatrix2D() and gCombineMatrix3D() as the parameter **a**. The new transformations are combined with **a** in accordance with the current transformation mode (as set gSetTransformMode()), the resulting transformations are returned as **t**.

The resulting sequence, stored in the transformation matrix array **t**, may be implemented using gSetTransform2D() or gSetTransform3D() depending on whether two or three dimensions are used.

Example showing Building and Combining Transformation matrices

The following program outputs the shape of a house as shown previously by calling the routine ‘house’ three times, the first without any transformations, the second with a shift and rotation, and the third with a scale, shift and rotation. The routine ‘house’ draws a house of width 400.0 and height 300.0 with the origin at the bottom left-hand corner.

C code

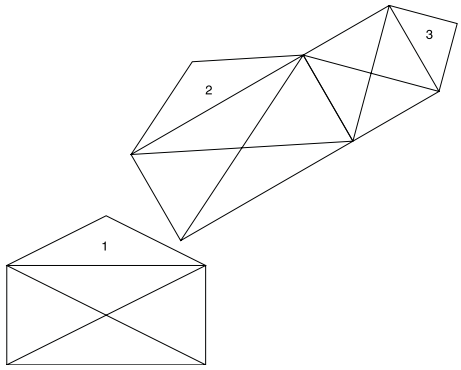
```
#include <gino-c.h>
main()
{
   void house(int number);
   GMAT2D a,t;
```

```
/* Build first transformation in a */
   gBuildMatrix2D(0.,0.,350.,250.,30.,1.,1.,&a);

/* Draw house number 1 - No transformations */
   house(1);


/* Set first transformation and draw house number 2 */
   gSetTransform2D(&a);
   house(2);


/* Add second transformation to first
   and draw house number 3 */
   gCombineMatrix2D(&a,200.,100.,300.,0.,270.,0.5,1.,&t);
   gSetTransform2D(&t);
   house(3);
}


void house(int number)
{
   gMoveTo2D(  0.,  0.);
   gDrawLineTo2D(400.,  0.);
   gDrawLineTo2D(400.,200.);
   gDrawLineTo2D(  0.,200.);
   gDrawLineTo2D(  0.,  0.);
   gDrawLineTo2D(400.,200.);
   gDrawLineTo2D(200.,300.);
   gDrawLineTo2D(  0.,200.);
   gDrawLineTo2D(400.,  0.);

   gMoveTo2D(190.,230.);
   gDisplayInteger(number,1);
}
```

F90 code

```
program build_house
use gino_f90

type (GMAT2D) ::  a,t

!  Build first transformation in a
    call gBuildMatrix2D(0.,0.,350.,250.,30.,1.,1.,a)

!  Draw house number 1 - No transformations
   call house(1)

!  Set first transformation and draw house number 2
   call gSetTransform2D(a)
   call house(2)

!  Add second transformation to first
!  and draw house number 3
   call gCombineMatrix2D(a,200.,100.,300.,0.,270.,0.5,1.,t)
   call gSetTransform2D(t)
   call house(3)
   stop
   end
```

```
subroutine house(number)
integer number
   call gMoveTo2D(  0.,  0.)
   call gDrawLineTo2D(400.,  0.)
   call gDrawLineTo2D(400.,200.)
   call gDrawLineTo2D(  0.,200.)
   call gDrawLineTo2D(  0.,  0.)
   call gDrawLineTo2D(400.,200.)
   call gDrawLineTo2D(200.,300.)
   call gDrawLineTo2D(  0.,200.)
   call gDrawLineTo2D(400.,  0.)

   call gMoveTo2D(190.,230.)
   call gDisplayInteger(number,1)
   end
```



The first transformation shifts the origin (bottom left hand corner) of the house by (350.0, 250.0) in the current units. The house is then rotated about the fixed point (0.0,0.0) by 30.0° and drawn without any scaling.

The second transformation shifts the house by 300.0 along the line of the current rotation angle. Using the fixed point as the centre of the house (200.0,100.0) the house is rotated by 270.0°. The house is scaled by 0.5 parallel to the floor of the house; the scaling is relative to the fixed point position, and therefore reduces the width of the house towards the centre.

Transformation Enquiry

To enquire about the state of transformation the user should use the transformation enquiry routine:

gEnqTransformState(ntran, dim, mode)

where:

ntran indicates	transformations off (= GOFF) or on (= GON)
dim indicates	transformations off (= GOFF)
	2-D transformations (= GON2D)
	3-D transformations (= GON3D) with no perspective
	3-D transformations (= -3) with perspective

and **mode** indicates space mode (GSPACE) or picture mode (GPICTURE).

Settings of **mode** are described below. The default when transformations are switched off is: **ntran**= GOFF, **dim**= GON2D and **mode**= GSPACE.

Transformation Mode

There are two possible methods of using transformations in GINO:

- In picture mode
- In space mode

Space mode is the default.

In space mode, each transformation is relative to the current space axis, that is it is executed with reference to the previously set up transformation. In picture mode, transformations are always relative to the original axis system - the picture axis. This is useful when viewing whole pictures since the whole picture can be shifted or rotated with reference to the original axis.

The routine for switching from one mode to the other is:

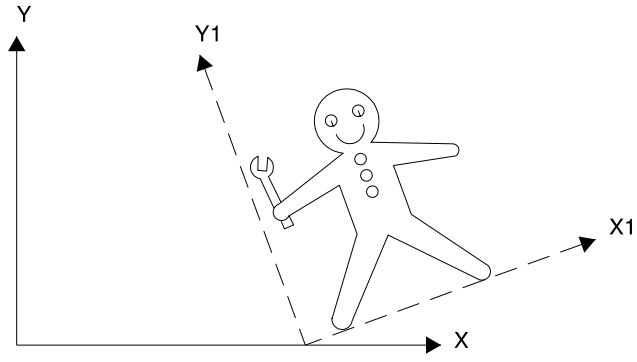
gSetTransformMode(mode)

The argument **mode** may take the value GSPACE or GPICTURE. The value GSPACE switches to space mode and GPICTURE switches to picture mode.

When working in picture mode, transformations must be applied in reverse order to that followed in space mode if the same effect is to be achieved. To produce, for example, the transformed gingerbread man in the figure below in picture mode, it would be necessary to call the routines in the following order:

```
gRotate2D(20.0);
gShift2D(50.0,0.0);
man();
```

```
call gRotate2D(20.0)
call gShift2D(50.0,0.0)
call man
```

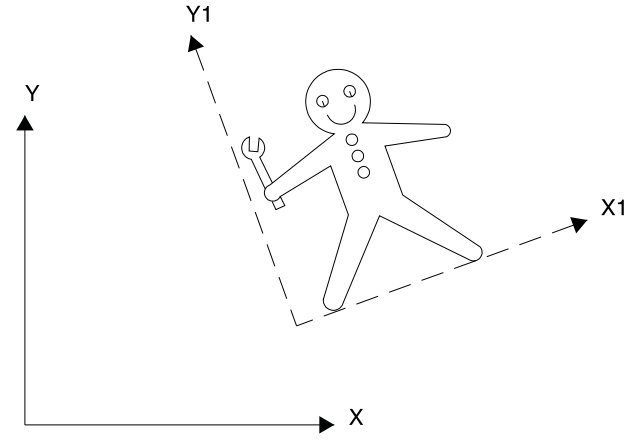


However, in space mode the order must be:

```
gShift2D(50.0,0.0);
gRotate2D(20.0);
man();
```

```
call gShift2D(50.0,0.0)
call gRotate2D(20.0)
call man
```

If the sequence gShift2D(), gRotate2D() were used in picture mode the result would be as shown below. This would produce the same result as the sequence gRotate2D(), gShift2D() when used in space mode.



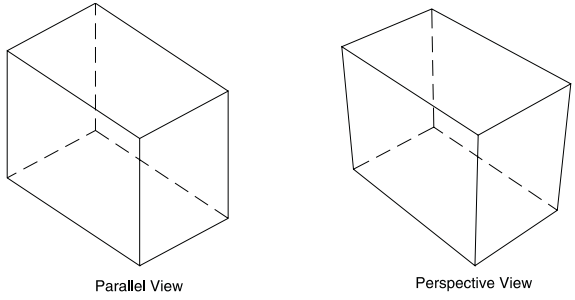
Chapter 23

VIEWING

Viewing Introduction

Models drawn in 3-D in GINO exist in a user-defined world with its own axes and dimensions. The viewing routines provide a window into that world and offer extensive control over the images ultimately seen. A model in the 3-D world is mapped onto a 2-D plane. Parallel or perspective views of the model can be set up. The routines allow a user to change his point of view of a model in respect of direction and distance. When the resulting image is displayed, it often appears as if a rotation or positioning has occurred. This is useful for many types of visualization, from engineering drawings to stereoscopic projection.

Parallel (isometric) views are suitable for technical work where dimensions may need to be attached to, or read from, a drawing. Perspective views give a better impression of how the model will actually appear. The figure below illustrates the difference.



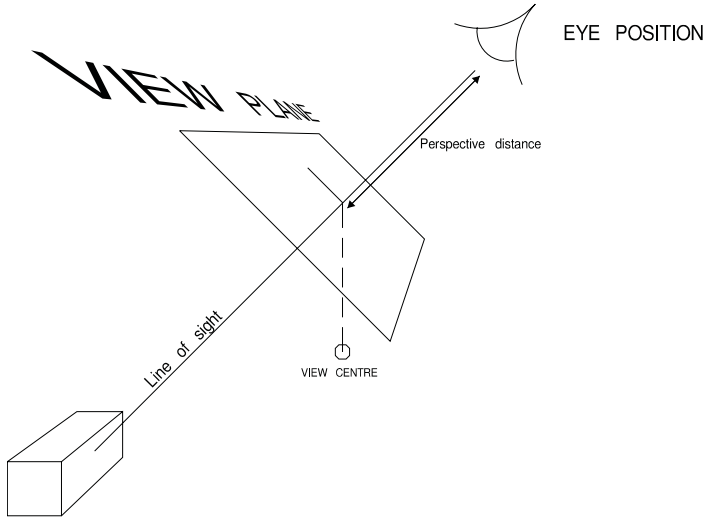
Comparison of Parallel and Perspective Views

Useful Concepts

Viewing essentially involves taking a model's three dimensional coordinates and operating on them in some way to produce a two dimensional image on a screen or piece of paper. If perspective is involved, the routines need to know three things:

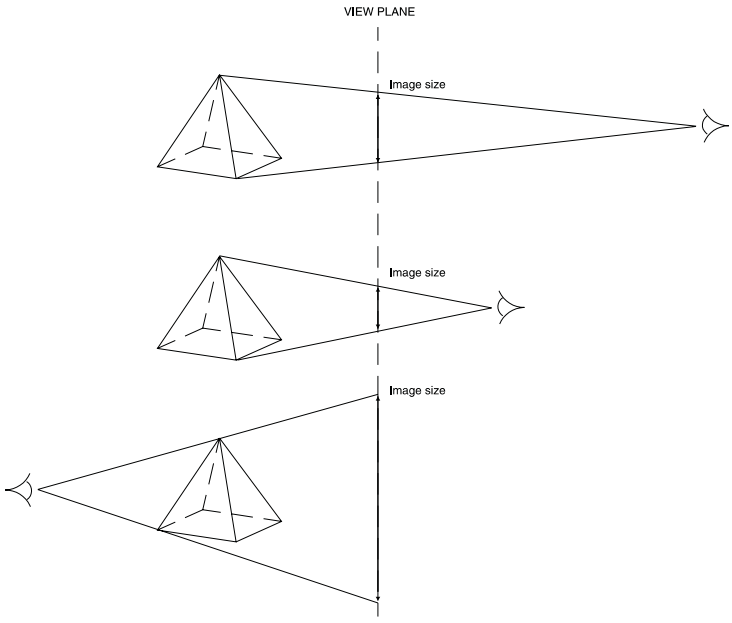
- The point from which the object is viewed - the **eye position**
- The direction of viewing - the **line of sight**
- The position of a plane in the model's world onto which the coordinates are to be projected. This plane is known as the **view plane**. Points on the view plane can be mapped directly onto a screen or paper.

The view plane is perpendicular to the line of sight. The distance between the eye and the view plane is known as the **perspective distance**. The point where the line of sight meets the view plane is the **view centre**.



Definitions for Viewing Routines

The previous diagram illustrates these definitions, while the effects of different eye positions are illustrated in the next figure.



Varying Eye Position and Viewing Directions

From View Plane to Paper

The two-dimensional image projected onto the view plane is closely related to what appears on the screen. The mapping of points may be controlled by using the routines described later in the section, however, GINO assumes sensible defaults. For example, by default the view centre is positioned at the centre of the current window or drawing area. If no window has been explicitly defined the device limits are used.

Similarly, the image is oriented to make the Y-axis of the model's world parallel with that of the drawing device. In the exceptional case where the line of sight is parallel to the Y axis, world and picture X axes are aligned.

The Basic Viewing Routines

Initially viewing parameters are supplied to one of three routines, gDefinePerspView(), gDefineSphericalView() or gDefineParallelView(). One of these must be set up before the model can be viewed. The first two deal with perspective drawing, while gDefineParallelView() establishes a parallel view. The viewing transformation itself is then created by a call to the routine:

gUpdateView()

The following sections describe the three basic routines in full and show exactly how to use them. In each case, one of the objects drawn is a box with sides 30, 40 and 50mm with one vertex at the origin of the world space (a listing of the routine CUBOID appears later).

Relating what happens on the output device to what is happening in world space may require some thought - working through the examples and altering the parameters will shed more light on the whole subject.

Perspective Views of a Volume

Routine gDefineSphericalView() provides a straightforward means of drawing an object whose coordinates are readily available. gDefineSphericalView() uses the fact that a sphere viewed from any direction is circular.

The user describes a sphere which completely encloses his model, and defines a viewing direction and a perspective distance (see the figure below). GINO can then calculate an eye point which causes the circular projection to fill the current window or drawing area as completely as possible.

gDefineSphericalView(xc, yc, zc, r, dx, dy, dz, d)

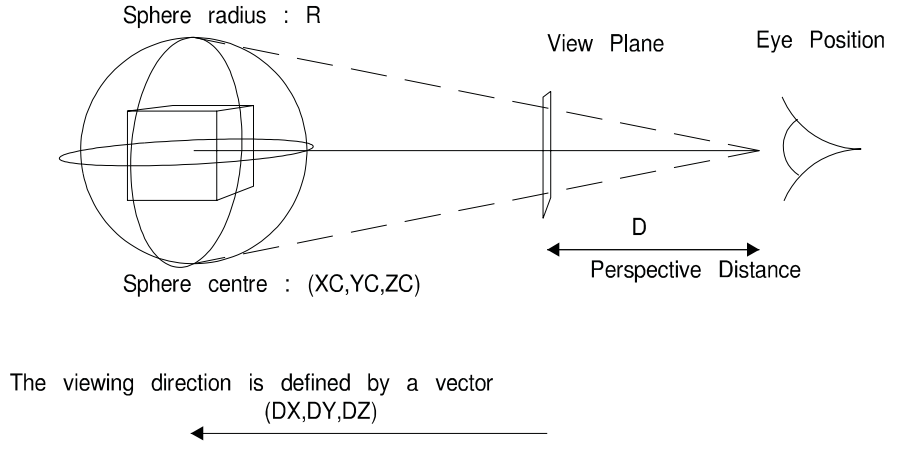
The sphere's centre is at a point (**xc**, **yc**, **zc**) in space coordinates and its radius on the view plane in picture coordinates is **r**. The line of sight is in direction (**dx**, **dy**, **dz**) and the perspective distance is **d**.

Thus to use gDefineSphericalView(), the statements:

```
gDefineSphericalView(xc,yc,zc,r,
      dx,dy,dz,d);
gUpdateView();
```

```
call gDefineSphericalView( &
          xc,yc,zc,r,dx,dy,dz,d)
call gUpdateView
```

are all that is required.



The Viewing Sphere

The figure below illustrates the use of gDefineSphericalView() by looking at a box, using the device limits as a window.

Notice that the line of sight is always towards the view centre; to look from a point in positive X, Y and Z towards a point near the origin, the direction vector requires **negative** components. The line of sight passes through the enclosing sphere's centre.

Code for a box viewed using a spherical view, as shown in the figure below, follows:

C code

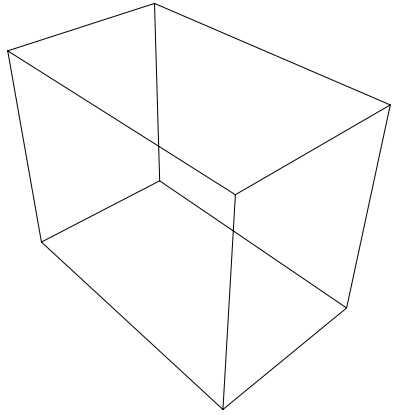
```
#include <math.h>
#include <gino-c.h>
#include "subs.h"
main ()
{
   GLIMIT window = {0.0, 180.0, 0.0, 140.0};
   float xmin,xmax,ymin,ymax,zmin,zmax;
   float xc,yc,zc,delx,dely,delz,radius,dpersp;

   gOpenGino();
   qMwin();

   gSetWindow2D(&window);
```

```
/* Define Max dimensions for the volume to be viewed */
   xmax=50.0;
   ymax=40.0;
   zmax=30.0;
   xmin=0.0;
   ymin=0.0;
   zmin=0.0;
/* Calculate coordinates of volume's centre */
   xc=0.5*(xmax+xmin);
   yc=0.5*(ymax+ymin);
   zc=0.5*(zmax+zmin);
/* Find radius of sphere */
   delx=xmax-xc;
   dely=ymax-yc;
   delz=zmax-zc;
   radius=sqrt(delx*delx+dely*dely+delz*delz);
   dpersp=200.0;
/* Establish view */
   gDefineSphericalView(xc,yc,zc,radius,
                        -1.0,-1.0,-0.9,dpersp);
   qUpdateView();
/* Draw box */
   cuboid(xmax,ymax,zmax);
   gSuspendDevice();
   gCloseGino();
}
#include "subs.c"
```

F90 code

```
program fig12_5
use gino_f90
TYPE (GLIMIT) :: window = GLIMIT(0.0, 180.0, 0.0, 140.0)
real xmin,xmax,ymin,ymax,zmin,zmax
real xc,yc,zc,delx,dely,delz,radius,dpersp

   call gOpenGino
   call gMwin
   call qSetWindow2D(window)
!  Define Maximum dimensions for the volume to be viewed
   xmax=50.0
   ymax=40.0
   zmax=30.0
   xmin=0.0
   ymin=0.0
   zmin=0.0
!  Calculate coordinates of volume's centre
   xc=0.5*(xmax+xmin)
   yc=0.5*(ymax+ymin)
   zc=0.5*(zmax+zmin)
! Find radius of sphere
   delx=xmax-xc
   dely=ymax-yc
   delz=zmax-zc
   radius=sqrt(delx*delx+dely*dely+delz*delz)
   dpersp=200.0
! Establish view
   call gDefineSphericalView(xc,yc,zc,radius, &
                             -1.0,-1.0,-0.9,dpersp)
   call gUpdateView
```

```
! Draw box
   call cuboid(xmax,ymax,zmax)
   call gSuspendDevice
   call gCloseGino
stop
end
include 'subs.f90'
```



A box viewed using a Spherical View

The figure below shows two different views of a large letter 'G' in user-defined windows. This object is drawn inside a box (in effect a three-dimensional window) of defined dimensions. The routine biggee() is listed later in the section.

Code for the figure below follows:

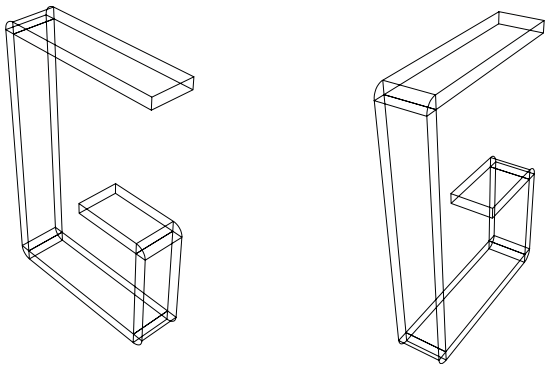
```
#include <math.h>
#include <gino-c.h>
#include "subs.h"
main ()
{
   GLIMIT window1 =
       { 0.0, 90.0, 0.0, 140.0},
       window2 =
       {90.0,180.0, 0.0, 140.0};
   float xmin,xmax,ymin,ymax;
   float zmin,zmax,xc,yc,zc,delx;
   float dely,delz,radius,dpersp;

   gOpenGino();
   gMwin();

   gSetWindow2D(&window1);
```

```
program fig12_6
use gino_f90

type (GLIMIT) :: window1 = &
   GLIMIT( 0.0, 90.0, 0.0, 140.0)
type (GLIMIT) :: window2 = &
   GLIMIT(90.0,180.0, 0.0, 140.0)
real xmin,xmax,ymin,ymax
real zmin,zmax,xc,yc,zc
real delx,dely,delz,radius,dpersp

   call gOpenGino
   call gMwin

   call gSetWindow2D(window1)
```

(As precious code up to the comment line '/* Establish view' */'

```
/*  Establish view */
   gDefineSphericalView(
         xc,yc,zc,radius,
        -1.0,-1.0,-0.9,dpersp);
   gUpdateView();
/* Draw letter G */
   biggee(xmax,ymax,zmax);
/* Set up new window */
    gSetWindow2D(&window2);
   dpersp=150.0;
/*  Establish new view */
   gSetTransform(GRESET);
   gDefineSphericalView(
          xc,yc,zc,radius,
          1.5,-1.2,-0.9,dpersp);
   gUpdateView();
/* Draw letter G */
   biggee(xmax,ymax,zmax);
   gSuspendDevice();
   gCloseGino();
}
#include "subs.c"
```

```
!  Establish view */
   call gDefineSphericalView( &
         xc,yc,zc,radius, &
         -1.0,-1.0,-0.9,dpersp)
   call gUpdateView
!  Draw letter G
   call biggee(xmax,ymax,zmax)
!  Set up new window
   call gSetWindow2D(window2)
   dpersp=150.0
!  Establish new view
   call gSetTransform(GRESET)
   call gDefineSphericalView( &
          xc,yc,zc,radius, &
          1.5,-1.2,-0.9,dpersp)
   call gUpdateView
!  Draw letter G
   call biggee(xmax,ymax,zmax)
   call gSuspendDevice
   call gCloseGino
stop
end
include "subs.f90"
```



Spherical Views with user defined windows

In the general case, given the coordinates of a box which could surround a particular object, the details of an enclosing sphere can be calculated easily. The code for the original box shows this happening. If the 'most negative' corner is (xmin,ymin,zmin), and the 'most positive' corner is (xmax,ymax,zmax), as in the figure below, the centre of the sphere is (xc,yc,zc) where:

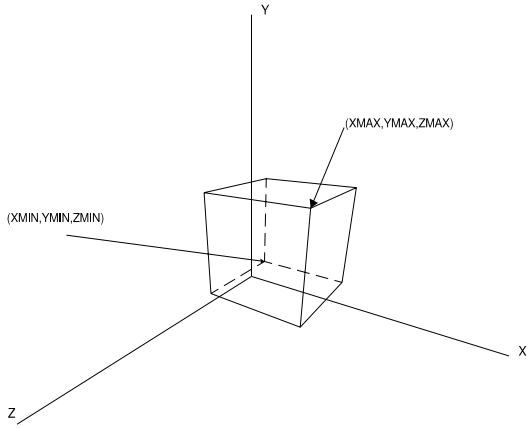
xc = (xmax+xmin)/2.0

yc = (ymax+ymin)/2.0

xc = (zmax+zmin)/2.0

and its radius is:

r = sqrt((xmax-xc)**2 +(ymax-yc)**2 +(zmax-zc)**2)



Calculating viewing sphere from enclosing box

Perspective View from a Point

Routine gDefinePerspView() offers an alternative approach to perspective viewing. Here, the user chooses an eye position, a viewing direction and the perspective distance.

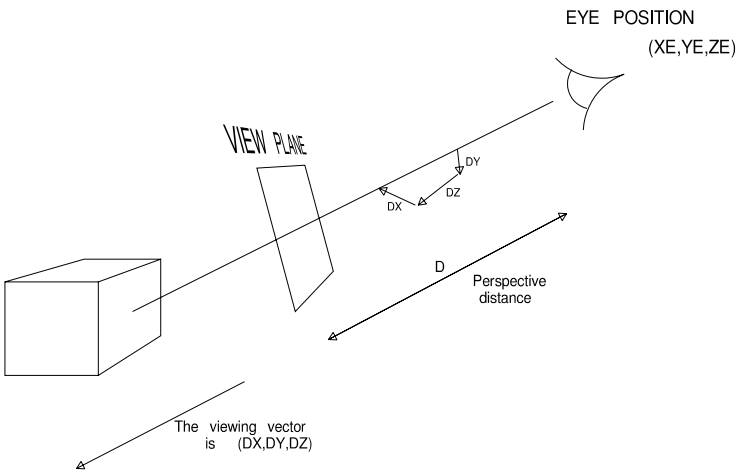
gDefinePerspView(xe, ye, ze, dx, dy, dz, d)

Selecting an eye position seems quite natural, but it is not always easy to make full use of the current window, something at which gDefineSphericalView() is very good. However, for visualization purposes gDefinePerspView() is more appropriate than gDefineSphericalView(). For example, it can be used to create a view of a model of an oil refinery as though standing on a particular walkway.

The figure below illustrates the parameters supplied to gDefinePerspView(). The eye is at (**xe**,**ye**,**ze**) and the viewing direction is along a vector (**dx**,**dy**,**dz**). The perspective distance is D.

```
gDefinePerspView(xe,ye,ze,
                 dx,dy,dz,d);
gUpdateView();
```

```
call gDefinePerspView(xe,ye,ze, &
                      dx,dy,dz,d)
call gUpdateView
```



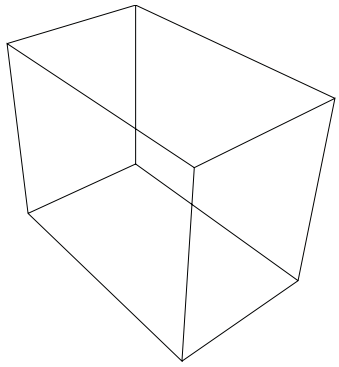
Perspective Viewing from a Point

In the following examples, the first of the figures below shows the familiar box, and the second shows the large letter ‘G’.

The code for the box shown below follows:

```
gDefinePerspView(
      100.0,90.0,80.0,
      -10.0,-9.0,-8.0,150.0);
gUpdateView();
cuboid(50.0,40.0,30.0);
```

```
call gDefinePerspView( &
      100.0,90.0,80.0, &
      -10.0,-9.0,-8.0,150.0)
call gUpdateView
call cuboid(50.0,40.0,30.0)
```

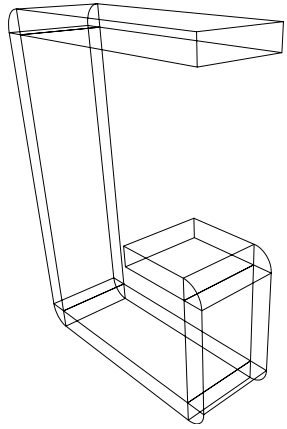


A Box viewed using Perspective View

Code for the big ‘G’ is shown here:

```
gDefinePerspView(
      120.0,90.0,80.0,
      -12.0,-9.0,-12.0,150.0);
gUpdateView();
biggee(60.0,80.0,20.0);
```

```
call gDefinePerspView( &
      120.0,90.0,80.0, &
      -12.0,-9.0,-12.0,150.0)
all gUpdateView
all biggee(60.0,80.0,20.0)
```



Perspective View of G

In the common case where a point on the line of sight is known, e.g. the centre of the object, the viewing direction is straightforward to calculate. If the view centre is also chosen to be this point the perspective distance can be easily calculated. Thus, given a point (xp,yp,zp) on the line of sight, the direction is given by:

dx=xp-xe

dy=yp-ye

dz=zp-ze

If the view plane is to pass through this point, the perspective distance, is given by:

d = sqrt(dx**2 + dy**2 + dz**2)

Parallel Projection

For parallel projection the eye position is irrelevant. All that needs to be supplied is a view direction and a view centre. (In this case the view centre is simply a point in the model's world which is mapped to the centre of the drawing window). Thus routine gDefineParallelView() is easy to use. The figure below shows what is involved.

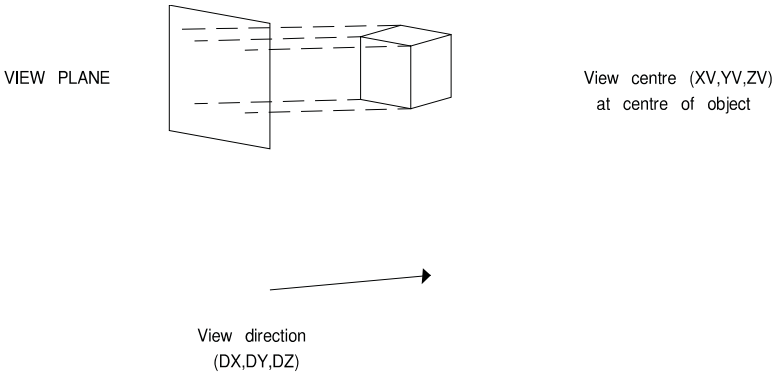
gDefineParallelView(dx, dy, dz, xv, yv, zv)

If the viewing direction is a vector (**dx**,**dy**,**dz**) and the view centre (**xv**,**yv**,**zv**) then:

```
gDefineParallelView(dx,dy,dz,
      xv,yv,zv);
gUpdateView()
```

```
call gDefineParallelView( &
                dx,dy,dz,xv,yv,zv)
call gUpdateView
```

sets up the required view.

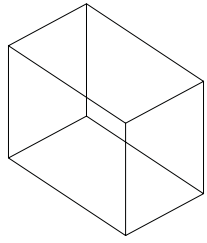


Parallel Viewing

The figure below illustrates a parallel view of a box, the code for which follows:

```
gDefineParallelView(
    -1.0,-1.0,-0.9,0.0,0.0,0.0);
gUpdateView();
cuboid(50.0,40.0,30.0);
```

```
call gDefineParallelView( &
     -1.0,-1.0,-0.9,0.0,0.0,0.0)
call gUpdateView
call cuboid(50.0,40.0,30.0)
```



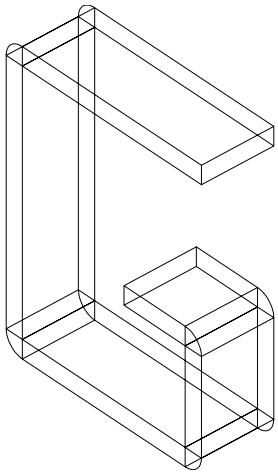
Box Viewed using Parallel Projection

The figure below illustrates the big ‘G’, the code for which follows:

As above, but:

```
biggee (45.0,60.0,15.0);
```

```
call biggee (45.0,60.0,15.0)
```



Parallel view of G

Setting Viewing Transformations

Viewing involves taking coordinates from a model's world and operating on them in some way to produce a two-dimensional image.

The operations depend on factors such as eye position, viewing direction and where the view plane falls. The three basic routines gDefineSphericalView(), gDefinePerspView() and gDefineParallelView() are a means of supplying these parameters. To create the final viewing transformation, gUpdateView() is then called.

If gSetTransform() is called with GRESET as its argument the current transformation is initialized. To re-establish a view, one of the three basic routines must be called prior to a call to gUpdateView(). However, if gSetTransform() is called with GOFF, GON or GINIT as an argument, the view parameters remain unaltered and the viewing transformation can be recreated by calling gUpdateView() only.

If one model is drawn which involves modelling transformations, e.g. rotations, a second can be drawn with the same view after a call to gSetTransform(GINIT). gSetTransform(GINIT) resets the transformation matrix but leaves the view parameters intact. The next call to gUpdateView() can then build a view from a known state rather than from an evolved, and therefore possibly unknown, state. e.g:

```
/* Set up a perspective view */
   gDefinePerspView(...);
   gUpdateView();
/* Draw first model */
   model1();
/* Throw this view away */
   gSetTransform(GRESET);
/* Set up parallel view */
   gDefineParallelView(...);
   gUpdateView();
/* Redraw first model with
   the new view */
   model1();
/* Reinitialize transformation
   but leave view parameters
   intact */
   gSetTransform(GINIT);
/* Regenerate view transform */
   gUpdateView();
/* Draw a second model */
   model2();
```

```
!  Set up a perspective view
   call gDefinePerspView(...)
   call gUpdateView
!  Draw first model
   call model1
!  Throw this view away
   call gSetTransform(GRESET)
!  Set up parallel view
   call gDefineParallelView(...)
   call gUpdateView
!  Redraw first model with
!  the new view
   call model1
!  Reinitialize transformation
!  but leave view parameters
!  intact
   call gSetTransform(GINIT)
!  Regenerate view transform
   call gUpdateView
!  Draw a second model
   call model2
```

To reset the viewing parameters but leave the current transformation intact, use gInitView(). This has the converse effect to gSetTransform(GINIT).

gInitView()

Use of Superseded Routine

The user should be warned about successive calls to gGenerateView() without resetting the GINO transformation as explained above. It is stressed that each time gGenerateView() is called, the viewing parameters are added to the CURRENT transformation matrix to create a new matrix; thus a second call to gGenerateView() without resetting the matrix will effectively add the viewing parameters twice, with unpredictable results.

gGenerateView()

Modifying the Drawing

The image of the user model (i.e. that image which is projected onto the screen or paper etc.) may be modified by changing the viewing transformation. The model itself need not be redefined.

Re-specifying the View

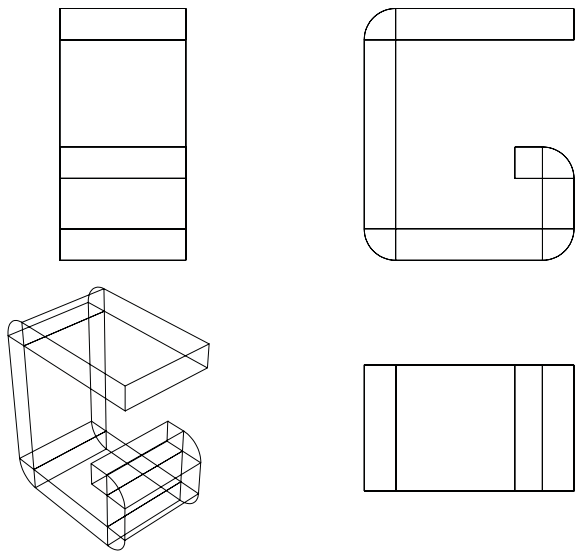
The most obvious means of changing the view parameters is to re-specify the view completely, separating calls to gDefinePerspView(), gDefineSphericalView(), or gDefineParallelView() and gUpdateView() by nullifying calls to gSetTransform(GRESET). The code example for the next figure shows this. However, it is often more economical to adjust a parameter individually. If transforming is switched on, and the user wishes to preserve the current transformation outside of reinitializing the view parameters, then a call to gInitView() would effect this, as opposed to gSetTransform(GRESET) which would discard it.

C code

```
        GLIMIT top_left = {0.0,90.0,70.0,140.0},
               top_right = {90.0,180.0,70.0,140.0},
               bottom_right = {90.0,180.0,0.0,70.0},
               bottom_left = {0.0,90.0,0.0,70.0};

    /*  Top left view */
        gSetWindow2D(&top_left);
        gDefineParallelView(1.0,0.0,0.0,25.0,30.0,15.0);
        gUpdateView();
        biggee(50.0,60.0,30.0);
        gSetTransform(GRESET);
    /*  Top right view */
        gSetWindow2D(&top_right);
        gDefineParallelView(0.0,-1.0,0.0,25.0,30.0,15.0);
        gUpdateView();
        biggee(50.0,60.0,30.0);
        gSetTransform(GRESET);
    /*  Bottom right view */
        gSetWindow2D(&bottom_right);
        gDefineParallelView(0.0,-1.0,0.0,25.0,30.0,15.0);
        gUpdateView();
        biggee(50.0,60.0,30.0);
        gSetTransform(GRESET);
    /*  Bottom left view (perspective) */
        gSetWindow2D(&bottom_left);
        r=sqrt(25.0*25.0+30.0+15.0*15.0);
        gDefineSphericalView(25.0,30.0,15.0,r,
                             -1.0,-1.0,-0.9,200.0);
        gUpdateView();
        biggee(50.0,60.0,30.0);
```

F90 code

```
type (GLIMIT) :: top_left = GLIMIT(0.0,90.0,70.0,140.0), &
                 top_right = GLIMIT(90.0,180.0,70.0,140.0), &
                 bottom_right = GLIMIT(90.0,180.0,0.0,70.0), &
                 bottom_left = GLIMIT(0.0,90.0,0.0,70.0)

!  Top left view
   call gSetWindow2D(top_left)
   call gDefineParallelView(1.0,0.0,0.0,25.0,30.0,15.0)
   call gUpdateView
   call biggee(50.0,60.0,30.0)
   call gSetTransform(GRESET)
!  Top right view
   call gSetWindow2D(top_right)
   call gDefineParallelView(0.0,-1.0,0.0,25.0,30.0,15.0)
   call gUpdateView
   call biggee(50.0,60.0,30.0)
   call gSetTransform(GRESET)
!  Bottom right view
   call gSetWindow2D(bottom_right)
   call gDefineParallelView(0.0,-1.0,0.0,25.0,30.0,15.0)
   call gUpdateView
   call biggee(50.0,60.0,30.0)
   call gSetTransform(GRESET)
!  Bottom left view (perspective)
   call gSetWindow2D(bottom_left)
   r=sqrt(25.0*25.0+30.0+15.0*15.0)
   call gDefineSphericalView(25.0,30.0,15.0,r, &
                             -1.0,-1.0,-0.9,200.0)
   call gUpdateView
   call biggee(50.0,60.0,30.0)
```



Re-specifying Views

Positioning the Image

By default, the image is positioned so that the view centre and the centre of the current window coincide. A call to:

gPosViewCentre(xp, yp)

maps the view centre onto the point (**xp**, **yp**) which is supplied in picture (screen) coordinates. The next figure, and its code, illustrates the use of gPosViewCentre(). Notice that one of the basic routines is called first, followed by any qualifying routines, such as gPosViewCentre(). Finally the transformation is defined by a call to gUpdateView().

Orientation of the Image

The default orientation, which aligns world and picture Y axes, can be changed using routine

gSetViewUpDirection(dx, dy, dz)

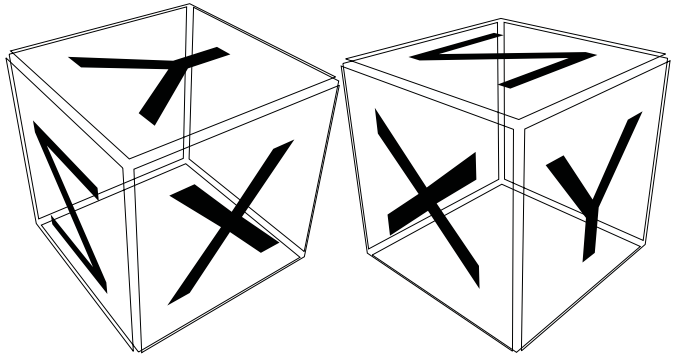
Where (**dx**, **dy**, **dz**) specify a vector direction in the model's world which will be mapped as parallel to the picture Y axis. The figure below shows how gSetViewUpDirection() can be used to change the orientation of a cube. Like gPosViewCentre() it is called after one of the basic routines and before gUpdateView().

C code

```
      GLIMIT window = {0.0,180.0,0.0,140.0};

      gSetWindow2D(&window);
      cside=40.0;
      sside=38.0;
/*  Enable gSetTransform() characters */
      gSetCharTransformMode(GON);
/*  First perspective view */
      gDefineSphericalView(0.0,0.0,0.0,60.0,
                           -1.0,-0.9,-0.7,100.0);
/*  Position using gPosViewCentre() */
      gPosViewCentre(60.0,70.0);
      gUpdateView();
      cube(cside,sside,0.0,0.0,0.0);
      gSetTransform(GRESET);
/*  Respecify view */
      gDefineSphericalView(0.0,0.0,0.0,60.0,
                           -1.0,-0.9,-0.7,100.0);
      gPosViewCentre(120.0,70.0);
/*  Re-orientate view */
      gSetViewUpDirection(0.0,0.0,1.0);
      gUpdateView();
      cube(cside,sside,0.0,0.0,0.0);
```

F90 code

```
type (GLIMIT) :: window = GLIMIT(0.0,180.0,0.0,140.0)

      call gSetWindow2D(window)
      cside=40.0
      sside=38.0
!  Enable gSetTransform() characters
      call gSetCharTransformMode(GON)
!  First perspective view
      call gDefineSphericalView(0.0,0.0,0.0,60.0, &
                          -1.0,-0.9,-0.7,100.0)
!  Position using gPosViewCentre
      call gPosViewCentre(60.0,70.0)
      call gUpdateView
      call cube(cside,sside,0.0,0.0,0.0)
      call gSetTransform(GRESET)
!  Respecify view
      call gDefineSphericalView(0.0,0.0,0.0,60.0, &
                          -1.0,-0.9,-0.7,100.0)
      call gPosViewCentre(120.0,70.0)
!  Re-orientate view
      call gSetViewUpDirection(0.0,0.0,1.0)
      call gUpdateView
      call cube(cside,sside,0.0,0.0,0.0)
```



Changing Position and Orientation of View

Moving Eye, View Plane or both

This section describes three routines:

gSetViewPlaneDistance(d)

gMoveViewCentre(s)

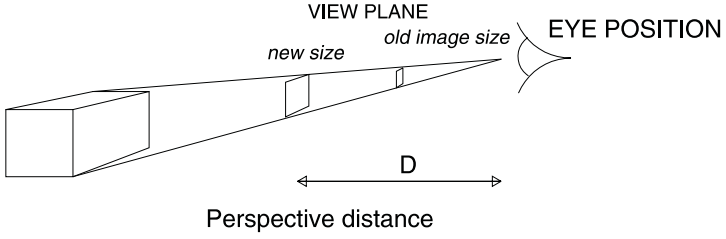
gSetViewEyeDistance(d)

A clear mental picture of the spatial relationship between eye, object and view plane is useful in understanding their effects. A quick glance at the following figure will suggest what is involved. Note the changes in image size with respect to changes in the perspective distance.

A more detailed description of each routine follows.

Zooming

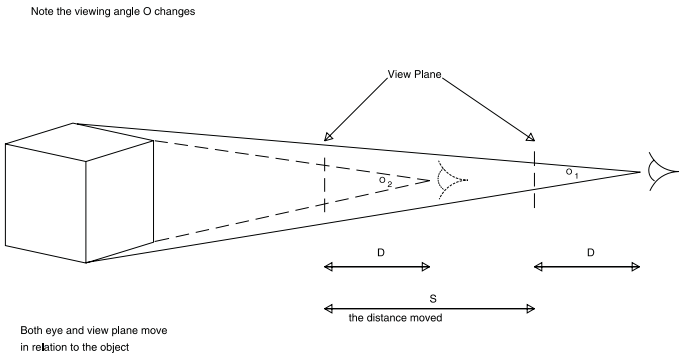
Zooming changes the size of the image without affecting the extent to which it is distorted by perspective. The figure below illustrates what happens; the view plane alone moves and thus the perspective distance changes. Routine gSetViewPlaneDistance() alters this distance to **d**. The size of the image changes proportionally with **d**.



Zooming by Setting View Plane Distance

Moving Eye and View Plane

If both eye and view plane are moved, so as to keep the perspective distance fixed, the image size changes. The effect is as if you are moving in relation to the object and thus the perspective distortion changes due to the alteration of the viewing angle. Routine gMoveViewCentre() is used to achieve this, where **s** is the distance moved along the line of sight.



Effect of Moving View Centre

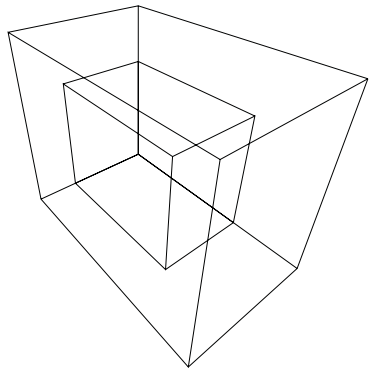
The figure above illustrates this point, and the figure below with its associated code offers an example:

C code

```
/* Set up initial view
   gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,100.0);
   gUpdateView();
/* Draw object */
   cuboid(50.0,40.0,30.0);
   gSetTransform(GRESET);
/* Re-establish view */
   gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,100.0);
/* Move eye and view plane 50mm closer to object */
   gMoveViewCentre(50.0);
   gUpdateView();
/* Redraw */
   cuboid(50.0,40.0,30.0);
```

F90 code

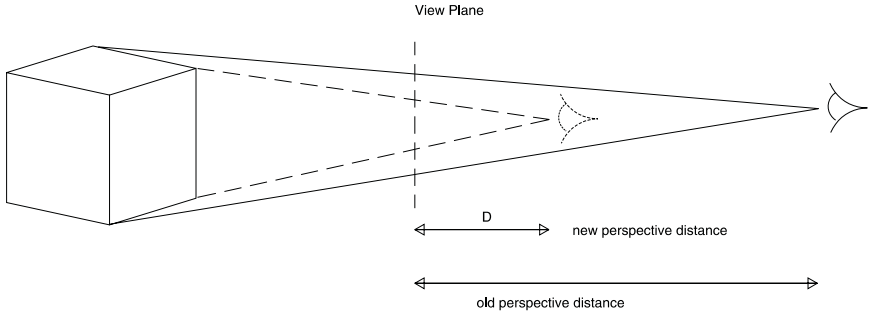
```
! Set up initial view
  call gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,100.0)
  call gUpdateView
! Draw object
  call cuboid(50.0,40.0,30.0)
  call gSetTransform(GRESET)
! Re-establish view
  call gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,100.0)
! Move eye and view plane 50mm closer to object
  call gMoveViewCentre(50.0)
  call gUpdateView
! Redraw */
  call cuboid(50.0,40.0,30.0)
```



Example of Moving View Centre

Moving the Eye Alone

Moving the eye without moving the view plane changes the perspective distance. The figure below illustrates this. The routine used is gSetViewEyeDistance() where **d** is the new perspective distance.



Effect of Setting New Eye Distance

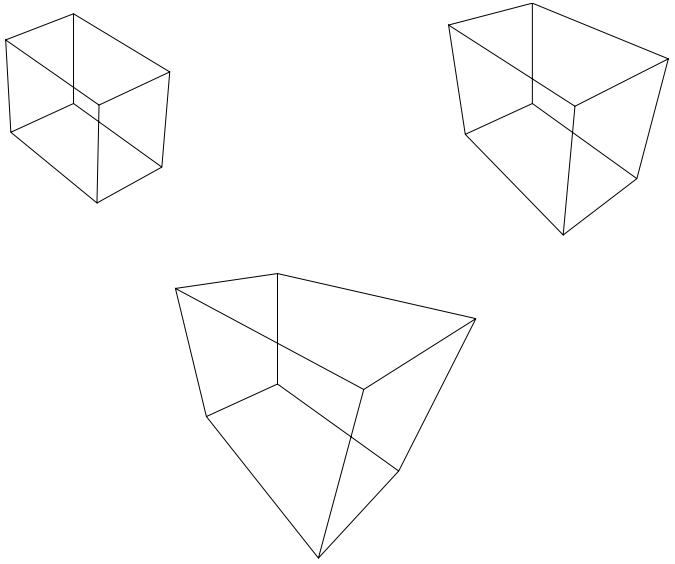
The figure below shows gSetViewEyeDistance() in use. (Note that if a negative value for **d** is supplied the viewing direction is reversed).

C code

```
/* Original view - note perspective distance is
   chosen to position view plane through the
   origin  */
   gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,156.0);
   gPosViewCentre(45.0,105.0);
   gUpdateView();
   cuboid(25.0,20.0,15.0);
   qSetTransform(GRESET);
   gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,156.0);
/* Change perspective distance to 65mm */
   gSetViewEyeDistance(65.0);
   gPosViewCentre(135.0,105.0);
   gUpdateView();
   cuboid(25.0,20.0,15.0);
   gSetTransform(GRESET);
   gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,156.0);
/* Reduce perspective distance to 45mm
   note the view plane is fixed - only the
   eye position can be varied using gSetViewEyeDistance() */
   gSetViewEyeDistance(45.0);
   gPosViewCentre(85.0,50.0);
   gUpdateView();
   cuboid(25.0,20.0,15.0);
```

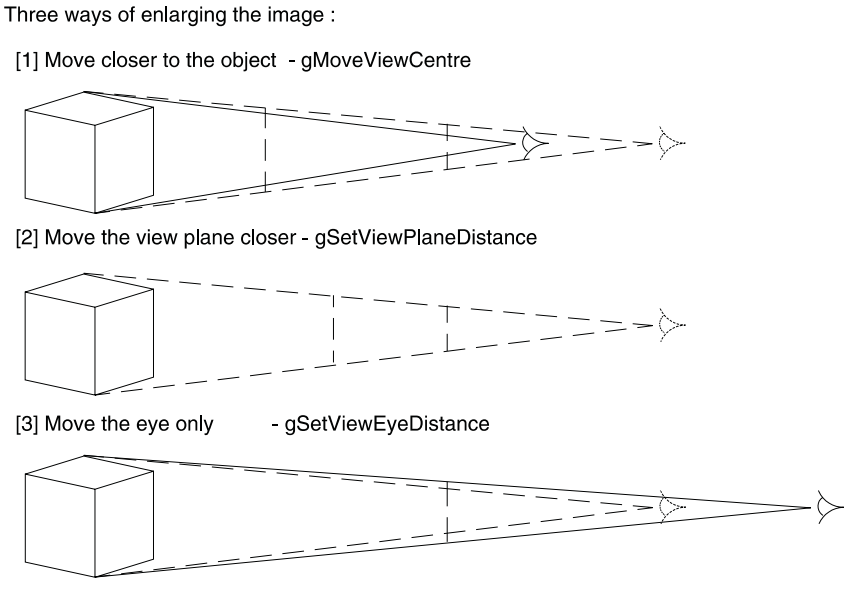
F90 code

```
!  Original view - note perspective distance is
!  chosen to position view plane through the
!  origin
   call gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,156.0)
   call gPosViewCentre(45.0,105.0)
   call gUpdateView
   call cuboid(25.0,20.0,15.0)
   call gSetTransform(GRESET)
   call gDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,156.0)
!  Change perspective distance to 65mm
   call gSetViewEyeDistance(65.0)
   call gPosViewCentre(135.0,105.0)
   call gUpdateView
   call cuboid(25.0,20.0,15.0)
   call gSetTransform(GRESET)
   call qDefinePerspView(100.0,90.0,80.0,-1.0,-0.9,-0.8,156.0)
!  Reduce perspective distance to 45mm
!  note the view plane is fixed - only the
!  eye position can be varied using gSetViewEyeDistance
   call gSetViewEyeDistance(45.0)
   call gPosViewCentre(85.0,50.0)
   call gUpdateView
   call cuboid(25.0,20.0,15.0)
```



Example of Altering Eye Position

The figure below compares the effects of gMoveViewCentre(), gSetViewEyeDistance() and gSetViewPlaneDistance():



Comparison of Altering Eye/Plane positions

The routine gSetViewEyeDistance() is a tricky routine to use because both viewing angle and the distance from the object change, and without some forethought the effects may be surprising. Generally gSetViewPlaneDistance() or gMoveViewCentre() are to be recommended instead, except in two particular circumstances.

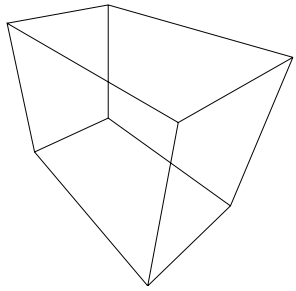
Firstly if the view plane passes through or near the object it is possible to adjust the extent to which a drawing is distorted by perspective. Reducing the perspective distance D increases the distortion. The earlier example shows this.

A second use of gSetViewEyeDistance() is to provide a perspective distance after setting up a parallel view. Remember that gDefineParallelView() takes the view centre and view direction as arguments. If gSetViewEyeDistance() is then called, enough information is available to draw a perspective view. In particular, if the view centre is a point within the object then the view plane will intersect the model and the situation described above arises.

This approach has all the advantages of using gDefineSphericalView() but without being tied to the current window. The figure below illustrates this point, but the user should experiment with this technique.

```
gDefineParallelView(
    -1.0,-0.9,-0.8,0.0,0.0,0.0);
gSetViewEyeDistance(100.0);
gUpdateView();
cuboid(50.0,40.0,30.0);
```

```
call gDefineParallelView( &
    -1.0,-0.9,-0.8,0.0,0.0,0.0)
call gSetViewEyeDistance(100.0)
call gUpdateView
call cuboid(50.0,40.0,30.0)
```



Setting Eye position with Parallel View

Changing the Line of Sight

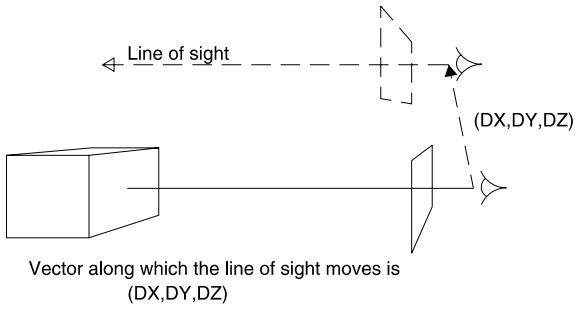
The line of sight can be shifted using gViewShift(), rotated using gViewRotate() or for parallel views gViewTurn() can be used.

gViewTurn(xr, yr, zr, dx, dy, dz, angle)

The routine gViewShift() moves the line of sight incrementally, and thus both view centre and eye position change (see below).

gViewShift(dx, dy, dz)

Generally the effect is as if the model were moved in space along a vector (**-dx**,**-dy**,**-dz**).

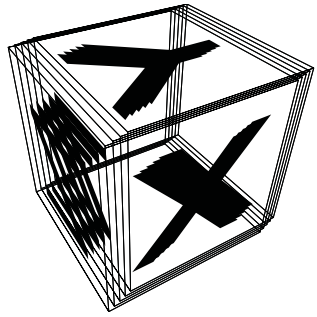


Effect of a View Shift

The figure below illustrates this and demonstrates the usefulness of gViewShift().

```
gSetCharTransformMode(GSOFT);
for ( i=1; i<=5; i++) {
   gSetTransform(GRESET);
   gDefineSphericalView(
         0.0,0.0,0.0,40.0,
        -1.0,-0.9,-0.6,150.0);
   delta = (float) i * 1.5;
   gViewShift(0.3*delta,0.0,
         delta);
   gUpdateView();
   cube(40.0,40.0,0.0,0.0,0.0);
}
```

```
call gSetCharTransformMode(GSOFT)
do i=1,5
   call gSetTransform(GRESET)
   call gDefineSphericalView( &
        0.0,0.0,0.0,40.0, &
       -1.0,-0.9,-0.6,150.0)
   delta = real(i) * 1.5
   call gViewShift(0.3*delta, &
        0.0, delta)
   call gUpdateView
   call cube(40.0,40.0,0.,0.,0.)
end do
```

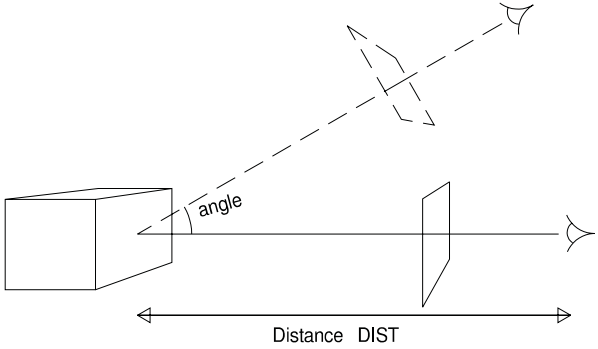


Example of Shifting View

The routine:

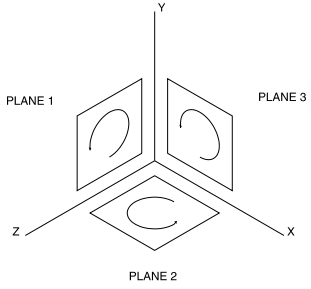
gViewRotate(plane, angle, dist)

allows the line of sight to be rotated such that the **angle** it makes with a **plane** is constant. The rotation is about a point **dist** current units from the eye, along the line of sight. It is often difficult to maintain an accurate mental picture of what is happening with gViewRotate(). The figure immediately below shows the basic idea and the figure following that illustrates the numbering and sign conventions. The axes and planes are right handed. Plane number 1 is orthogonal to the X axis, plane 2 to the Y axis and plane 3 to the Z axis.



Effect of Rotating View

Looking from the positive side of a plane anti-clockwise rotation is positive. (If plane 1 passes through the origin and is viewed from a position 100 units along the X axis, then the view point is on the plane's positive side).

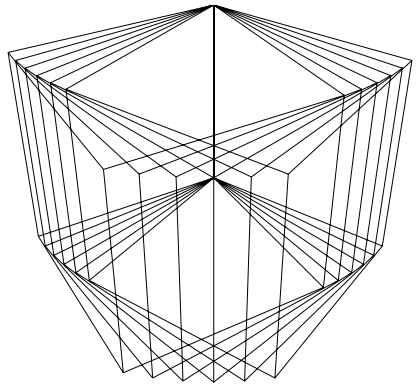


Plane Numbering Convention

The routine gViewRotate() can be very useful. If the view centre is defined to lie within the model and the distance S is set equal to the perspective distance then the effect is as if the model were being rotated in space. The next figure shows this.

```
for (i=1; i<=6; i++) {
   gSetTransform(GRESET);
/* Perspective distance is chosen
   to put view c plane
   through the origin */
   gDefinePerspView(
      100.0,100.0,100.0,
     -1.0,-1.0,-1.0,173.2);
   delta = (float)(i-3)*5.0;
   gViewRotate(2,delta,173.2);
   gUpdateView();
   cuboid(40.0,40.0,40.0);
}
```

```
do i=1,6
  call gSetTransform(GRESET)
! Perspective distance is chosen
! to put view c plane
! through the origin
  call gDefinePerspView( &
       100.0,100.0,100.0, &
      -1.0,-1.0,-1.0,173.2)
  delta = real(i-3)*5.0
  call gViewRotate(2,delta,173.2)
  call gUpdateView
  call cuboid(40.0,40.0,40.0)
end do
```



Example of Rotating View

Note that the distance about which the view is rotated is equal to the distance between the eye and the back edge of the cube.

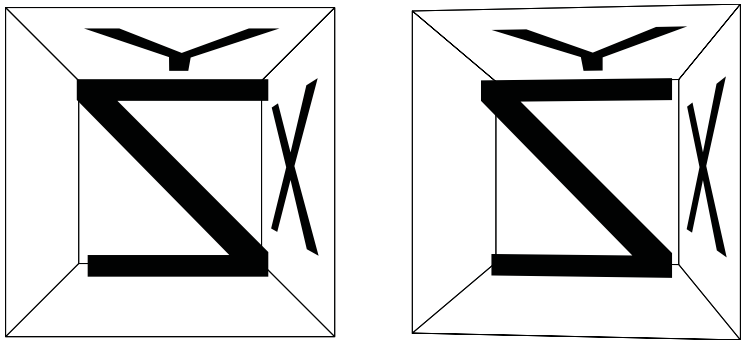
i.e. dist = sqrt(xeye*xeye + yeye*yeye + zeye*zeye)

Other values can be used if the view is required to be rotated about a different point.

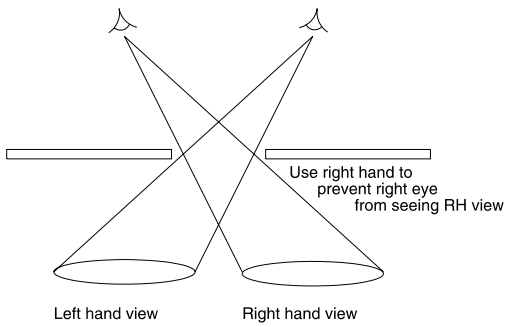
The next figure shows two projections of a lettered cube, one drawn using gViewRotate(). They are a stereoscopic pair and the figure following them shows how to look at them. It may take a while to bring the 3-D image into focus.

```
gSetCharTransformMode(GSOFT);
gDefinePerspView(0.0,0.0,70.0,
      0.0,0.0,-1.0,70.0);
gPosViewCentre(55.0,70.0);
gUpdateView();
cube(40.0,40.0,0.0,0.0,0.0);
gSetTransform(GRESET);
gDefinePerspView(0.0,0.0,70.0,
      0.0,0.0,-1.0,70.0);
gPosViewCentre(125.0,70.0);
gViewRotate(2,3.0,70.0);
gUpdateView();
cube(40.0,40.0,0.0,0.0,0.0);
```

```
call gSetCharTransformMode(GSOFT)
call gDefinePerspView(0.0,0.0, &
     70.0,0.0,0.0,-1.0,70.0)
call gPosViewCentre(55.0,70.0)
call gUpdateView
call cube3(40.0,40.0,0.0,0.0,0.0)
call gSetTransform(GRESET)
call gDefinePerspView(0.0,0.0, &
     70.0,0.0,0.0,-1.0,70.0)
call gPosViewCentre(125.0,70.0)
call gViewRotate(2,3.0,70.0)
call gUpdateView
call cube3(40.0,40.0,0.0,0.0,0.0)
```



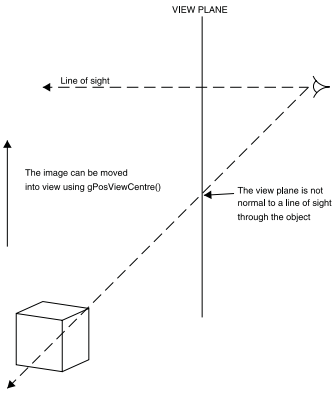
Stereoscopic Pair



How to View Previous Figure

Projections onto an Oblique Plane

By definition, the line of sight is orthogonal to the view plane. Nevertheless projections onto angled planes can be set up. The following figure illustrates how this is done. Although the line of sight does not pass through the object, an image is generated on the view plane which can be brought into the current window using gPosViewCentre().

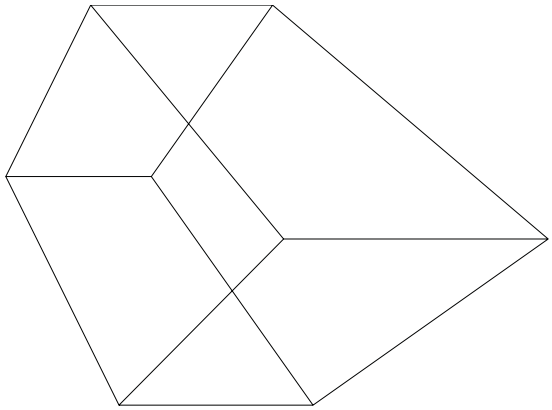


Oblique Viewing

The figure below shows an oblique projection. Notice that gPosViewCentre() positions the view centre, which is defined to be on a line orthogonal to the view plane - some calculation may be necessary to place the drawing exactly where it is required.

```
gDefinePerspView(
    100.0,100.0,100.0,
    -1.0,-1.0,0.0,125.0);
/*  gPosViewCentre() places the
    view centre at the quoted
    point */
gPosViewCentre(0.0,70.0);
gUpdateView();
cuboid(50.0,40.0,30.0);
```

```
call gDefinePerspView( &
     100.0,100.0,100.0, &
    -1.0,-1.0,0.0,125.0)
! gPosViewCentre() places the
! view centre at the
! quoted point
call gPosViewCentre(0.0,70.0)
call gUpdateView
call cuboid(50.0,40.0,30.0)
```



Example of an Oblique View

Saving and Restoring View Parameters

There are two pairs of routines for saving and setting the view parameters. These are:

gGetViewParams(vdata)

gGetViewState(vstate)

gSetViewParams(vdata)

gSetViewState(vstate)

where **vdata** is an array of 15 values and **vstate** is a structure of type GVIEWSTATE. Both contain the same data, but the structure is more accessible through its various elements as described in the reference section.

These routines can be used to extract information about the current view, save and restore a particular view setting, or even modify the current viewing parameters. Note that after calling either of gSetViewParams() or gSetViewState(), the actual values passed are not activated until the current view is updated using gUpdateView().

These routines do not affect the current modelling transformation which can be saved and reset independently of the viewing parameters using gGetTransform3D() and gSetTransform3D(), gSaveTransform() and gRestoreTransform() or gPushTransform() and gPopTransform() (see page 371).

Modifying the View Matrix

The following routine can be used to modify the current view by applying a modification matrix:

gModifyView(a)

where **a** is a 16 element (4x4) real array containing the modification matrix. This routine is useful for advanced applications including the generation of shadows (see page 342).

Listings of the Routines used in this Chapter

C code

```
void cube3(float cside,float sside,
          float x,float y,float z)
{
/* Characters      , y, z,  ,  , x */
   int i,asc[6] = {32,89,90,32,32,88};

   gPushTransform();
   gShift3D(x,y,z);
   for (i=0; i<=3; i++) {
      squar3(sside,cside,asc[i]);
      gRotate3D(GXAXIS,90.0);
   }
   gRotate3D(GYAXIS,90.0);
   squar3(sside,cside,asc[4]);
   gRotate3D(GYAXIS,180.0);
   squar3(sside,cside,asc[5]);
   gPopTransform();
}

void squar3(float a,float cside,int nlettr)
{
   float del;

   del=a/2.0;
   gMoveTo3D(del,del,cside/2.0);
   gDrawLineBy3D(0.0,-a,0.0);
   gDrawLineBy3D(-a,0.0,0.0);
   gDrawLineBy3D(0.0,a,0.0);
   gDrawLineBy3D(a,0.0,0.0);
   gMoveTo3D(-0.6*del,-0.6*del,-cside/2.0);
   gSetCharSize(a*0.82,0.6*a);
   gDisplayAsciiChar(nlettr);
}
```

```
void biggee(float gwide,float ghigh,float gdeep)
/* Draw a 3D letter G which fits in a box
   gwide by ghigh by gdeep */
{
   float thickn,gbarht,gbarwd;

   gSaveTransform();
/* Thickness of bars arbitrary multiple of depth
   thickn=0.25*gdeep;
/* Lower horizontal bar */
   cuboid((gwide-2.0*thickn),thickn,gdeep);
/* Upper horizontal bar */
   gShift2D(0.0,ghigh-thickn);
   cuboid((gwide-thickn),thickn,gdeep);
/* Left hand vertical bar */
   gShift2D(-thickn,0.0);
   gRotate2D(-90.0);
   cuboid((ghigh-2.0*thickn),thickn,gdeep);
   gRestoreTransform();
/* Horizontal insert */
   gbarht=(9.0/20.0)*ghigh;
   gbarwd=0.8*gbarht;
   gShift2D((gwide-2.0*thickn),gbarht-thickn);
   cuboid(-(gbarwd-2.0*thickn),thickn,gdeep);
/* Vertical insert */
   gRotate2D(-90.0);
   cuboid((gbarht-2.0*thickn),thickn,gdeep);
   gRestoreTransform();
/* Bottom left corner */
   gShift2D(-thickn,0.0);
   filler(thickn,gdeep);
/* Top left corner */
   gShift2D(0.0,ghigh);
   gRotate2D(-90.0);
   filler(thickn,gdeep);
/* Bottom right */
   gShift2D(ghigh,gwide);
   gRotate2D(180.0);
   filler(thickn,gdeep);
/* Corner of insert */
   gShift2D(gbarht,0.0);
   gRotate2D(90.0);
   filler(thickn,gdeep);
   gRestoreTransform();
}

void filler(float fwidth,float fdepth)

/*  Draw a quarter cylinder radius fwidth,
    depth fdepth */
{

   gMoveTo3D(0.0,fwidth,fdepth);
   gDrawArcTo2D(fwidth,fwidth,fwidth,0.0,GANTICLOCKWISE);
   gMoveBy3D(0.0,0.0,-fdepth);
   gDrawArcTo2D(fwidth,fwidth,0.0,fwidth,GCLOCKWISE);
}
```

```
void cuboid(float alengt,float height,float depth)


/*  Draw box of dimensions alengt,height,depth in xyz */
{
   gMoveTo3D(alengt,height,depth);
/* Front edges */
   gDrawLineBy3D(-alengt,0.0,0.0);
   gDrawLineBy3D(0.0,-height,0.0);
   gDrawLineBy3D(alengt,0.0,0.0);
   gDrawLineBy3D(0.0,height,0.0);
/* Side edges */
   gDrawLineBy3D(0.0,0.0,-depth);
   gDrawLineBy3D(0.0,-height,0.0);
   gDrawLineBy3D(0.0,0.0,depth);
/* Top edges */
   gMoveTo2D(0.0,height);
   gDrawLineBy3D(0.0,0.0,-depth);
   gDrawLineBy3D(alengt,0.0,0.0);
   gMoveBy3D(-alengt,0.0,0.0);
/* Rear edges */
   gDrawLineBy3D(0.0,-height,0.0);
   gDrawLineBy3D(0.0,0.0,depth);
   gMoveBy3D(0.0,0.0,-depth);
   gDrawLineBy3D(alengt,0.0,0.0);
}

file : subs.h

void cube3(float cside,float sside,float x,float y,float z);
void squar3(float a,float cside,int nlettr);
void biggee(float gwide,float ghigh,float gdeep);
void filler(float fwidth,float fdepth);
void cuboid(float alengt,float height,float depth);
```

F90 code

```
subroutine cube3(cside,sside,x,y,z)
use gino_f90
real cside,sside,x,y,z
! Characters      , y, z,  ,   , x
   integer :: asc(6) = (/ 32,89,90,32,32,88 /)

   call gPushTransform
   call gShift3D(x,y,z)
   do i=1,4
      call squar3(sside,cside,asc(i))
      call gRotate3D(GXAXIS,90.0)
   end do
   call gRotate3D(GYAXIS,90.0)
   call squar3(sside,cside,asc(5))
   call gRotate3D(GYAXIS,180.0)
   call squar3(sside,cside,asc(6))
   call gPopTransform
return
end
```

```
      subroutine squar3(a,cside,nlettr)
      use gino_f90
      real a,cside
      integer nlettr
         real del

         del=a/2.0
         call gMoveTo3D(del,del,cside/2.0)
         call gDrawLineBy3D(0.0,-a,0.0)
         call gDrawLineBy3D(-a,0.0,0.0)
         call gDrawLineBy3D(0.0,a,0.0)
         call gDrawLineBy3D(a,0.0,0.0)
         call gMoveTo3D(-0.6*del,-0.6*del,-cside/2.0)
         call gSetCharSize(a*0.82,0.6*a)
         call gDisplayAsciiChar(nlettr)
      return
      end

      subroutine biggee(gwide,ghight,gdeep)
      use gino_f90
      real gwide,ghigh,gdeep
      !  Draw a 3D letter G which fits in a box
      !  gwide by ghigh by gdeep

         real thickn,gbarht,gbarwd

         call gSaveTransform
      !  Thickness of bars arbitrary multiple of depth
         thickn=0.25*gdeep
      !  Lower horizontal bar
         call cuboid((gwide-2.0*thickn),thickn,gdeep)
      !  Upper horizontal bar
         call gShift2D(0.0,ghigh-thickn)
         call cuboid((gwide-thickn),thickn,gdeep)
      !  Left hand vertical bar
         call gShift2D(-thickn,0.0)
         call gRotate2D(-90.0)
         call cuboid((ghight-2.0*thickn),thickn,gdeep)
         call qRestoreTransform
      !  Horizontal insert
         gbarht=(9.0/20.0)*ghight
         gbarwd=0.8*gbarht
         call gShift2D((gwide-2.0*thickn),gbarht-thickn)
         call cuboid(-(qbarwd-2.0*thickn),thickn,qdeep)
      !  Vertical insert
         call gRotate2D(-90.0)
         call cuboid((gbarht-2.0*thickn),thickn,gdeep)
         call qRestoreTransform
      !  Bottom left corner
         call gShift2D(-thickn,0.0)
         call filler(thickn,qdeep)
      !  Top left corner
         call gShift2D(0.0,ghight)
         call gRotate2D(-90.0)
         call filler(thickn,gdeep)
```

```
!  Bottom right
   call gShift2D(ghight,gwide)
   call gRotate2D(180.0)
   call filler(thickn,gdeep)
!  Corner of insert
   call gShift2D(gbarht,0.0)
   call gRotate2D(90.0)
   call filler(thickn,gdeep)
   call gRestoreTransform
return
end

subroutine filler(fwidth,fdepth)
use gino_f90
real fwidth,fdepth
!  Draw a quarter cylinder radius fwidth,
!  depth fdepth

   call gMoveTo3D(0.0,fwidth,fdepth)
   call gDrawArcTo2D(fwidth,fwidth,fwidth,0.0,GANTICLOCKWISE)
   call gMoveBy3D(0.0,0.0,-fdepth)
   call gDrawArcTo2D(fwidth,fwidth,0.0,fwidth,GCLOCKWISE)
return
end

subroutine cuboid(alengt,height,depth)
use gino_f90
real alengt,height,depth
!  Draw box of dimensions alengt,height,depth in xyz

   call gMoveTo3D(alengt,height,depth)
!  Front edges
   call gDrawLineBy3D(-alengt,0.0,0.0)
   call gDrawLineBy3D(0.0,-height,0.0)
   call gDrawLineBy3D(alengt,0.0,0.0)
   call gDrawLineBy3D(0.0,height,0.0)
!  Side edges
   call gDrawLineBy3D(0.0,0.0,-depth)
   call gDrawLineBy3D(0.0,-height,0.0)
   call gDrawLineBy3D(0.0,0.0,depth)
!  Top edges */
   call gMoveTo2D(0.0,height)
   call gDrawLineBy3D(0.0,0.0,-depth)
   call gDrawLineBy3D(alengt,0.0,0.0)
   call gMoveBy3D(-alengt,0.0,0.0)
!  Rear edges
   call gDrawLineBy3D(0.0,-height,0.0)
   call gDrawLineBy3D(0.0,0.0,depth)
   call gMoveBy3D(0.0,0.0,-depth)
   call gDrawLineBy3D(alengt,0.0,0.0)
return
end
```

Chapter 24

PICTURE SEGMENTS

Picture Segments Introduction

Picture Segments provide a means of 'labelling' parts of a picture as well as building structures that can be manipulated in their entirety without redrawing the elements that make them up. Their use is very varied, but the following types of application can take advantage of segment facilities:

- Creating designs from components
- Component or menu option selection
- 3D hierarchical structure creation and manipulation
- Hardcopy of drawing to printer or plotter
- Progressive interactive design package

GINO provides the following segment facilities:

- Hardware and/or software segment operation
- Segment creation, extension, deletion, rename
- Visibility, Sensitivity, Highlighting attributes
- Segment transformation at device level
- Segment structures; copies, references and groups
- Editing of modelling transformations within segments
- Light pen simulation
- Archiving and restoring segment store

In the development of graphics devices, segment facilities were not initially provided in the terminal as it required an on-board processor and memory to control and store the information. Later on, some devices did add these facilities and GINO was one of the first packages to provide access through its segment routines. With the development of workstations, again different models exist with and without segment facilities. Some segment facilities are included with the GINO OpenGL drivers (WOGL and GLX) but note that these drivers do not provide all the facilities documented below (see page 271 and Appendix B).

GINO provides the facility to use segments whether or not the device has its own hardware segment facilities by providing a store of segment information which is maintained on the host computer. (In the case of workstations the host is the workstation itself). This file is called the Software Display File or SDF and it can be held in memory or on disk at the discretion of the user.

In spite of GINO's philosophy of emulating hardware facilities in software, the complexity of segments and the wide variation of hardware facilities that are available, means that a slightly different approach has been adopted in this area. Where an application is run on a device without segment facilities, a warning message will be output when the first segment is opened, and GINO will open a disk based Software Display File to handle all segment facilities in software. However, it is advisable, where an application uses segments to study the following sections carefully and specifically set up segment handling as required by the application. This will remove the warning message being generated.

The main routine used to control the segment handling mode is:

gSetSegMode(sw)

sw=GHARDWARE (default) disables software emulation and segment handling will be done by the hardware only, if the device has segment facilities. The user should check in Appendix B under the required device to see if such facilities exist.

sw=GMIXWARE switches on software emulation and a display file is maintained by GINO. GINO will attempt to use hardware facilities wherever possible to ensure optimum speed of operation, however, all segment information is stored in the display file. Where a particular segment facility is not available in the hardware, the software display file will be used.

sw=GSOFTWARE switches on software emulation and ignores any hardware segment facilities.

It should be noted that hardware segment facilities may differ from those provided through GINO's software display file. This is largely due to the evolving nature and complexity of segment facilities and where no standard hardware method exists.

In order to cater for the widest possible scope of segment facilities, GINO has adopted the highest level of sophistication with regard to its display file by storing untransformed, unclipped 3D coordinates together with user changes to modelling transformations. However, most of the graphics devices that provide hardware segment facilities store only 2D picture coordinates after transformations and clipping to device limits. In addition, some segment facilities listed above such as references and editing may not be provided in hardware.

Therefore, when developing an application using segments the following suggestions should be taken into consideration:

- If hardware segments are available, check which facilities are actually provided against those required by the application. If all the required facilities are provided on all the devices on which the application is to be run then the most efficient operation will be provided using **sw**=GHARDWARE (the default).
- If the application is to be used on a variety of devices, some with hardware segment facilities and some without, or the application requires some facilities not provided by the hardware, it is advisable to set **sw**=GMIXWARE. The setting of **sw** can, of course, be made dependent on the device selected.
- Where segment facilities provided by hardware are inconsistent with those provided by GINO, for example where the application requires segment structures defined in a 3D coordinate system and the hardware segment facilities are only at the 2D level, it is necessary to use **sw**=GSOFTWARE.
- If segment information is required to be carried over to a secondary output device for hard copy purposes or required for archiving and restoring, **sw** should normally be set to GSOFTWARE to ensure a copy of the information is held by GINO and any hardware segment facilities are not used.

Software Display File Storage

The software display file (SDF) maintained by GINO can be held in program memory or on a direct-access scratch file according to available resources. The routine to declare a workspace area in memory is gDefineSegWorkspace(), which is described below. If no such area is declared, GINO will open and use a disk file when the first segment is opened. The amount of space required for such a display file is very difficult to establish but it can very easily be extensive.

gDefineSegWorkspace(nw)

The routine gDefineSegWorkspace() declares an area within the total workspace area and therefore must be preceded by a call to gSetWorkspaceLimit() (see page 33). The total workspace area must obviously include at least sufficient space for the requirements of gDefineSegWorkspace.

```
gOpenGino();
gSetWorkspaceLimit(10000);
gDefineSegWorkspace(10000);
```

```
call gOpenGino
call gSetWorkspaceLimit(1,10000)
call gDefineSegWorkspace(10000)
```

Note that the workspace is freed with the routine gCloseGino(). The routine gEnqSegWorkspace() can be used to enquire how much space has been allocated for segment storage within the total workspace area and how much free space is left.

gEnqSegWorkspace(nw, nfree)

Segment Building

Picture segments are opened and closed using the following routines:

gOpenSeg(nseg)

gCloseSeg()

Picture segments are created by calls to gOpenSeg() with different values of **nseg**. Segment names are defined by the positive integer number **nseg** between the range 1 and 32767. All drawing between a gOpenSeg()/gCloseSeg() pair is included within the named picture segment. If gOpenSeg() is called from within a picture segment, this first segment is automatically closed by GINO with a call to gCloseSeg(). If gOpenSeg() is called with an existing segment number, the segment is first deleted before a new one is opened.

If any drawing routines are called outside a picture segment, GINO extends segment 0 (the same as gOpenSeg(0)). Segment 0 is a dustbin segment for everything that is drawn and not sent to a specific segment. It cannot be manipulated, redrawn or retained in any way.

The routine gExtendSeg() reopens **nseg** so that more drawing can be added to it.

gExtendSeg(nseg)

The routine gEnqOpenSeg() can be used to enquire which segment number is currently opened for drawing. This may return zero indicating that the dustbin segment is open.

gEnqOpenSeg(nseg)

A segment can be renamed using the routine gRenameSeg(), where the segment **nseg** becomes **newseg**. If a segment called **newseg** already exists, it is deleted from the display file before **nseg** is renamed.

gRenameSeg(nseg, newseg)

Segments can be removed from the display file and screen using gDeleteSeg(). Where software segments are being used, segments are removed from the screen by re-drawing them in the current background colour. This may leave holes in other segments which may need to be repaired by the application at a suitable time as described in ‘Segment Redrawing and Repairing’.

gDeleteSeg(nseg)

Note that while the routine gNewDrawing() may clear the screen of segments as part of its operation, segments are not actually deleted from any software or hardware display file by this routine, but are simply marked as invisible. If all current segments are required to be deleted, gDeleteSeg(GALL) should be called.

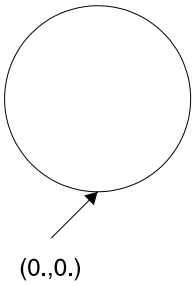
Segment Anchor

The segment anchor is a reference point from which all segment elements can be seen as relative to. It is important to set the correct anchor point when segment transformations are required (see page 431). If the first drawing statement within a segment is visible, the segment anchor will be at the pen position when gOpenSeg() is called;

For example:

```
/* DRAW CIRCLE WITH ANCHOR AT
   (0.0,0.0) */
xanc=0.0;
yanc=0.0;
rad=20.0;
gMoveTo2D(xanc,yanc);
gOpenSeg(100);
gDrawArcBy2D(0.0,rad,0.0,0.0,0);
gCloseSeg();
```

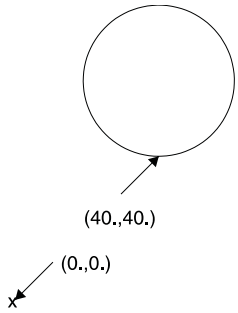
```
! DRAW CIRCLE WITH ANCHOR AT
! (0.0,0.0)
xanc=0.0
yanc=0.0
rad=20.0
call gMoveTo2D(xanc,yanc)
call gOpenSeg(100)
call gDrawArcBy2D(0.,rad,0.,0.,0)
call gCloseSeg
```



Otherwise, the segment anchor is fixed to the end of the first invisible vector within the picture segment:

```
/* DRAW CIRCLE WITH ANCHOR AT
   (40.0,40.0) */
xanc=40.0;
yanc=40.0;
rad=20.0;
gMoveTo2D(0.0,0.0);
gOpenSeg(100);
gMoveTo2D(xanc,yanc);
gDrawArcBy2D(0.0,rad,0.0,0.0,0);
gCloseSeg();
```

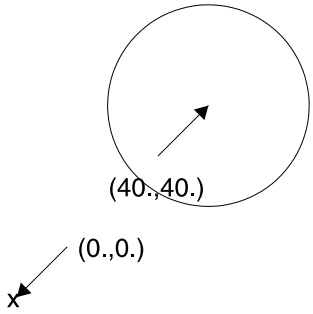
```
! DRAW CIRCLE WITH ANCHOR AT
! (40.0,40.0)
xanc=40.0
yanc=40.0
rad=20.0
call gMoveTo2D(0.0,0.0)
call gOpenSeg(100)
call gMoveTo2D(xanc,yanc)
call gDrawArcBy2D(0.,rad,0.,0.,0)
call gCloseSeg
```



An additional move can be included in order to start the visible part of the segment away from the anchor.

```
/* DRAW CIRCLE WITH ANCHOR AT
   CENTRE OF CIRCLE */
xanc=40.0;
yanc=40.0;
rad=20.0;
gMoveTo2D(0.0,0.0);
gOpenSeg(100);
gMoveTo2D(xanc,yanc);
gMoveBy2D(0.0,-rad);
gDrawArcBy2D(0.0,rad,0.0,0.0,0);
gCloseSeg();
```

```
!  DRAW CIRCLE WITH ANCHOR AT
!  CENTRE OF CIRCLE
xanc=40.0
yanc=40.0
rad=20.0
call gMoveTo2D(0.0,0.0)
call gOpenSeg(100)
call gMoveTo2D(xanc,yanc)
call gMoveBy2D(0.0,-rad)
call gDrawArcBy2D(0.,rad,0.,0.,0)
call gCloseSeg
```



Picture Segment Body

Picture segments contain all drawing elements that are generated between calls to gOpenSeg() (or gExtendSeg()) and gCloseSeg(). These include all lines, arcs, characters, polygons and filling elements as well as the selection of attributes associated with these elements. Changes to modelling transformations are also stored.

The picture segment does not include changes to line, filling and colour tables or windowing and masking information.

If polygon definition and picture segments are used together, the user should remember that a polygon definition is fixed in picture space, whereas picture segments, given suitable output hardware (see page 447), may be dragged. The dragging operation does not modify the definition. Thus subsequent output based on the polygon, e.g. boundary drawing or area fill, will be drawn where it was originally defined in picture space.

In addition, in order to ensure that a polygon definition does not cross a picture segment boundary, gCloseSeg() forces an internal call to gEndPolygon(). This closes any polygon definition that may be open.

Segment Manipulation

Picture segments have a number of attributes. When first created a segment is visible, but not sensitive or highlighted (flashing). Any of the following picture segment attributes can be altered by using the routines below.

When a change is made to a segment using the routines in this section its effect is immediate. If the software emulation of segments is active and the device is unable to effect the change, GINO will update the display possibly by erasing the segment and redrawing it according to its new attribute settings.

Visibility:

gSetSegVis(nseg, vis)

Visibility can be switched off (**vis**=GINVISIBLE) or on (**vis**=GVISIBLE), without deleting the segment from the display file. The software emulation of this action is to redraw the segment in colour zero (background colour) to make the segment invisible and in its correct colours when making it visible. Making a segment invisible may leave holes in underlying graphics which can be repaired using the routine gDrawSeg().

Sensitivity:

gSetSegHit(nseg, sens)

Hit-sensitivity is the capacity of a segment to be detected in any search of the display file. Such searches are carried out by gEnqSegHit() (see page 439) or events utilizing light-pens or cursor selection (see page 447). As the default setting of this attribute is for a segment not to be sensitive, the user must make the segments that are to be detected sensitive with this routine.

Highlighting:

gMarkSeg(nseg, mark)

Highlighting of a segment can be switched on or off (**mark**=GUNMARK or GMARK). The highlighting of a segment can take the form of flashing between two colours, increasing the colour intensity or other means.

The software emulation of this action is to draw the whole segment using a single colour index. By default this is the highest colour index available on the current device but can be changed by calling:

gSetSegMarkColour(col)

The actual colour used can be set using gDefineRGB() etc. (see page 205). When highlighting is switched off the segment is redrawn in its correct colours.

Picture Segment Transformations

Picture segments also contain a segment transformation matrix as part of their attributes. By default this contains a shift to the segment anchor and default scale and rotation values of 1.0 and 0.0 respectively.

The following routines can be used to alter the segment transformation:

gMoveSegTo2D(nseg, x, y)

gMoveSegBy2D(nseg, dx, dy)

gSetSegTransform(nseg, xsca, ysca, ang, xpos, ypos)

gSetSegTransform2D(nseg, a)

The routines gMoveSegTo2D() and gMoveSegBy2D() re-position the segment anchor, absolutely (x,y) or relatively (dx,dy).

The routines gSetSegTransform() and gSetSegTransform2D() change the complete segment transformation. In the case of gSetSegTransform(), the changes are passed as separate elements, the scale and rotation are applied about the anchor position and the anchor is then re-positioned at **xpos**, **ypos**. For gSetSegTransform2D(), the change is passed in a 3x2 transformation matrix which contains the required scale, rotation and shift values with respect to the original position of the segment. Thus a unit matrix will reset the segment to be displayed at the original position it was created. The routines gBuildMatrix2D() and gCombineMatrix2D() can be used to build or compose a suitable matrix for gSetSegTransform2D() (see page 371).

All changes to a segment transformation are in picture coordinates and affect the representation of the segment on the display screen, ie. after any modelling or viewing transformation has taken place.

Segment Enquiry

The attributes of a segment may be enquired if SDF is being used or the hardware is able to make such an enquiry. If SDF is able to provide information that the hardware is unable to provide, it will add information in order to provide the settings of all the segment attributes.

gEnqSegAttribs(nseg, att)

where **nseg** is the segment's name and the structure **att** will contain the segment attributes. If **att.exist**=0 the segment has not been found in the display file or no information can be returned. If **att.exist**=1 then the segment exists and if **att.exist**=2, it exists and it is a member of a segment group (see page 437). **att.vis**, **att.sens** and **att.mark** return the current setting for visibility, hit-sensitivity and highlighting. **att.anchor** is itself a structure of type GPOINT3 containing the x,y and z coordinates of the segment anchor in picture space (**att.anchor.z** will always = 0.0).

The picture segment transformation can be enquired either separately or as a matrix using the following routines:

gEnqSegTransform(nseg, xsca, ysca, ang, anchor)

gEnqSegTransform2D(nseg, a)

The routines gEnqSegTransform() and gEnqSegTransform2D() complement gSetSegTransform() and gSetSegTransform2D() respectively and if hardware segments are being used the correct enquiry routine should be used. If SDF is active it is always possible to enquire the complete matrix using gEnqSegTransform2D() but gEnqSegTransform() can only be used if gSetSegTransform() has been called.

Segment Redrawing and Repairing

All changes made to the state of a picture segment are carried out immediately on request. Where hardware segments are being used, this will result in a visible change on the display screen, for example in the form of the removal or re-positioning of a segment. Where software segments are being used, most changes of a segment state will require the deletion of the segment in its old state and re-display in its new state. This is carried out by drawing the segment in the current background colour (using the GINO state when the segment was created), thus removing it from the screen, and re-displaying the segment in the new state.

These operations may leave gaps in other segments, particularly where solid areas of colour are present. GINO provides two solutions to this problem:

Firstly, one or more segments may be re-drawn in their current state by using the routine:

gDrawSeg(nseg)

where **nseg** is the segment to be redrawn. In fact, the entire display file may be ‘refreshed’ by calling gDrawSeg(GALL).

Secondly, segments may be defined in XOR mode. Under these circumstances, GINO does not remove the segment by drawing it in the current background colour, but by drawing it in its own colour (thus removing it from the display). Note that the use of XOR mode may result in the segment being displayed in different colours over different colour backgrounds. A segment may be defined in XOR mode by calling gSetPenType(GXOR) immediately after opening the segment.

eg.

```
gOpenSeg(10);
gSetPenType(GXOR);
gSetLineColour(GWHITE);
```

```
call gOpenSeg(10)
call gSetPenType(GXOR)
call gSetLineColour(GWHITE)
```

Segments may also need to be redrawn after any modification to the environment in which the segment may exist and which it is subject to. Such examples are where line or hatch attribute tables have been changed or modelling and/or viewing transformations have been altered.

Segment Structures

More sophisticated segment structures may be built by combining drawing elements from more than one segment into a single segment. This can take place in one of three forms: Copying, Hierarchical Structures and Segment Groups.

Copying

The complete contents of one segment may be copied into the current segment with the routine:

gCopySeg(nseg, pos)

This routine appends a copy of the specified segment into the current segment, and can therefore be used for duplicating or for merging a number of segments together. The copied segment may either be positioned at the current pen position or at the original position of the copied segment depending on the value of **pos**.

A copied segment will always appear in the same form as it was defined as all the information for the segment is inserted, element by element into the current segment. After copying has taken place the current segment is usually redrawn to reflect its new contents.

Hierarchical Segment Structures

A segment hierarchy is useful when an object being drawn also has a hierarchical structure or different components are repeated and so may be referred to more than once but need only defining once. There are however, two forms of hierarchical segment structure that exist and it is important to be aware which is being used as this will affect the methodology of constructing such hierarchical structures and output obtained from using them.

Firstly, there is the stack based system where referencing a segment is like calling a routine such that the graphical state is stored whilst the segment is being drawn, and restored at the end of this process. Alternatively there is the list based system where no such stack is maintained. In both cases the current graphical state will affect a referenced segment (line colour etc.), but in the first, the graphical state is restored at the end of the reference, whereas in the second, the graphical state at the end of the referenced segment is carried back into the calling segment.

GINO's SDF uses the stack based system which is in turn based on the PHIGS model, but OpenGL (and hence GINO's WOGL driver) uses the second model.

In either case, hierarchical segment structures are built using the routine:

gInsertSegRef(nseg)

where a reference to **nseg** is placed in the current segment using this routine. The segment **nseg** is not copied but a pointer to it is maintained in the display file.

In most systems, the segment **nseg** need not exist at the time the reference is made or it may be replaced or extended at any time in the future. If such changes are made to referenced segments it is necessary to redraw the parent segment (with gDrawSeg()) to bring the structure up-to-date. SDF permits a structure of up to 10 depths of references. This is not true of OpenGL where the referenced segment MUST exist before referencing it.

As stated above, the referenced segment always inherits segment attributes, drawing attributes and modelling transformations from its parent, therefore its form and orientation can be altered for each reference made to it. If however, the referenced segment contains changes to the current output state, on a stack based system these are activated but only until the end of the reference when the state is returned to that at the start of the reference, whereas on a list based system these are carried back into the parent.

Use of Modelling Transformations within Segments

As well as inheriting drawing attributes, a referenced segment also inherits the current modelling transformation from its parent. This allows a single referenced segment to appear in different positions and orientations within one structure.

For example:

```
gOpenSeg(10);
gInsertSegRef(2);
gShift2D(100.0,0.0);
gInsertSegRef(2);
gCloseSeg();
```

```
call gOpenSeg(10)
call gInsertSegRef(2)
call gShift2D(100.0,0.0)
call gInsertSegRef(2)
call gCloseSeg
```

This will position segment 2 at two different positions as part of segment 10. Whenever a 2D or 3D transformation routine or transformation control routine (see page 371) is called, which changes the current transformation, a complete matrix is stored in the body of the segment and its application to future drawing elements is subject to the current transformation mode. The same principles of altering the current state and restoring the state on exit apply to modelling transformations as for drawing attributes.

GINO also provides a facility to modify the copy of the transformation matrix stored within a segment allowing local segment editing. This is achieved by placing a TAG in the segment prior to the transformation and using the segment editing routines to change the transformation matrix. Note that although TAGs can be placed anywhere within a segment, only modelling transformation matrices can be edited within the body of a segment.

gInsertSegTag(tag)

gEditSeg2D(nseg, tag, t, swi)

gEditSeg3D(nseg, tag, t, swi)

The routine gInsertSegTag() places a tag element within the current segment, with the user defining the meaning of the tag value. The routines gEditSeg2D() and gEditSeg3D() replace the modelling transformation within **nseg** which is positioned immediately after the tag named **tag**. The structure **t** contains the required transformation matrix and **swi** determines if the matrix is to be applied according to the current transformation mode (see page 371) or replace the current modelling transformation when the segment is redrawn.

For example, if we add a tag just before the change of transformation in the previous example, it would be possible to replace the SHIFT by some other transformation using gEditSeg2D():

```
GMAT2D t;
/* Create segment with two
   references */
   itag1=1;
   gOpenSeg(10);
   gInsertSegRef(2);
   gInsertSegTag(itag1);
   gShift2D(100.0,0.0);
   gInsertSegRef(2);
   gCloseSeg();
/* Generate new matrix and edit
   segment */
   gCombineMatrix2D(0.0,0.0,
      50.0,0.0,0.0,2.0,2.0,&t);
   gSetSegVis(10,GINVISIBLE);
   gEditSeg2D(10,itag1,&t,0);
   gSetSegVis(10,GVISIBLE);
```

```
real t(16) t
! Create segment with two
! references
   itag1=1
   call gOpenSeg(10)
   call gInsertSegRef(2)
   call gInsertSegTag(itag1)
   call gShift2D(100.0,0.0)
   call gInsertSegRef(2)
   call gCloseSeg
!  Generate new matrix and edit
!  segment
   call gCombineMatrix2D(0.0,&
     0.0,50.0,0.0,0.0,2.0,2.0,t)
   call gSetSegVis(10,GINVISIBLE)
   call gEditSeg2D(10,itag1,t,0)
   call gSetSegVis(10,GVISIBLE)
```

Here the shift in ‘x’ of 100.0 is replaced by a matrix containing a shift of 50.0 in ‘x’ and a scale of 2.0 in both ‘x’ and ‘y’. The segment has to be redrawn to see the effect of the change of transformation and this is done with the use of gSetSegVis() making the segment invisible, and then visible after the new transformation has taken place.

Segment Groups

In addition to segment hierarchies, GINO provides the means to manipulate groups of segments in a similar way to a single segment. This facility is useful for sets of discrete items such as menu items which need individual identifiers as well as being manipulated as a group. Segment groups are also identified by a positive integer number but this number should not conflict with existing individual segment numbers.

Segment groups are created using the routine:

gDefineSegGroup(ngrp, nsega, nsegb)

The group of segments, the name of which is **ngrp**, will consist of all the segments between **nsega** and **nsegb** inclusive. (Note that **nsegb** must be greater than **nsega**).

Segments can be ‘degrouped’ using the routine:

gRemoveSegGroup(ngrp)

The group name can be any valid segment number within the permissible range. The default range is 1-32767 but this can be restricted by calling the routine:

gDefineGroupRange(ngrpa, ngrpb)

Following calls to this routine, segment groups cannot be created with names outside the range **ngrpa**-**ngrpb**.

At any stage in the program, the range of the segments which constitute a particular group can be obtained using the routine:

gEnqSegGroup(ngrp, nsega, nsegb)

Once a group has been created it can be referenced by the segment manipulation routines described in 'Segment Manipulation'. When a call to one of these routines is executed, GINO checks the segment number for a group name. If a group exists with that name, the routine will be carried out for the group, otherwise the routine will be executed for a single segment.

For example:

Set range of group	gDefineGroupRange(100,199)
Create group number 103 consisting of segments 20-29	gDefineSegGroup(103,20,29)
Make segment 25 invisible	gSetSegVis(25,GINVISIBLE)
Make all segments in group 103 hit-sensitive	gSetSegHit(103,GSENSITIVE)
Make single segment 104 hit-sensitive	gSetSegHit(104,GSENSITIVE)
Error -out of range - no group created (206 subsequently treated as single segment)	gDefineSegGroup(206,30,39)

It should be noted that in addition to setting the range of segment group numbers, the routine gDefineGroupRange() resets all the group information and 'degroups' any existing groups.

Note: The segment group table is maintained entirely by GINO and not through any hardware segment facilities. The maximum number of segment groups that can be defined at any one time is 50.

Implicit Segment Groups

By using negative values for the picture segment number, manipulation routines can be carried out on implicit groups defined by 'all but' the segment number. If the segment number refers to a group, then the routine acts upon all segments outside that group. If the segment number refers to a single segment the routine acts on all segments except that specified.

For example:

Create group	gDefineSegGroup(200,64,91)
Make group invisible	gSetSegVis(200,GINVISIBLE)
Make segment 84 visible	gSetSegVis(84,GVISIBLE)
Make all segments outside range 64-91 hit-sensitive	gSetSegHit(-200,GSENSITIVE)

For individual segments:

Make all segments except 84 marked	gMarkSeg(-84,GMARK)
Delete all segments except 84	gDeleteSeg(-84)
Delete all segments	gDeleteSeg(-1)

There is an exception to the last example where the argument (-1) is equivalent to GALL which means **all** segments. If picture segment 1 exists and any of the segment routines mentioned in 'Segment Manipulation' (including gDeleteSeg()) are called with the argument (GALL), then the action upon segment 1 will include segment 1.

Light Pen Simulation

The following routine may be used to simulate a light pen hit on a hit-sensitive picture segment.

gEnqSegHit(nseg, x, y, radius)

returns in **nseg** the hit-sensitive picture segment identifier closest to the hit centre (**x**, **y**) within the radius of the hit area **radius**. If none are found **nseg** is set to -1.

When using hardware segment facilities, the action of gEnqSegHit() will depend on the device being used and users should refer to the Appendix B document for that device.

When software segment facilities are being used, gEnqSegHit() will search the entire display file searching for the closest output primitive within the specified tolerance and return the segment number in which it resides.

Dragging

The user program can set a suitable terminal to dragging mode by using the routine:

gDragSeg(nseg)

The terminal will allow segment **nseg** to be dragged (i.e. follow the cursor) until a key is pressed. Information is returned by gGetEventRecord() (see page 447). **event.key** is set to the integer identifier of the key, as in gWaitForEvent().

event.pos is set to the picture coordinates of the anchor of the segment being dragged. All segments except segment 0 can be dragged. In general, dragging would follow an event which returned a segment number.

Software Display Files Across Devices

Normally, when a drawing is created on a graphics device and a copy is required on a printer or plotter for hard copy purposes, it is necessary to re-create the complete drawing from the user's application. However, if drawings are stored in the Software Display File by placing them in one or more segments, the graphics information can be carried over to another device because the Software Display File is NOT deleted on a call to gCloseDevice() when the current device is closed down. (The segment group table is also maintained across device nominations.) This facility can also be taken advantage of, by composing picture segments while the gDummy() device is active, ie. their generation cannot be seen, and utilizing them when required on the required output device.

The following points should be noted when carrying over segment information from one device to another:

- When generating the initial segments, the value of **sw** in the call to gSetSegMode() must be set to GMIXWARE or GSOFTWARE to ensure a copy of segment information is stored in GINO's Software Display File (see page 423). Although **sw**=GMIXWARE can be used in these circumstances, it is advisable to use **sw**=GSOFTWARE to ensure that hardware segment facilities are not used. The reason for this is that any hardware segment store may get out of step with GINO's Software Display File with unpredictable results. (**sw**=GMIXWARE can be used where the application generates segments on an interactive device and limits itself to segment drawing on a secondary output device which does not have segment facilities).
- Remember that all information outside calls to gOpenSeg()/gExtendSeg() and gCloseSeg() will NOT be stored, and therefore will not be available across devices.
- Changes to colour, line and hatch tables are not stored in SDF. This means that they must be set for each output device as required. (This allows the possibility of different colour and broken line settings for different devices taking advantage of various hardware facilities, without affecting the segment information itself).
- Changes to window and mask settings are not stored in SDF.
- The routine gDefinePictureUnits() and Viewports do not affect the coordinate values stored by SDF so if either are being used, remember to call the same routine again for the second device.
- If relying on defaults such as colour and line thickness in the creation part of the program, SDF will interrogate the current device and insert the relevant values for that device into the storage area. If initially drawing to a screen, this would result in colour 10 and line thickness of approx. 0.3mm being stored. When replaying to a printer such as Postscript, this will result in vectors being invisible (white) and possibly too thick. To avoid this problem, try to set known defaults or values in the creation program such as gSetLineColour(GBLACK) and gSetLineWidth(0.0).

The Software display file can be carried across from one device to another as many times as required and new segments may be added or existing ones modified throughout this process. The segment store (and group table) is however, initialized when the routine gOpenGino() is called and is therefore only available between calls to gOpenGino() and gCloseGino().

The following example illustrates an automatic hard-copy generation of a picture:

```
#include <gino-c.h>
main ()
{
/* Initialize GINO and nominate
   X windows device */
   gOpenGino();
   gXwin();
/* Set up storage for display
   file in memory */
   gSetWorkspaceLimit(5000);
   gDefineSegWorkspace(5000);
   gSetSegMode(GSOFTWARE);
/* Open Segment for subsequent
   picture */
   gOpenSeg(1);
   picture();
   gCloseSeg();
   gCloseDevice();
/* Nominate plotter device */
   gHp7475();
/* Redraw all segments from
   display file */
   gDrawSeg(-1);
   gCloseDevice();
/* Free memory used for Software
   Display File */
   gCloseGino();
}
```

```
Program main

use gino_f90
! Initialize GINO and nominate
! X windows device
   call gOpenGino
   call gXwin
! Set up storage for display
! file in memory
   call gSetWorkspaceLimit(5000)
   call gDefineSegWorkspace(5000)
   call gSetSegMode(GSOFTWARE)
! Open Segment for subsequent
! picture
   call gOpenSeg(1)
   call picture
   call gCloseSeg
   call gCloseDevice
! Nominate plotter device
   call gHp7475
! Redraw all segments from
! display file
   call gDrawSeg(-1)
   call gCloseDevice
! Free memory used for Software
! Display File
   call gCloseGino
   stop
```

Note that hardware segments cannot be used across different devices and they have to be recreated on the second device in such circumstances. This includes segments created on an OpenGL screen device and then required to be drawn on a printer as each device has a different device context and therefore different segment store. The segments have to be recreated after switching to the printer device before being redrawn.

Archiving and Restoring Software Display File

The complete contents of the Software Display File can be archived to a disk file and restored later within the same program or a different one using the following routines:

gArchiveSegs(file)

gRetrieveSegs(file)

In each case **file** is a file pointer or unit returned from the function gFopen() which references a file to or from which the Software Display is archived.

The routine gArchiveSegs() copies the contents of the Software Display File from either memory or the current scratch file to a permanent file while gRetrieveSegs() copies a previously archived file into memory or scratch file for subsequent use on the current graphics device. The routine gRetrieveSegs() will remove all currently existing segments before restoring the contents of the archived file.

It is necessary to call gSetSegMode() with **sw** = GMIXWARE or GSOFTWARE before calling gArchiveSegs() or gRetrieveSegs() to ensure the Software Display File is active (see page 423). It is also advisable to call gSetSegMode(GSOFTWARE) before calling gRetrieveSegs() to prevent hardware segment facilities being used as these will not be in step with GINO's software display file.

The following example illustrates the use of archiving and restoring the Software Display File over two separate programs:

C code

```
#include <gino-c.h>
main ()
{
/*    PROGRAM ONE      */
      GFILE *file;

/*    OPEN GRAPHICS DEVICE */
      gOpenGino();
      xxxxx();

/*    INITIALIZE SDF IN MEMORY */
      gSetWorkspaceLimit(10000);
      gDefineSegWorkspace(10000);
      gSetSegMode(GSOFTWARE);

/*    CREATE SEGMENT STRUCTURES */
      gOpenSeg(10);
...
...
...
      gCloseSeg();

/*    ARCHIVE SEGMENT STORE */
      file = gFopen("SDFFILE","w");
      gArchiveSegs(file);
      gFclose(file);

/*    CLOSE GRAPHICS DEVICE AND GINO */
      gCloseDevice();
      gCloseGino();
}
```

```
#include <gino-c.h>
main ()
{
/*    PROGRAM TWO      */
      GFILE *file;

/*    OPEN GRAPHICS DEVICE */
      gOpenGino();
      xxxxx();

/*    INITIALIZE DISPLAY FILE */
      gSetSegMode(GSOFTWARE);

/*    RESTORE SEGMENT STORE INTO SDF SCRATCH FILE
      (gDefineSegWorkspace() NOT CALLED) */
      file = gFopen("SDFFILE","r");
      gRetrieveSegs(file);
      gFclose(file);

/*    DRAW ALL STORED SEGMENTS */
      gDrawSeg(-1);
...
...
...
      gCloseDevice();
      gCloseGino();
}
```

F90 code

```
program one
use gino_f90
common rbuf(10000)
!   PROGRAM ONE
      integer file
!
!   OPEN GRAPHICS DEVICE
!
      call gOpenGino
      xxxxx
!
!   INITIALIZE SDF IN MEMORY
!
      call gSetWorkspaceLimit(1,10000)
      call gDefineSegWorkspace(10000)
      call gSetSegMode(GSOFTWARE)
!
!   CREATE SEGMENT STRUCTURES
!
      call gOpenSeg(10)
...
...
...
      call gCloseSeg
!
!  ARCHIVE SEGMENT STORE
!
      file=gFopen('SDFFILE',GWRITE)
      call gArchiveSegs(file)
      call gFclose(file)
```

```
!
!   CLOSE GRAPHICS DEVICE AND GINO
!
      call gCloseDevice
      call gCloseGino
      stop
      end
```

```
program two
use gino_f90

!  PROGRAM TWO
      integer file
!
!  OPEN GRAPHICS DEVICE
!
      call gOpenGino
      call xxxxx
!
!  INITIALIZE DISPLAY FILE
!
      call qSetSeqMode(GSOFTWARE)
!
!  RESTORE SEGMENT STORE INTO SDF SCRATCH FILE
!  (gDefineSegWorkspace() NOT CALLED)
!
      file=gFopen('SDFFILE',GREAD)
      call gRetrieveSegs(file)
      call qFclose(file)
!
!  DRAW ALL STORED SEGMENTS
!
      call gDrawSeg(-1)
...
...
      call gCloseDevice
      call gCloseGino()
      stop
      end
```

Chapter 25

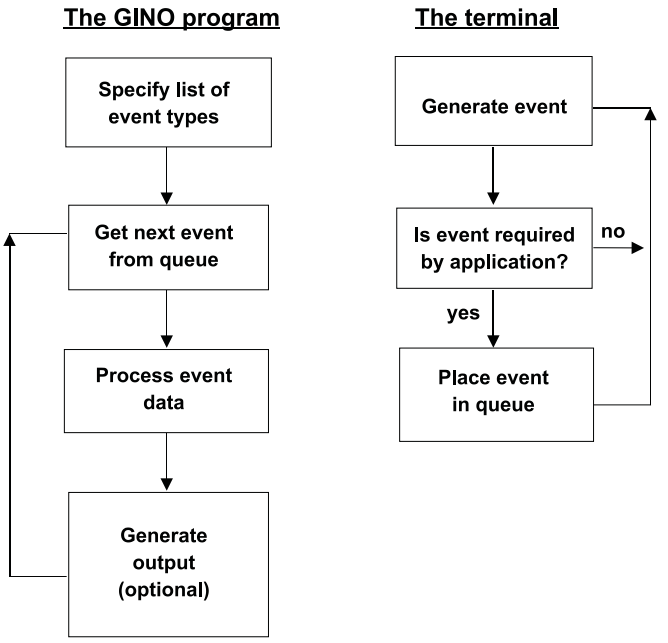
ADVANCED INTERACTION

Advanced Interaction Introduction

GINO enables information to be input from any type of graphics terminal from raster and refresh displays to window based workstations and PCs. The following section describes advanced input facilities available on devices and workstations that cater for several types of input and maintain event queues. These facilities also enable programmers to write event based applications on X window workstations and terminals. Users should refer to Appendix B to see if these facilities are available on the requested device.

In general, when using intelligent displays, the terminal is connected as a satellite or server to a host computer or application client. The GINO program is resident in the host computer and data is being transmitted to and from the terminal or workstation. Before reaching the input stage, the application will prepare the display by creating menu buttons, hit sensitive picture segments or prompt messages. During this stage, some terminals will remain idle whereas highly interactive terminals or workstations may be generating input events from other windows all of which are thrown away, as the GINO application is not ready for them.

When the application reaches the input phase of the program, it is necessary to tell the terminal which input events the application is going to be interested in. The typical mode of working would then be as follows:



Events are generated from input devices such as keyboards, function boxes, light pens, mouse movements etc. An event is generated when the state of an input device is changed. This is usually done by the program user depressing a key, moving the mouse or executing a similar external action. If the event is of a type requested by the application, it will be placed in a queue, otherwise it will be thrown away. Meanwhile the application program needs to fetch the events from the queue and process them accordingly.

Thus two processes are going on at the same time: the terminal is generating events from the user's actions and the application program is processing them as and when they arrive on the queue. When writing such an application it is advisable to program around an event loop. Further information on this is given in 'Queues'.

Associated with any event there may be additional data which is returned with it. This may include the key pressed, or a screen coordinate or a series of values from a digitizer. This information is passed back with the event and is available to the GINO programmer.

Programming in a windowing environment

In a window environment there are many other types of events that can take place, some of which the application is interested in. These include resize events, where the user has decided to change the window size or a change of focus event when the pointer enters or leaves the window. These are also catered for with these advanced input facilities.

Several other event types are taken care of automatically by the window driver and the GINO programmer need not be concerned with them. These include expose and iconize events where an area of the window or the whole window are exposed after having been hidden. It must be remembered however, that these events can only be acted upon if the application program is frequently re-entering the driver. This can either be for graphics output or input such as the event loop described above.

Event Types

The following event types are catered for within GINO:

GNULL	Null event	Indicates that the event queue is to be sampled and/or no valid event has occurred.
GKEYPRESS	Key or function button press	A single key on a keyboard or button on a mouse.
GSEGMENT	Picture segment number (no key)	Returned when using an input device such as a light pen.
GSEGMENTANDKEY	Picture segment number and key or function button	Returned when using the cursor or mouse such that a key is required to generate the event.
GLOCATOR	Screen position and key or function button	Returned when the cursor/pointer is being used to indicate the screen position. The event is returned when a key or function button is pressed.
GSTRING	Text string	One or more keys from a keyboard terminated by a record terminator (carriage return etc.).
GREALS	Real values	A stream of real values returned from a coordinate system not associated with the screen. i.e. A tablet or data logger.
GINTEGERS	Integer values	A stream of integer values returned from a coordinate system not associated with the screen. i.e. A tablet or data logger.

GMOVEMENT	Pointer, mouse or tablet movement	Returned if any pointer, mouse or tablet movement has taken place.
GKEYRELEASE	Key or function button release and screen position	If a single key or mouse button is released.
GRESIZE	Window resize	If the application user has changed the size of the window in which the application is running.
GPOINTERLEAVING	Pointer leaving window	If the pointer has left the window.
GPOINTERENTERING	Pointer entering window	If the pointer enters the window.
GMOUSEWHEEL	Movement of mouse wheel(s)	If mouse wheel(s) are moved

Requesting Event Types

In order to process an event of a given type, the device must be set to place such event in its event queue. This may be done using the routine:

gAddEventType(intype)

This can be called repeatedly to generate a list of possible event types. An example of setting the terminal to generate several event types is where the application may wish to know about all button/key presses and pointer movements in order to drag a picture across the screen while a button is pressed:

```
gAddEventType(GKEYPRESS);
gAddEventType(GMOVEMENT);
gAddEventType(GKEYRELEASE);
```

```
call gAddEventType(GKEYPRESS)
call gAddEventType(GMOVEMENT)
call gAddEventType(GKEYRELEASE)
```

Some devices can only enable one data type at a time and, in this case, the one that is enabled is the one that was requested last.

When the device has any event type set, it is placed in input mode and a special cursor or pointer shape is displayed to indicate to the user that the mode has changed. Where the terminal can display different cursor or pointer shapes the routine gSetCursorType() can be used to set the shape required.

Deleting Event Types

An event type previously requested may be deleted from the list using the routine:

gRemoveEventType(intype)

All event types may be removed from the list by setting **intype** to GALL. If gRemoveEventType() is called to delete the event type for which the terminal is currently set, then the terminal assumes that none are set and it comes out of input mode. Under these circumstances, any graphics cursor is removed or special pointer shape is reverted to its default.

Getting Next Event

The GINO program may process the next event using the routine:

gWaitForEvent(intype)

where **intype** returns the type of the event that has been processed (i.e. that which was at the head of the input queue when gWaitForEvent() was called).

If there were no events in the queue, either because none have been generated or none of the requested types have been generated, then gWaitForEvent() will wait for an event of a requested type to be placed in the input queue and then return. The exception to this is if the GNULL event type has been requested, in which case, gWaitForEvent() will return with **intype** set to GNULL if there are no other events happening. This passes control back to the application rather than locked inside the routine gWaitForEvent().

Reading Event Data

All information associated with an event can be obtained using the routine:

gGetEventRecord(intype,everec)

where **everec** is a structure of type GEVEREC containing the following elements:

int	key	code of any key/mouse button pressed or released
int	impkey	implement number that generated the key code
int	impdat	implement number of the device that generated the data values
int	nseg	segment number from events GSEGMENT and GSEGMENTANDKEY
GPOINT	pos	screen coordinates associated with any of the events
int	nargs	gives the number of data values returned in event.args or event.iargs
float	args[80]	array of length 80 containing real values from event types GREALS and GRESIZE
int	iargs[80]	array of length 80 containing integer values from event types GSTRING, GINTEGERS and GMOUSEWHEEL

Event GSTRING returns the ASCII codes of the text string in **everec.iargs**, and event GRESIZE returns the width and height of the resized window in the first two elements of the array **everec.args**.

Event GMOUSEWHEEL caters for the movement of the PC Intellipoint mouse wheel(s). Values are returned in the **everec.iargs** array (±120 equates to one notch in either direction).

It is recommended that gGetEventRecord() is called immediately after gWaitForEvent() as it is possible that some of the event information can be changed by other GINO routines.

Keys

The **key** value returned by gGetEventRecord() (or gGetCursorEvent()) represents the ASCII code of the key pressed or being pressed as part of the event. The key value may have been generated from the keyboard, mouse or other implement as indicted by the **impkey** setting (see page 454).

As well as standard ASCII values being returned GINO returns non-ASCII values to indicate the action of special mouse and keyboard keys as follows:

key	**Keyboard Key**
0-127	Standard 7-bit ASCII characters
76	Left mouse button
77	Middle mouse button
82	Right mouse button
513-532	Function keys 1-20 (i.e. function key n + 512)
533	Left arrow
534	Right arrow
535	Up arrow
536	Down arrow
537	Prior/Page Up
538	Next/Page Down
539	Insert
540	Home

541	End
542	Shift/TAB
544	Numerical key pad /
545	Numerical key pad *
546	Numerical key pad -
547	Numerical key pad +
548	Numerical key pad 'Enter'
549	Numerical key pad .
550-559	Numerical key pad 0-9
560	Print Screen
561	Pause
562	Select
563	Execute
564	Help

The following keyboard combinations are also returned by gGetEventRecord(), but where the required combination is not catered for it is suggested that the function gEnqKeyState() is used (see page 457).

key	**Keyboard Key**
X + 64	SHIFT + special key > 511 (i.e. SHIFT F1 = 577)
X - 64	CTRL + special key > 511 (i.e. CTRL F1 = 449)
X + 1024	ALT + any key (i.e. ALT P = 1104, ALT F1 = 1537)

If no key was typed, **everec.key**= 0.

Event Generating Implements

In general, it is not necessary for the GINO program to be aware of the implement on which an event was generated since the main interest is in the type of data returned and not in how the input was generated. However, some terminals may have more than one implement type and the program may need to distinguish between them. This is particularly the case for the key or button press, in that it could be generated from the keyboard, function keys, mouse or tablet. When an event is generated, the information is obtained by calling gGetEventRecord(). this includes the identification of the implement on which the key was pressed and that of the implement which generated the data (in **everec.impkey** and **everec.impdat**). A list of implement types is given below:

0	Screen
100	Keyboard
200	Function box
300	Light pen
400	Joystick/arrow keys/thumb wheels/mouse
500	Tablet/digitizer
600	Valuator

Event Programming

In order to program an event loop as suggested in ‘Introduction’, the following code template can be used:

C code

```
#include <gino-c.h>
GEVEREC  event;
int event_type;

/* Set up event types */
   gAddEventType(GKEYPRESS);
   gAddEventType(GMOVEMENT);
   gAddEventType(GKEYRELEASE);
   qAddEventType(GRESIZE);
/* Enter event loop */
   for (;;) {
      gWaitForEvent(&event_type);
      if(event_type == 0) continue;

      gGetEventRecord(event_type, &event);
```

```
      switch (event_type) {
         case GKEYPRESS: {
/* Process key/button press */
          if(event.key == 113) break;
...
...      }
         case GMOVEMENT: {
/* Process mouse movement */
...
...      }
         case GKEYRELEASE: {
/* Process key/button release */
...
...      }
         case GRESIZE: {
/* Process resize event */
...
...      }
         default : continue;
      }
   }
```

F90 code

```
use gino_f90
type (GEVEREC)  event
integer event_type

!  Set up event types
   call gAddEventType(GKEYPRESS)
   call gAddEventType(GMOVEMENT)
   call gAddEventType(GKEYRELEASE)
   call gAddEventType(GRESIZE)
!  Enter event loop
   do
      call gWaitForEvent(event_type)
      if(event_type == 0) cycle

      call gGetEventRecord(event type, event)
      select case (event_type)
         case (GKEYPRESS)
!  Process key/button press
          if(event%key == 113) goto 999
...
…
         case (GMOVEMENT)
!  Process mouse movement
...
…
         case (GKEYRELEASE)
!  Process key/button release
...
…
         case (GRESIZE)
!  Process resize event
...
…
         case default
      end select
   end do
999
```

Queues

Events may be queued by the intelligent terminal. The user program may interrogate the state of the queue, or delete the queue by using the routines:

gEnqQueueLength(len)

gDeleteEventQueue()

The information returned in **len** depends on the terminal. In some cases the actual number of waiting events may be returned, but the terminal may only be able to indicate that there are at least **len** events waiting.

For example, to interrogate the event queue and read an event when there is an event waiting:

```
/* Interrogate length of queue */
   gEnqQueueLength(&len);
/* Enter loop if some events */
   while ( len > 0) {
/* Read event and process */
      gWaitForEvent(&event_type);
      :
      :
/* Interrogate length of queue
   again */
      gEnqQueueLength(&len);
   }
```

```
! Interrogate length of queue
  call gEnqQueueLength(len)
! Enter loop if some events
  do while ( len > 0)
! Read event and process
     gWaitForEvent(event_type)
     :
     :
! Interrogate length of queue
! again
     call gEnqQueueLength(len)
  end do
```

Consult the device driver Appendix B for information on whether queues are supported.

Mouse Position

Two routines are provided to set and enquire the current position of the mouse pointer:

gSetMousePos(env,xpos,ypos)

gEnqMousePos(env,point)

where **env** is the environment of the request. This can be either GSCREEN where the mouse position is relative to the complete screen or GDRAWINGAREA where the mouse position is relative to the current drawing area or window. In both cases the mouse position is measured in pixels relative to the top left corner of the particular environment.

The routine gEnqMousePos() returns the mouse position in a structure of type GPIXEL containing both the x and y coordinates.

Users should note that it is the convention of GUI programs that the mouse position is moved only in very rare cases, so the routine gSetMousePos() should be used sparingly.

Keyboard State

In addition to the value of the key pressed as part of an event, the current state of any key on the keyboard of the current device may be enquired at any point in a graphics application using the following function:

state = gEnqKeyState(key)

where **key** is the keycode of the key being enquired. The function returns a value of 0 if the key is not pressed and a value 1 if the key is pressed at the time the call on the function is made.

The key code table used by gEnqKeyState() is the same as the **everec.key** values returned by gGetEventRecord() (see above) with the following additions:

key	Special Key Description
-1	Left mouse button
-2	Middle mouse button
-3	Right mouse button
-20	Shift
-21	Ctrl
-22	Alt
-30	Caps Lock
-31	Num Lock
-32	Scroll Lock

Therefore, the code to test whether both the Ctrl and ‘A’ keys are currently pressed down is as follows:

```
/* Check for Ctrl A */
   if(gEnqKeyState(-21) &&
      gEnqKeyState(65)) {

}
```

```
! Check for Ctrl A
  if(gEnqKeyState(-21) .and.
     gEnqKeyState(65)) then

  endif
```

Chapter 26

SYSTEM UTILITIES

System Utilities Introduction

Within any graphics application there is often a need to access system utilities not provided as part of the programming language in which the application is written. Some of these utilities may be provided as extensions to the programming language or as additional libraries associated with the language or as part of the operating system. In any event, by their very nature, there is no common interface to these utilities, and so any attempt to use them would render the graphics application implementation dependent.

This section of the document describes a number of routines that have shown to be commonly required by graphical applications, and have been included in the GINO library to offer a common interface to such utilities. The utilities are grouped in the following sub-sections:

- Directory handling
- Time and date utilities
- Other system utilities
- Random number generation
- String handling

File and Directory Handling

GINO provides a number of file and directory handling utilities. While the interface to these routines is the same for any implementation of GINO, the structure of directory and file names vary from system to system so it becomes quite difficult to use these routines in a way that is independent of the system in which the application is running.

Under the most common GINO implementations, directory and file names occur in the following forms:

Operating System	Disk or Node	Directory	Filename	Wild Chars
UNIX	node	/dir1/dir2/	file.ext (n.n)	* or ? or []
OpenVMS	DISK:	[DIR1.DIR2]	FILE.EXT (n.n)	* or %
DOS	DISK:	\DIR1\DIR2\	FILE.EXT (8.3)	*
MS WINDOWS	DISK:	\DIR1\DIR2\	file.ext (n.n)	*

The two routines to enquire and set the current working directory are as follows:

gEnqWorkingDir(directory [,slen])

err=gSetWorkingDir(directory)

The former returns the current working directory in the string **directory** (in C/C++ this may alternatively be as a pointer to a malloc'ed character string, which should be freed after use) and the latter is a function returning an error code if the setting operation is unsuccessful. eg.

```
   char *dir;
/* Enquire current working dir */
   dir=gEnqWorkingDir(NULL,0);
/* Set working directory */
   err=gSetWorkingDir(dir);
   free(dir);
```

```
   character*80 dir
! Enquire current working dir
   call gEnqWorkingDir(dir)
! Set working directory
   ierr=gSetWorkingDir(dir)
```

The string passed to gSetWorkingDir() must conform to the correct format for a directory name under the operating system being used as will the string returned by gEnqWorkingDir().

The two functions provided to return a list of file names are gGetDirList() and gGetFullDirList(), with the former simply returning a list of file names in the current directory, while gGetFullDirList() will return a list of files matching any permitted directory search pattern together with their associated attributes.

err=gGetDirList(n, files [,flen])

err=gGetFullDirList(pattern, n, files, types, dirdates, sizes [,flen])

In both cases the user must pass addresses to the appropriate arrays which should be of length **n**. Assuming there are files in the current directory or that match the search pattern, at most **n** file names (and their associated attributes) will be returned in those arrays. **n** is set to the actual number of entries returned by each routine. If the arrays are not large enough to hold all the entries in the directory, some of the names will not be returned and **n** will remain unchanged on exit.

An extra argument, **flen**, is required in the C/C++ binding of these routines to inform the GINO function how much space has been allocated for the list of file names. This should equate to the maximum length required for each file name, effectively the width of the array of character strings.

The search pattern passed to gGetFullDirList() is the string **pattern** which must conform to that permitted under the implementation being used and may include disk and directory name and/or wild characters. In addition to the file names, gGetFullDirList() returns the file type (as a bit pattern), the date last modified (or created) and the file size in bytes.

The following functions provide utilities for creating, copying, renaming and deleting files and or directories:

err=gCreateDir(directory)

err=gCopyFile(filea, fileb)

err=gRenameFile(filea, fileb)

err=gRemoveDir(directory)

err=gRemoveFile(file)

Where each argument is a character string and each function returns a value of zero if the operation is successful. System dependent error codes are returned if unsuccessful.

The following example shows a usage of the above routines in an implementation independent way:

C code

```
   FILE *fp;
   char dir[80];

/* Enquire current working dir */
   gEnqWorkingDir(dir,80);
/* Create new temp directory */
   if(gCreateDir("temp") == 0) {
     if(gSetWorkingDir("temp") == 0) {
/* Create file */
       if((fp=fopen("temp.txt","w")) != NULL) {
         fprintf(fp,"A new file\n");
         fclose(fp);
/* Rename file */
         if(gRenameFile("temp.txt","new.txt") == 0)
/* Remove file */
           gRemoveFile("new.txt");
         else
           gRemoveFile("temp.txt");
       }
/* Reset current directory */
     gSetWorkingDir(dir);
     }
/* Remove temporary directory */
     gRemoveDir("temp");
   }
```

F90 code

```
  character*80 dir

! Enquire current working dir
  call gEnqWorkingDir(dir)
! Create new temp directory
  if(gCreateDir('temp').eq.0) then
! Change to temp directory
    if(gSetWorkingDir('temp').eq.0) then
! Create file
      open(unit=10,file='temp.txt',status='new',iostat=ier)
      if(ier.eq.0) then
        write(10,*) 'A new file'
        close(10)
! Rename file
        if(gRenameFile('temp.txt','new.txt').eq.0) then
! Remove file
          ier=gRemoveFile('new.txt')
        else
          ier=gRemoveFile('temp.txt')
        end if
      end if
! Reset current directory
      ier=gSetWorkingDir(dir)
    end if
! Remove temp directory
    ier=gRemoveDir('temp')
  end if
```

Time and Date Utilities

The GINO library also provides a common interface to the following date and time utilities:

Return system date	gEnqSysDate(date)
Return system time	gEnqSysTime(time)
Return date and time as string	gEnqSysDateStr(date [,slen])
Unpack system date/time	gReturnDirDate(dir_date, date, time)
Pause	gTimeDelay(millisec)

The routine gEnqSysDateStr() returns the current date in the string **date** (in C/C++ this may alternatively be as a pointer to a malloc'ed character string, which should be freed after use). The routine gReturnDirDate() is used solely to unpack system date and time information returned by the system utility gGetFullDirList().

Other System Utilities

GINO also provides a number of other miscellaneous utilities which have shown to be useful in graphics applications.

Command-line arguments

Whilst the standard argc and argv arguments to main() are available to GINO-C programmers to access the command line arguments, Fortran 90 programmers need to use the following routine:

gEnqSysArgs(n, args)

This routine returns the command line arguments including the application program name in the character array **args**. The variable **n** is passed as the size of the array **args** and is returned with the number of elements filled. Note that the program name is returned in **args(1)** and the first command line argument is returned in **args(2)** and so on.

The following code shows the equivalent methods of returning command-line arguments in C and F90:

```
main(int argc, char *argv[])
{
int nargs;
char name[30];

/* Get number of arguments */
   nargs=argc;

/* Get program name */
   strcpy(name,argv[0]);
```

```
character*30 args(6)
character*30 name
n=6
call gEnqSysArgs(n,args)

! Get number of arguments
   nargs=n

/* Get program name
   name=args(1)
```

The format of the application name varies according to the implementation and may or may not include the complete path or a .exe extension.

Enquire User Name

gEnqSysUsername(uname ,[slen])

The routine gEnqSysUsername() returns a login or access name in most multi-user implementations, or an implementation dependent string in single user or PC implementations. The name is returned in the string **uname** (in C/C++ this may alternatively be as a pointer to a malloc'ed character string, which should be freed after use).

```
char *name;
name=gEnqSysUsername(NULL,0);
```

```
character*128 name
call gEnqSysUsername(name)
```

Environment Variable Settings

The routine gEnqSysEnviron() will return the setting of a system environment variable as appropriate to the implementation.

gEnqSysEnviron(environ, setting [,slen])

The string is returned in the string **setting** (in C/C++ this may alternatively be as a pointer to a malloc'ed character string, which should be freed after use).

```
char *envname = "PATH";
char *path;
gEnqSysEnviron(envname,NULL,0);
path=free(path);
```

```
character*4   :: envname = 'PATH'
character*256 :: path
call gEnqSysUsername(envname, &
                     path)
```

The means of setting such variables will vary from system to system but the following are common examples:

UNIX:

setenv DISPLAY portos:0

OpenVMS:

DEFINE DISPLAY ALPHA2:0

DOS/WINDOWS:

SET TZ=GMT0BST

If the requested variable is not set, or cannot be returned, the gEnqSysEnviron() returns a NULL or blank string.

System Command Execution

The routine gExecuteSysCommand() provided a means to execute a system command supplied in the character string **command**.

err=gExecuteSysCommand(command,[gShow,gSuspend,gHandle])

By default, the requested command will be executed as a separate process on the host with control being passed back to the calling application immediately. The optional arguments may however be used to control visible state of the new process (on Windows platforms), the suspension state of the calling application and return the new processes' handle if available.

The function returns an error flag which is set to a system dependent non zero value if the execution was not possible or failed for some reason. Any output that is generated from the command will be directed to the standard output stream and may therefore appear in the application window or display. A common means of avoiding this (as it may corrupt graphics output at the same location) is to redirect output to an alternative location or file as shown in the following examples:

```
err=gExecuteSysCommand("ls >/dev/null")          UNIX
err=gExecuteSysCommand("DIR/OUTPUT=NULL")        OpenVMS
err=gExecuteSysCommand("DIR >NUL")               DOS
```

Task Priority

Two routines are provided to set and enquire the current application task priority:

gSetSysPriority(pri)

gEnqSysPriority(pri)

where the possible settings for **pri** are:

GREALTIME	Sets task priority to highest possible. This should only be done for very short periods of time as it will prevent any other application from operating, including mouse operation
GHIGH	Sets task priority above normal
GNORMAL	Normal
GLOW	Sets task priority below normal
GIDLE	Sets task to idle state where task is only continued if no other applications or system processes are running

Sound System Speaker

The routine gPlaySound() attempts to use any sound facilities available to sound the specified note for the required duration in milliseconds.

gPlaySound(freq, time)

This routine is currently only implemented under Windows and some other DOS implementations of GINO. Under Windows, the standard 'beep' will play for all positive values of **freq**, but negative values can be used to access the Windows alert sounds.

Random Number Generation

Two system utilities are provided to enable both non-repeating and repeated random number generation.

gSetRandSeed(seed)

gGetRand(rand)

The routine gSetRandSeed() will set an integer seed value to a compiler-dependent pseudo random number generator, while the routine gGetRand() will return a random number between 0.0 and 1.0.

Note that GINO will provide a seed based on some time/date information when initialized so if an application needs a non-repeating random number sequence there is no need to call gSetRandSeed().

String Handling

A single function gTrueLen() returns the length of a character string up to the last non blank character.

len=gTrueLen(string)

This function is useful when examining the contents of a string returned by a GINO routine or system function.

Chapter 27

ROUTINE SPECIFICATIONS

An Introduction to Routine Specifications

The following pages provide the routine specifications for all the GINO routines, in alphabetical order.

Each routine specification provides the following information:

- A syntax of the routine name and its arguments. All C functions are of type **void** unless otherwise stated. A list of the arguments follow, with their associated type and a description. Where certain GINO constants or other values have special meaning for the argument, these are given with the description.
- A description of the task the routine performs.
- A reference to the relevant sections of this document (and to any other appropriate documentation) that discusses the use of the routine.

The use of the routines is discussed in the preceding chapters along with example programs using these routines and associated output.

gAddEventType

Syntax

C/C++:	**void gAddEventType**(int intype);

F90:	**subroutine gAddEventType**(intype) integer, intent(in) :: intype

Arguments

intype

Event type

= GNULL,	Null event type
= GKEYPRESS,	Key or button press
= GSEGMENT,	Picture segment number
= GSEGMENTANDKEY,	Picture segment number and key/button
= GLOCATOR,	Screen position and key/button press
= GSTRING,	Text string
= GREALS,	String of real values
= GINTEGERS,	String of integer values
= GMOVEMENT,	Pointer, mouse or tablet movement
= GKEYRELEASE,	Key or button release
= GRESIZE,	Window resize event
= GPOINTERLEAVING,	Pointer leaving window
= GPOINTERENTERING,	Pointer entering window
= GMOUSEWHEEL,	Mouse wheel movement

Description

The routine gAddEventType() adds an event type to the list of event types which may be expected in subsequent calls to gWaitForEvent() and sets the preferred event type to **intype**. The routine gAddEventType() has the effect of enabling all the implements that may generate the specified data type and sets the terminal in the mode for sending that data. The terminal will remain in that mode until:

(a) Another call to gAddEventType() requesting a different event type

(b) A call to gRemoveEventType() deleting the event type

(c) The terminal operator changes the mode by local action

The routine gAddEventType() must be called before any data can be returned from the routine gWaitForEvent().

See Also

Page 450
gRemoveEventType
gWaitForEvent

gArchiveSegs

Syntax

C/C++:	**void gArchiveSegs**(GFILE *fp);

F90:	**subroutine gArchiveSegs**(unit) integer, intent(in) :: unit

Arguments

fp
GINO-C file pointer to archive Software Display File

unit
Fortran 90 file unit to archive Software Display File

Description

The routine gArchiveSegs() archives the complete contents of the Software Display File either from memory or file into a sequential file. The file may later be restored using gRetrieveSegs().

The file should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

If the Software Display File is not active (ie. gSetSegMode() has not been called) an error message is output and no action is taken. The complete contents of the Software Display File will be archived except any contents of segment zero (the non-retained segment). If there is a file open on **fp** then it will be closed and deleted.

See Also

Page 442
gFopen
gRetrieveSegs
gSetSegMode

gBuildMatrix

Syntax

C/C++:	**void gBuildMatrix2D**(float xo, float yo, float dx, float dy, float angz, float sx, float sy, GMAT2D t2d); **void gBuildMatrix3D**(float xo, float yo, float zo, float dx, float dy, float dz, float angx, float angy, float angz, float sx, float sy, float sz, GMAT3D t3d);

F90:	**subroutine gBuildMatrix2D**(xo,yo,dx,dy,angz,sx,sy,t2d) **subroutine gBuildMatrix3D**(xo,yo,zo,dx,dy,dz,angx,angy,angz,sx,sy,sz,t3d) real, intent(in) :: xo,yo,zo,dx,dy,dz real, intent(in) :: angx,angy,angz,sx,sy,sz real, intent(out) :: t2d(6),t3d(16)

Arguments

xo,yo,zo
Point about which scaling and rotation take place

dx,dy,dz
Translation distances

angx,angy,angz
Angle of rotation in degrees

sx,sy,sz
Scaling factors

t2d,t3d
Output transformation matrix

Description The routines gBuildMatrix2D() and gBuildMatrix3D() are called to build a transformation matrix containing a rotation, a translation, and scaling factors. The scaling, rotation, and translation are all relative to the point **xo, yo, (zo)**.

See Also Page 378
gCombineMatrix

gCGMInterpreter

Syntax

C/C++:	**void gCGMInterpreter**(int code, GFILE *fp, int nseg, int mode, int errlev);

F90:	**subroutine gCGMInterpreter**(code,unit,nseg,mode,errlev) integer, intent(in) :: code,unit,nseg,mode,errlev

Arguments

code
CGM encoding type

= GCGMCHAR,	Character encoding
= GCGMBINARY,	Binary encoding

fp
GINO-C file pointer

unit
Fortran 90 file unit

nseg
Picture segment number
(reserved for future use)

mode
Interpretation mode

= GABSOLUTE,	Metric metafiles drawn the same size that they were generated
= GMAPPED,	Abstract and Metric metafiles scaled to fit the current window limits
= GTRANSFORMED,	Same as GABSOLUTE but subject to current transformation

errlev
Error checking level

= GNO,	No error checking
= GFAST,	Fast error checking
= GFULL,	Full error checking

Description The routine gCGMInterpreter() interprets a complete CGM metafile. The metafile may be either character or binary encoded but the interpreter must be informed of which through the **code** argument.

The metafile should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

Abstract metafiles are drawn such that one VDC unit = one GINO picture unit (in GABSOLUTE and GTRANSFORMED modes).

The routine gCGMInterpreter() may alter the current state of GINO in that the colour table may be changed within a metafile, but all line and character attributes, window and transformation states are restored at the end of a metafile.

See Also Page 69
gFopen

gClearPolygonWorkspace

Syntax

C/C++:	**void gClearPolygonWorkspace**(void);
F90:	**subroutine gClearPolygonWorkspace**

Arguments None

Description Polygon vertices are always stored at the next available location in the polygon workspace. A call to gClearPolygonWorkspace() reinitializes the workspace so that vertices are stored starting from the first location in the workspace. This deletes all existing polygons.

The routine gClearPolygonWorkspace() is ignored if gSetPolygonMode(GOFF) was called beforehand.

See Also Page 250
gSetPolygonMode

gClearViewport

Syntax

C/C++:	**void gClearViewport**(void);
F90:	**subroutine gClearViewport**

Arguments None

Description The routine gClearViewport() clears the current viewport area by filling it with the background colour.

See Also Page 51, 221

gCloseAuxDrawingArea

Syntax

C/C++:	**void gCloseAuxDrawingArea**(int ident);

F90:	**subroutine gCloseAuxDrawingArea**(ident) integer, intent(in) :: ident

Arguments ***ident***
Auxiliary drawing area identifier (2 to MAXAUX)

Description The routine gCloseAuxDrawingArea() is a device independent routine for closing and deleting an auxiliary drawing area that has been opened using gOpenAuxDrawingArea(). The drawing area may have been visible or invisible depending on the value of the identifier. The closure of a visible drawing area (even number) will also remove its associated invisible drawing area or backing store.

The identifier (**ident**) must be in the range 2 to MAXAUX*2+1, where MAXAUX is the maximum number of auxiliary drawing areas that the currently nominated device can handle. This value can be obtained through the routine gEnqDeviceState().

If an auxiliary drawing area is closed while output is being directed to it, all further output will be directed to the default drawing area/window (1).

See Also Page 50
gOpenAuxDrawingArea
gEnqDeviceState

gCloseCGMFile

Syntax

C/C++:	**void gCloseCGMFile**(void);

F90:	**subroutine gCloseCGMFile**

Arguments None

Description Closes CGM metafile that has been opened by gOpenCGMFile().

See Also Page 70
gOpenCGMFile

gCloseDevice

Syntax

C/C++:	**void gCloseDevice**(void);
F90:	**subroutine gCloseDevice**

Arguments None

Description The routine gCloseDevice() terminates output to the currently nominated device.

If any further pictures are to be drawn, a device must be nominated prior to calling any further GINO routines.

If a new device is nominated without a previous call to gCloseDevice() or if gCloseGino() is called, output to the old device is terminated.

See Also Page 52
gCloseGino
gSuspendDevice

gCloseGino

Syntax

C/C++:	**void gCloseGino**(void);
F90:	**subroutine gCloseGino**

Arguments None

Description The routine gCloseGino() terminates GINO, releasing any graphics device that is nominated (implicit gCloseDevice()). It may be called in a program when GINO's facilities are no longer required. All information and definitions prior to gCloseGino() are discarded. Invoking gOpenGino() after a call to gCloseGino() reinitializes GINO.

Because gCloseGino() leaves GINO in a known state, starting up GINO after a call to gCloseGino() does not necessarily require a call to routine gOpenGino() to ensure proper initialization. Any call to a GINO routine will trigger proper initialization.

See Also Page 26, 40
gCloseDevice
gOpenGino

gCloseSeg

Syntax

C/C++:	**void gCloseSeg**(void);

F90:	**subroutine gCloseSeg**

Arguments None

Description The routine gCloseSeg() closes the currently opened picture segment.

If no picture segment is open, an error is generated.

If a new picture segment is opened while a picture segment is open, or if a device is released while a picture segment is open, an internal call to gCloseSeg() is made by GINO.

See Also Page 64, 426

gCombineMatrix

Syntax

C/C++:	**void gCombineMatrix2D**(GMAT2D a2d, float xo, float yo, float dx, float dy, float angz, float sx, float sy, GMAT2D t2d); **void gCombineMatrix3D**(GMAT3D a3d, float xo, float yo, float zo, float dx, float dy, float dz, float angx, float angy, float angz, float sx, float sy, float sz, GMAT3D t3d);

F90:	**subroutine gCombineMatrix2D**(a2d, xo, yo, dx, dy, angz, sx, sy, t2d) **subroutine gCombineMatrix3D**(a3d, xo, yo, zo, dx, dy, dz, angx, angy, angz, sx, sy, sz, t3d) real, intent(in) :: a2d(6),a3d(16),xo,yo,zo,dx,dy,dz real, intent(in) :: angx,angy,angz,sx,sy,sz real, intent(out) :: t2d(6),t3d(16)

Arguments

a2d,a3d
Input transformation matrix

xo,yo,zo
Point about which scaling and rotation take place

dx,dy,dz
Translation distances

angx,angy,angz
Angle of rotation in degrees

sx,sy,sz
Scaling factors

t2d,t3d
Output transformation matrix

Description The routines gCombineMatrix2D() and gCombineMatrix3D() computes a transformation matrix that is the composition of the input matrix **a2d** or **a3d** with a rotation, translation, and scaling factors.

The scaling, rotation and translation are all relative to the point **xo, yo, (zo)**.

See Also Page 378
gBuildMatrix

gConvertInteger

Syntax

C/C++:	**void gConvertInteger**(int number, int nwidth, char string[]);

F90:	**subroutine gConvertInteger**(number, nwidth, string) integer, intent(in) :: number,nwidth character*(*), intent(out) :: string

Arguments

number
Integer value

nwidth
Field width

< 0,	Left-justified
= 0,	No string returned
> 0,	Right-justified

string
Character string containing integer value

Description The routine gConvertInteger() returns the integer value **number** as a decimal character string in the argument **string**. The format of the number is in exactly the same format as that output by gDisplayInteger(). The string starts with a minus sign for values less than zero.

If the number occupies less than **nwidth** character positions, the string is padded out with spaces. If the number is longer than **nwidth** characters, the string is filled with asterisks.

For positive values of **nwidth**, the number is right-justified. If **nwidth** is less than zero the number is left-justified. gConvertInteger() returns nothing if **nwidth** is zero.
Field width is limited to 32 character positions. If it exceeds this, it is set to 32 characters and a warning message is output.

See Also Page 140
gDisplayInteger

gConvertReal

Syntax

C/C++:	**void gConvertRealExponent**(float value, int nwidth, int nplace, char string[]); **void gConvertRealFixed**(float value, int nwidth, int nplace, char string[]); **void gConvertRealFloat**(float value, int nwidth, char string[]);

F90:	**subroutine gConvertRealExponent**(value, nwidth ,nplace, string) **subroutine gConvertRealFixed**(value, nwidth, nplace, string) **subroutine gConvertRealFloat**(value, nwidth, string) real, intent(in) :: value integer, intent(in) :: nwidth,nplace character*(*), intent(out) :: string

Arguments

value
Real value

nwidth
Field width

< 0,	Left-justified
= 0,	No string returned
> 0,	Right-justified

nplace
Number of places after the decimal point

string
Character string containing value

Description

The gConvertReal set of routines converts the supplied real value into a character string in three different floating point formats. Either as a decimal floating-point, a decimal fixed point or a floating point character string in the argument **string**. The format of the number is exactly the same as that output by the equivalent gDisplayReal routine.

In all three routines, if the number occupies less than **nwidth** character positions, the string is padded out with spaces. On the other hand, if the number is longer than **nwidth** characters, the string is filled with asterisks. For positive values of **nwidth**, the number is right-justified. If **nwidth** is less than zero, the number is left-justified. All routines return a blank string if **nwidth** is zero. The field width is limited to 32 character positions. If it exceeds this, it is set to 32 characters and a warning message is output.

In the case of gConvertRealExponent(), the number consists of a mantissa followed by a decimal exponent. The mantissa consists of a decimal point (preceded by a minus sign if **value** is less than zero) followed by the number of digits specified by **nplace**. If **nplace** is less than zero, its absolute value is used and a warning message is output. The number is rounded in the last decimal place. A zero is output in front of the decimal point if the number is positive and left-justified or if the number is right-justified and there is room for a leading zero. The exponent occupies 4 character positions. The first character is the letter 'E' and the remaining characters contain the exponent value, right-justified and preceded by a minus sign for negative exponents. Any gap after the letter 'E' is padded out with spaces. If the exponent is too large, the string is filled with asterisks.

In the case of gConvertRealFixed() the number consists of an integer part (preceded by a minus sign if **value** is less than zero) followed by a decimal point and a fractional part. The number of digits for the fractional part is specified by **nplace**. If **nplace** is less than zero, its absolute value is used and a warning message is output. The number is rounded in the last decimal place. If the integer part is zero, a leading zero is output unless **value** is less than zero and there is only room for the minus sign before the decimal point.

In the case of gConvertRealFloat() the mantissa consists of a decimal point (preceded by a minus sign if **value** is less than zero) followed by (**nwidth**-6) digits, up to a maximum of 6 digits. If the field width is less than 6, the string is filled with asterisks. The number is rounded in the last decimal place. A zero is output in front of the decimal point if the number is positive and left-justified or if the number is right-justified and there is room for a leading zero. The exponent occupies 4 character positions. The first character is the letter 'E' and the remaining characters contain the exponent value, right-justified and preceded by a minus sign for negative exponents. Any gap after the letter 'E' is padded with spaces. If the exponent is too large, the string is filled with asterisks.

See Also

Page 140
gDisplayRealExponent
gDisplayRealFixed
gDisplayRealFloat

gCopyFile

Syntax

C/C++:	**int gCopyFile**(char filea[], char fileb[]);

F90:	**integer function gCopyFile**(filea, fileb) character*(*), intent(in) :: filea, fileb

Arguments

filea
Name of existing file

fileb
Name of new file

Description

The system utility gCopyFile() copies the contents of **filea** into a new file called **fileb**. Either or both file names may be simple file names in the current working directory or full path names.

The function returns an integer value which equal zero if the copy has been successful. A system dependent error code is returned if the copy fails.

See Also Page 461
gMakeDir
gRemoveFile
gRenameFile

gCopyPixelArea

Syntax

C/C++:	**void gCopyPixelArea**(int source, int dest, int ix, int iy, int width, int height, int ixd, int iyd);

F90:	**subroutine gCopyPixelArea**(source, dest, ix, iy, width, height, ixd, iyd) integer, intent(in) :: source,dest,ix,iy integer, intent(in) :: width,height,ixd,iyd

Arguments

source
Source drawing area identifier

= 0,	Screen
= 1,	Backing store (default)
> 1,	User generated drawing area

dest
Destination drawing area identifier

= 0,	Screen
= 1,	Backing store (default)
> 1,	User generated drawing area

ix,iy
Origin of pixel area to copy

width
Width of pixel area

height
Height of pixel area

ixd,iyd
Destination origin

Description

The routine gCopyPixelArea() provides a means to copy a pixel area within the same drawing area or from one drawing area to another. Multiple drawing areas are available on devices that either have a backing store or can generate additional drawing areas using the routine gOpenAuxDrawingArea(). Both the source and destination drawing area identifiers must be within the range of available identifiers, the maximum number being obtained through the routine gEnqDeviceState(). Note that the default drawing area for all devices is always 1. If the device has a backing store can you copy a pixel area to any available drawing area, however if copied to an even-number drawing area, the area will not be automatically repaired.

The pixel area to be copied is specified in terms of an origin (**ix**,**iy**) relative to the top left of the drawing area, and a width and height (**width**,**height**). The whole area is copied such that the new origin is positioned at the coordinate **ixd**,**iyd** on the destination drawing area, again relative to the top left corner. If any of the copied area is outside the drawing area limits, it will be clipped by the hardware.

See Also

Page 203
gOpenAuxDrawingArea
gSelectDrawingArea
gEnqDeviceState

gCopySeg

Syntax

C/C++:	**void gCopySeg**(int nseg, int pos);

F90:	**subroutine gCopySeg**(nseg, pos) integer, intent(in) :: nseg,pos

Arguments

nseg
Picture segment or segment group number

> 0,	Copy segment(s) specified by segment **nseg**
= -1,	Copy all segments
< -1,	Copy all segments except those specified by **nseg**

pos
Position of copy of segment

= GCURRENT,	At the current pen position
= GANCHOR,	At the position of **nseg**

Description

The routine gCopySeg() is used to make a copy of the specified segment into the current segment. The position occupied by the copy of the segment is indicated by **position**.

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(0), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also

Page 434
gSetSegMode

gCreatePlanarShadowMatrix

Syntax

C/C++:	**void gCreatePlanarShadowMatrix**(GPOINT3 plane[], GPOINT3 *light, GMAT3D a3);

F90:	**subroutine gCreatePlanarShadowMatrix**(plane, light, a3) type (GPOINT3), intent(in) :: plane(3),light, real, intent(in) :: a3(16)

Arguments

plane
Array of three points representing a planar surface

light
Position of light source

a3
3D viewing transformation matrix

Description

The routine gCreatePlanarShadowMatrix() constructs a 4x4 viewing transformation matrix required to generate a squashed view of a scene as per a shadow on a planar surface. The planar surface is represented by the three points passed in **plane** (from which a normal is calculated) and the position of the light source is passed in the GPOINT3 structure **light**.

In order to create the shadow, the returned matrix **a3** should be passed to the routine gModifyView(), after which all the objects in the scene should be redrawn in black or a suitably dark, slightly transparent, material.

See Also

Page 342
gModifyView

gDebug

Syntax

C/C++:	**void gDebug**(GFILE *fp, int level);

F90:	**subroutine gDebug**(unit, level) integer, intent(in) :: unit,level

Arguments

fp
GINO-C file pointer

unit
Fortran 90 File unit

level
The level of information to be output

= GSTANDARD,	Lists all device driver entries
= GEXTRA,	Lists all device driver entries and extra monitoring

Description

The routine gDebug() traces the flow of information in GINO. It sits between GINO and a device driver and writes to the specified GINO file unit all the information that passes back and forth. As far as possible, gDebug's output mirrors the calls to GINO routines in the user's program.

The routine gDebug() must be called prior to the call to the device driver's nomination routine. Note that gDebug() makes an implicit call to gCloseDevice() and gExtendSeg().

The debug file should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate. The file is neither opened or closed by the gDebug() routine itself.

Both debug levels monitor the data sent to and received from the currently nominated device driver, but the GEXTRA level also monitors changes to windowing and masking state as well as the line mode and state of transformation current at each gOpenSeg() and gCloseSeg().

Control of gDebug() output can be made with gSetDebugSwitch().

See Also

Page 31, 42
gSetDebugSwitch
gCloseDevice
gFopen

gDefineBrokenLineStyle

Syntax

C/C++:	**void gDefineBrokenLineStyle**(int brk, GBRKSTY *rep);

F90:	**subroutine gDefineBrokenLineStyle**(brk, rep) integer, intent(in) :: brk type (GBRKSTY), intent(in) :: rep

Arguments

brk
Broken line index 1 to 256 only

rep.mode
Line mode (which may be one of the following)

= GSOLID
= GCONTDASH
= GCONTCHAIN
= GDISCONTDASH
= GDISCONTCHAIN

rep.repeat
Repeat length in current units

rep.dash
Dash length in current units

rep.dot
Dot length in current units (for chained lines)

Description

The routine gDefineBrokenLineStyle() allows the user to redefine the line type parameters associated with a specified index **brk** in the broken line table. There are 256 entries in the table and if **brk** is outside the range of 1 to 256, an error message is output and no further action is taken.

The new representation is defined in the structure **rep** which is of type GBRKSTY.

If **rep.mode** is out of range, a warning message is output and a default value of GSOLID is assumed. If **rep.mode** is GSOLID, the line type is solid and the values in **rep.repeat**, **rep.dash** and **rep.dot** are ignored. Otherwise these values specify the dimensions in current units of the broken line pattern. If any of **rep.repeat**, **rep.dash** or **rep.dot** is less than zero, a warning message is output and the absolute value is used. If **rep.repeat** is less than **rep.dash** for a dashed line type or if **rep.repeat** is less than **rep.dash**+**rep.dot** for a chained line type, a warning message is output and the line type defaults to being solid.

rep.mode also specifies how the broken line pattern is positioned when lines are output. If **rep.mode** is either GCONTCHAIN or GCONTDASH (continuous line), the pattern carries on from one line to the next. If **rep.mode** is either GDISCONTCHAIN or GDISCONTDASH (discontinuous line), the pattern is centred along each straight line segment and, if necessary, scaled down so that each line segment starts and ends with an equal length dash.

When a call to gSetBrokenLine() selects a line type in the range 1 to 256, the corresponding line type parameters come into effect. If a call to gDefineBrokenLineStyle() redefines the line type parameters associated with the current line type, i.e. a call to gSetBrokenLine(brk) precedes the call to gDefineBrokenLineStyle(), the new line type parameters immediately come into effect.

If the line type cannot be generated by the device exactly as specified, it will be generated by GINO's software. Appendix B should be consulted to see whether the device can cope with broken line types that are specified in terms of the line type parameters. gSetBrokenLineMode() may be called to force GINO to generate all broken lines, thereby ensuring that they are output correctly on any device.

When a device is nominated the line type parameters are set to the following set of 16 default values repeated throughout the table:

brk	mode	repeat	dash	dot	style
1(GSHORTDASHED)	GDISCONTDASH	6.0	4.0	0.0	short dashed
2(GSHORTDOTTED)	GDISCONTDASH	4.0	0.8	0.0	short dotted
3(GSHORTCHAINED)	GDISCONTCHAIN	10.0	6.0	1.0	short chained
4(GLONGDASHED)	GDISCONTDASH	12.0	8.0	0.0	long dashed
5(GLONGDOTTED)	GDISCONTDASH	8.0	1.6	0.0	long dotted
6(GLONGDOTTED)	GDISCONTCHAIN	20.0	12.0	2.0	long chained
7(GDOTTED)	GDISCONTDASH	1.5	0.5	0.0	dotted
8	GDISCONTDASH	2.0	1.0	0.0	dotted
9	GDISCONTDASH	3.0	2.0	0.0	dashed
10	GDISCONTDASH	6.0	5.0	0.0	dashed
11	GDISCONTDASH	10.0	8.0	0.0	dashed
12	GDISCONTDASH	15.0	12.0	0.0	dashed
13	GDISCONTCHAIN	6.0	3.0	0.5	chained
14	GDISCONTCHAIN	8.0	5.0	0.5	chained
15	GDISCONTCHAIN	12.0	8.0	0.5	chained
16	GDISCONTCHAIN	16.0	12.0	1.0	chained

See Also Page 124
gSetBrokenLine
gSwitchBrokenLineStyles

gDefineFog

Syntax

C/C++:	**void gDefineFog**(int mode, int colour, ...);

F90:	**subroutine gDefineFog**(mode, colour, gStart, gEnd, gDensity) integer, intent(in) :: mode,colour real, optional, intent(in) :: gStart, gEnd, gDensity

Arguments

mode
Fog mode

= GNONE,	No fog
= GLINEAR,	Linear for depth-cueing
= GEXP1,	Exponential for cloud and heavy fog
= GEXP2,	Exponential for smoke and weather haze

colour
Fog colour (index or 24bit true colour value)

Optional Args

gStart
Start depth for linear fog (default = 0.0)

gEnd
End depth for linear fog (default = 3D viewport Z range)

gDensity
Fog density for exponential modes (default = 0.0025)

Description

The routine gDefineFog() specifies the current fog **mode** and **colour**. When switched on, in conjunction with depth buffering (see gSetShadingMode()), objects further away from the viewer are blended into the specified fog **colour** according to the specified **mode**.

The optional arguments **gStart** and **gEnd** are used for GLINEAR mode, where objects in front of **gStart** are displayed normally, objects between **gStart** and **gEnd** are linearly blended into the fog colour and objects behind **gEnd** are displayed totally in the fog colour. The default range of values for **gStart** and **gEnd** are 0.0 to the 3D viewport Z range which represents the eye-point to the furthest possible distance of the scene.

The GEXP1 and GEXP2 modes represent more realistic atmospheric fog modes, with the former using an exponential function of - (**gDensity** * viewing distance) and the latter using a function of - (**gDensity** * viewing distance) squared. Values in the range 0.5 to very small give realistic effects.

See Also

Page 337
gSetShadingMode
gSetViewport3D

gDefineGroupRange

Syntax

C/C++:	**void gDefineGroupRange**(int ngmin, int ngmax);

F90:	**subroutine gDefineGroupRange**(ngmin, ngmax) integer, intent(in) :: ngmin,ngmax

Arguments

ngmin
The beginning of the range of segment numbers to be used as groups numbers

ngmax
The end of the range of segment numbers to be used as groups numbers

Description

The routine gDefineGroupRange() defines a range of numbers to be used for segment groups and initializes the group table.

Subsequent calls to gDefineSegGroup() may not define any segment group outside the defined range. When a device is nominated the range is set from 1 to 32767.

If **ngmin** > **ngmax** an error message is output and no further action is taken.

See Also

Page 438
gDefineSegGroup

gDefineHatchStyle

Syntax

C/C++:	**void gDefineHatchStyle**(int fill, GHATSTY *rep);

F90:	**subroutine gDefineHatchStyle**(fill, rep) integer, intent(in) :: fill type (GHATSTY), intent(in) :: rep

Arguments

fill
Hatch style index

rep.pitch
Distance between hatch lines in current units

rep.angle
Orientation of hatch lines in degrees measured counterclockwise from the picture X axis

rep.xshift
Displacement of hatch pattern in current units in local X direction

rep.yshift
Displacement of hatch pattern in current units in direction 90° counterclockwise from local X direction

rep.xshear
Shear angle of hatch pattern in degrees with shear parallel to local X direction

rep.xhatch
Cross hatch switch

= GOFF,	Hatch in one direction only
= GON,	Hatch in two directions (cross hatch)

Description

GINO maintains a table of 256 hatch styles which are predefined. gDefineHatchStyle() allows them to be changed, giving complete control over the definition of hatch styles. **fill** points to the entry to be changed and therefore identifies a hatch style.

rep.pitch defines the spacing of hatch lines. Note that **rep.xshift**, **rep.yshift** and **rep.xshear** are defined with respect to local axes. The local axes are defined as the picture axes rotated through angle **rep.angle** and shifted by **rep.xshift** and **rep.yshift**. The local Y axis is sheared parallel to the local X axis. Hatch lines are generated parallel to the local X axis. Note, therefore, that **rep.yshift** is a displacement perpendicular to the hatch lines.

Cross hatching may be selected (**rep.xhatch**=GON). In this case a second set of hatch lines will be drawn across the first set parallel to the local Y axis, when filling an area. When a broken line type is specified with a continuous pattern in the routine gDefineBrokenLineStyle(), the second pattern is adjusted so that the dashes intersect. The effect of **rep.xshift** and **rep.xshear** is not apparent when single-hatching (**rep.xhatch**=GOFF) with solid lines. The absolute value of **rep.pitch** is used. GINO will not hatch with a spacing between lines that is less than the pen width. Therefore **rep.pitch**=0.0 specifies an efficient solid fill irrespective of the device.

See Also

Page 172
gDefineBrokenLineStyle

gDefineHLS

Syntax

C/C++:
```
void gDefineHLS(int col, float hue, float light, float sat);
```

F90:
```
subroutine gDefineHLS(col, hue, light, sat)

integer, intent(in) :: col
real, intent(in) :: hue,light,sat
```

Arguments

col
Colour index

< 0,	Dummy definition
= 0,	Background colour
> 0,	Index up to device capability

hue
Hue angle in degrees, 0.0 to 360.0

= 0.0,	Red

= 120.0,	Green
= 240.0,	Blue

light
Lightness, 0.0 (Black) to 1.0 (White)

sat
Saturation, 0.0 to 1.0

Description

The routine gDefineHLS() redefines the colour and intensity identified by **col** in terms of hue, lightness and saturation.

hue specifies the colour's position in the spectrum, **light** specifies the intensity of the colour on a linear scale from black (0.0) to white (1.0) and **sat** specifies the departure of the colour from grey. The most saturated colours occur when **light** is set to 0.5 and as **light** approaches either extreme all colours merge into black or white irrespective of the value of saturation. For zero saturation, lightness defines a grey-scale irrespective of hue. If **col** is set to zero, it identifies the background colour. gDefineHLS() has no effect if **col** is out of range for the device.

The action of gDefineHLS() depends on the colour capabilities of the device and whether gSetColourInfo() has been called. The default action is for gDefineHLS() to set a colour definition into a table indexed by **col** ready for being selected by the routine gSetLineColour(). If the device is operating in 24-bit direct-colour mode however, gDefineHLS() changes the values of a pseudo palette from which RGB values are extracted if colour indices are used in gSetLineColour().

light and **sat** are clipped to the range 0.0 to 1.0. If either value is clipped a warning message is output.

The HLS coordinates are converted to RGB before being transmitted to the device. Setting **col** less than zero stores the RGB values but does not transmit them to the device.

See Also

Page 216

gDefineHSV

Syntax

C/C++:
```
void gDefineHSV(int col, float hue, float sat, float value);
```

F90:
```
subroutine gDefineHSV(col, hue, sat, value)

integer, intent(in) :: col
real, intent(in) :: hue,sat,value
```

Arguments

col
Colour index

< 0,	Dummy definition
= 0,	Background colour
> 0,	Index up to device capability

hue
Hue angle in degrees, 0.0 to 360.0

= 0.0,	Red
= 120.0,	Green
= 240.0,	Blue

sat
Saturation, 0.0 to 1.0

value
Value, 0.0 to 1.0

Description

The routine gDefineHSV() redefines the colour and intensity identified by **col** in terms of hue, saturation and value.

hue specifies the colour's position in the spectrum, **sat** specifies the departure of the colour from grey, and **value** specifies the departure of the colour from black. For zero saturation, the value **value** defines a grey-scale, irrespective of hue. If **col** is set to zero, it identifies the background colour. gDefineHSV() has no effect if the device does not allow variations of colour or intensity or if **col** is out of range for the device.

The action of gDefineHSV() depends on the colour capabilities of the device and whether gSetColourInfo() has been called. The default action is for gDefineHSV() to set a colour definition into a table indexed by **col** ready for being selected by the routine gSetLineColour(). If the device is operating in 24-bit direct-colour mode however, gDefineHSV() changes the values of a pseudo palette from which RGB values are extracted if colour indices are used in gSetLineColour().

The values **sat** and **value** are clipped to the range 0.0 to 1.0. If either value is clipped a warning message is output.

The HSV coordinates are converted to RGB before being transmitted to the device. Setting **col** less than zero stores the RGB values but does not transmit them to the device.

See Also

Page 214
gSetColourInfo

gDefineLightSource

Syntax

C/C++:	**void gDefineLightSource**(int light, int colour, ...);

F90:

```
subroutine gDefineLightSource(light, colour, gDir, gAtten1, gAtten2, gPos,  gConc,
gSpread, gSpec)

integer, intent(in) :: light,colour

type (GPOINT3), optional, intent(in) :: gDir,gPos
real, optional, intent(in) :: gAtten1,gAtten2,gConc,gSpread
integer, optional, intent(in) :: gSpec
```

Arguments

light
Light source number (1-8)

colour
Light colour (index or 24bit true colour value)

Optional Args ***gDir***
Direction vector

gAtten1
Point light constant attenuation factor (default = 1.0)

gAtten2
Point light linear attenuation factor (default = 0.0)

gPos
Spot light position

gConc
Spot light concentration in range 0.0 to 100.0 (default =0.0)

gSpread
Spot light spread angle in range 0.0 to 360.0 (default = 180.0)

gSpec
Specular component colour (index or 24bit true colour value)

Description The routine gDefineLightSource() specifies the attributes of an individual light source, with its type depending on the number of optional arguments that are defined according to the following table:

Type	Obligatory	Optional
Ambient	light,colour	gSpec
Directional	light,colour,gDir	gSpec
Point Light	light,colour,gPos	gAtten1,gAtten2,gSpec
Spot Light	light,colour,gPos,gDir	gConc,gSpread,gSpec

When defining a directional light source the direction vector **gDir** points to the source of light, but when defining a spot light, the direction vector **gDir** defines the direction in which the light is shining.

Lights must be switched on to work (see gSetLightSwitch()).

See Also Page 329
gSetLightSwitch

gDefineLineStyle

Syntax

C/C++:	**void gDefineLineStyle**(int line, GLINSTY *rep);

F90:	**subroutine gDefineLineStyle**(line ,rep) integer, intent(in) :: line type (GLINSTY), intent(in) :: rep

Arguments

line
Line style index

= GCURRENT,	Current line style
= 1 - 256,	Stored line style

rep.vis
Line visibility (see gSetLineVis())

rep.brk
Broken line type (see gSetBrokenLine())

rep.col
Colour index (see gSetLineColour())

rep.width
Line width (see gSetLineWidth())

rep.type
Pen type (see gSetPenType())

rep.end
Line end (see gSetLineEnd())

Description

GINO keeps a table of line styles in which line attributes may be stored for later use. A call to gDefineLineStyle() allows definition of all attribute values for table entry **line** when **line** equals 1 to 256. When **line** is zero, the current line attributes are redefined. If **line** is out of range, an error message is output and no further action is taken.

The routine gDefineLineStyle() stores the absolute values of **rep.brk**, **rep.col**, **rep.width**, **rep.type**, **rep.end** and outputs a warning message for any negative values.

When a device is nominated (before any calls to gDefineLineStyle()) each table entry is set to the default for the current line attributes except for the colour index, which is set to the corresponding line style index (**rep.col** = **line**):

vis	brk	col	width	type	end
1(visible)	0(solid)	**line**	0.2mm (or device default)	0 (device default)	0(no ends)

See Also

Page 129
gSetLineColour
gSetLineEnd
gSetLineVis
gSetLineWidth
gSetPenType

gDefineMaterial

Syntax

C/C++:	**void gDefineMaterial**(int mat, GMATSTY *rep);

F90:	**subroutine gDefineMaterial**(mat, rep) integer, intent(in) :: mat type (GMATSTY), intent(in) :: rep

Arguments

mat
Material table index (1-256)

rep.ambient
Ambient reflection coefficient (0.0 -> 1.0)

rep.diffuse
Diffuse reflection coefficient (0.0 -> 1.0)

rep.specular
Specular reflection coefficient (0.0 -> 1.0)

rep.shine
Specular concentration (shininess) (%)

rep.trans
Translucence (filtering) - requires blending to be switched on (0.0 - 1.0)

Description

GINO keeps a table of material properties to be used in conjunction with material colours when setting the effect of lights on the surface of facets or objects. Up to 256 entries may be defined containing lighting coefficients and values. The current setting for the front and back faces of facets are set using the routine gSetMaterialIndex().

Initially, the first three entries of the table are set to the following values:

mat	ambient	diffuse	specular	shine	trans	Description
1	0.3	0.6	0.0	30.0	1.0	normal
2	0.3	0.6	1.0	30.0	1.0	plastic
3	0.3	0.6	1.0	100.0	1.0	shiny

The ambient, diffuse and specular coefficients are multiplied by the current colour values (as set by gSetMaterialColour()) to give the actual material properties of each face.

Translucence values less than 1.0 (opaque) are only utilized if surface blending is switched on by the gSetShadingMode() routine.

See Also

Page 340
gSetMaterialColour
gSetMaterialIndex
gSetShadingMode

gDefineNullChar

Syntax

C/C++:	**void gDefineNullChar**(int nul);

F90:	**subroutine gDefineNullChar**(nul) integer, intent(in) :: nul

Arguments

nul
Switch for representation of 0 (zero) character

= GNOSLASH,	Without / (default)
= GSLASH,	With /
= GTICK,	With tick ‘

Description

The routine gDefineNullChar() sets the required representation of the 0 (zero) character for the default GINO software font.

See Also

Page 152
gSetCharFont

gDefineParallelView

Syntax

C/C++:	**void gDefineParallelView**(float dx, float dy, float dz, float xcen, float ycen, float zcen);

F90:	**subroutine gDefineParallelView**(dx, dy, dz, xcen, ycen, zcen) real, intent(in) :: dx,dy,dz real, intent(in) :: xcen,ycen,zcen

Arguments

dx,dy,dz
Direction of viewing

xcen,ycen,zcen
Position of view centre in space coordinates

Description

The routine gDefineParallelView() defines a parallel view along the direction (**dx,dy,dz**).

The view plane is normal to this direction and cuts the line of sight at the view centre.

If all of **dx**, **dy** and **dz** are zero, an error message is output and no further action is taken.

See Also

Page 396

gDefinePerspView

Syntax

C/C++:	**void gDefinePerspView**(float xe, float ye, float ze, float dx, float dy, float dz, float dist);

F90:	**subroutine gDefinePerspView**(xe, ye, ze, dx, dy, dz, dist) real, intent(in) :: xe,ye,ze,dx,dy,dz,dist

Arguments

xe,ye,ze
Eye position in space coordinates

dx,dy,dz
Direction of viewing

dist
Perspective distance

Description

The routine gDefinePerspView() defines a perspective view from the eye position (**xe,ye,ze**) along the line specified by the direction vector (**dx,dy,dz**). The view plane (defined to be normal to this line) and the view centre are at a distance **dist** from the eye position. If the perspective distance **dist** or all of **dx**, **dy** and **dz** are zero, an error message is output and no further action is taken.

See Also

Page 393

gDefinePictureUnits

Syntax

C/C++:	**void gDefinePictureUnits**(float umils);

F90:	**subroutine gDefinePictureUnits**(umils) real, intent(in) :: umils

Arguments

umils
Number of millimetres which equal one current paper unit

Description

When a device is nominated, the default unit of measure is a millimetre. A call to gDefinePictureUnits() will set the current units so that one new unit of measure is equal to **umils** millimetres, e.g. a call to gDefinePictureUnits(25.4) sets the current paper units to inches. gDefinePictureUnits() must be called before the start of any graphical output. If **umils** is less than or equal to zero, an error message is output and the current units are not changed.

Users may, at any time, set up a viewport to map user coordinates of any range onto the current device or paper limits using the routine gSetViewport2D().

See Also

Page 44
gSetViewport2D

gDefinePixelPacking

Syntax

C/C++:	**void gDefinePixelPacking**(int nbp, int nrb, int npw, int ndir, int dir);

F90:	**subroutine gDefinePixelPacking**(nbp, nrb ,npw, ndir, dir) integer, intent(in) :: nbp,nrb,npw,ndir,dir

Arguments

nbp
The number of bits per pixel

nrb
The number of relevant bits

npw
The number of pixels per word

ndir
Pixel order within machine word

= + 1,	Normal direction
= -1,	Reverse direction

dir
Drawing direction

= 1,	Start top left, access horizontally (default)
= 2,	Start top left, access vertically
= 3,	Start top right, access horizontally
= 4,	Start top right, access vertically
= 5,	Start bottom left, access horizontally
= 6,	Start bottom left, access vertically
= 7,	Start bottom right, access horizontally
= 8,	Start bottom right, access vertically

Description

The routine gDefinePixelPacking() defines the users pixel data characteristics as used by gDrawPixelArea() and gGetPixelArea(). It defines the form of bit packing or unpacking between the users data storage and the actual device.

The bit characteristics are those of the integer data stored in the pixel array. In the case of gDrawPixelArea(), gDefinePixelPacking() defines the format of the pixel information that is already stored in the pixel array and is required to be unpacked and sent to the device. In the case of gGetPixelArea(), gDefinePixelPacking() defines the format that is required in the pixel array after packing the pixel data from the device.

Where more than one pixel is stored in a word, **ndir** specifies the order. The normal order is for the first pixel to be stored at the high end and the last pixel to be stored at the low end. If pixels are stored in the reverse order then **ndir** should be set negative.

dir specifies the order in which pixel data is stored in the pixel array. This does not effect packing but the order in which elements of the pixel array are accessed.

If **nbp**, **nrb**, **npw** or **ndir** are zero then no bit packing or unpacking is done and one full integer word is assumed to carry the information for one pixel on the device.

If any of **nbp**, **nrb**, **npw** are negative or **dir** is out or range a warning message is output and no change is made to the current pixel definition.

If the number of relevant bits (**nrb**) is greater that the number of bits per pixel (**nbp**) or the number of pixels per word (**npw**) * number of bits per pixel (**nbp**) is greater than 32 then a warning messages is output and no change is made to the current pixel definition.

See Also

Page 194
gDrawPixelArea
gGetPixelArea

gDefinePointWorkspace

Syntax

C/C++:	**void gDefinePointWorkspace**(int nw);

F90:	**subroutine gDefinePointWorkspace**(nw) integer, intent(in) :: nw

Arguments

nw

Number of real words to be set aside for storing internal point vertices.

= 0,	Delete point workspace
> 0,	Create or change size of point workspace

Description

The routine gDefinePointWorkspace() creates a workspace to store internal vertices generated by all the 2D and 3D drawing routines. Each vertex requires four real words. Allowance must be made for this amount of space when calling gSetWorkspaceLimit().

To change the size of the point workspace, gDefinePointWorkspace() may be called more than once. A call to gDefinePointWorkspace(0) deletes the point workspace and frees the space in the total workspace area. Any value less than zero will generate an error message.

Point storage is switch on and off using the routine gSetPointMode() and points are returned to the application using either gReturnInternalPoints2D() or gReturnInternalPoints3D().

See Also

Page 104, 291
gReturnInternalPoints2D
gReturnInternalPoints3D
gSetPointMode
gSetWorkspaceLimit

gDefinePolygonWorkspace

Syntax

C/C++:	**void gDefinePolygonWorkspace**(int nw);

F90:	**subroutine gDefinePolygonWorkspace**(nw) integer, intent(in) :: nw

Arguments

nw

Number of real words to be set aside for storing polygon vertices

= 0,	Delete polygon workspace
> 0,	Create or change size of polygon workspace

Description

The routine gDefinePolygonWorkspace() creates a workspace to store polygon vertices. Each vertex requires two real words and each polygon requires a header of eight real words.

Therefore if:

p = Number of polygons

v = Total number of polygon vertices to be stored

Then

nw = 2v+8p.

Allowance must be made for this amount of space when calling gSetWorkspaceLimit().

To change the size of the polygon workspace gDefinePolygonWorkspace() may be called more than once. A call to gDefinePolygonWorkspace(0) deletes the polygon workspace and frees the space in the total workspace area. Any value less than zero will generate an error message.

See Also

Page 104, 245, 291
gSetWorkspaceLimit

gDefineRGB

Syntax

C/C++:	**void gDefineRGB**(int col, float red, float green, float blue);

F90:	**subroutine gDefineRGB**(col, red, green, blue) integer, intent(in) :: col real, intent(in) :: red,green,blue

Arguments

col

Colour index

<0,	Dummy definition

= 0,	Background colour
> 0,	Index up to device capability

red
Red intensity, 0.0 to 1.0

green
Green intensity, 0.0 to 1.0

blue
Blue intensity, 0.0 to 1.0

Description

The routine gDefineRGB() redefines the colour and intensity identified by **col** in terms of red, green, blue intensities. **col**=0 identifies the background colour. gDefineRGB() has no effect if the device does not allow variations of colour or intensity or if **col** is out of range for the device.

red, **green** and **blue** are clipped to the range 0.0 to 1.0 If any value is clipped a warning message is output.

The action of gDefineRGB() depends on the colour capabilities of the device and whether gSetColourInfo() has been called. The default action is for gDefineRGB() to set a colour definition into a table indexed by **col** ready for being selected by the routine gSetLineColour(). If the device is operating in 24-bit direct-colour mode however, gDefineRGB() changes the values of a pseudo palette from which RGB values are extracted if colour indices are used in gSetLineColour().

Note that communication of colour values, to all devices, is done using the RGB system. All other colour coordinate systems are converted to the RGB system. Setting **col** <0 stores the **red**, **green** and **blue** values but does not transmit them to the device.

Devices with colour tables set the values in the table to a default set of colours at initialization (see gSetLineColour()), however gDefineRGB() can be used as a device qualifying routine (i.e. called before any drawing routine) to change the table before initialization and thus preventing the screen from flashing when changing from one colour to another.

See Also

Page 212
gEnqColourInfo
gSetColourInfo
gSetLineColour

gDefineSegGroup

Syntax

C/C++:	**void gDefineSegGroup**(int ngrp, int ngmin, int ngmax);

F90:	**subroutine gDefineSegGroup**(ngrp, ngmin, ngmax) integer, intent(in) :: ngrp,ngmin,ngmax

Arguments

ngrp
Segment group number

ngmin
Start segment number

ngmax
End segment number

Description

The routine gDefineSegGroup() defines a segment group. The segment group number **ngrp** must lie within the range defined by a call to gDefineGroupRange(). If gDefineGroupRange() has not been called, the range defaults to 1 to 32767.

A segment group consists of all the picture segments that fall within the range, **ngmin** to **ngmax**, which is defined by the call to gDefineSegGroup(). If **ngmin** or **ngmax** is out of range or if **ngmin** exceeds **ngmax**, an error message is output and no further action is taken.

Segment groups are maintained across device nomination. Up to 50 segment groups may be defined. gDefineSegGroup() redefines a segment group corresponding to **ngrp** that already exists. If there is no more space in the segment group table, an error message is output and no further action is taken.

See Also

Page 437
gDefineGroupRange

gDefineSegWorkspace

Syntax

C/C++:	**void gDefineSegWorkspace**(int nw);

F90:	**subroutine gDefineSegWorkspace**(nw) integer, intent(in) :: nw

Arguments

nw
Number of real words to be set aside for storing Software Display Files

= 0,	Delete software display file
>0,	Create or change size of display file

Description

If the software emulation of segment facilities is required the user may opt to store this information either in a scratch file or in a workspace area.

The routine gDefineSegWorkspace() both creates a workspace within the gSetWorkspaceLimit() workspace and switches storage of segments to that workspace.

As the gDefineSegWorkspace() workspace is part of the gSetWorkspaceLimit() workspace, gSetWorkspaceLimit() must be called before gDefineSegWorkspace() and include at least as much space as required by gDefineSegWorkspace().

To change the size of the software display file gDefineSegWorkspace() may be called more than once. A call to gDefineSegWorkspace(0) deletes the display file and frees the space in the total workspace area. Any value less than zero will generate an error message.

The gDefineSegWorkspace() workspace is maintained across device nominations along with any segment information it may contain. This allows segments to be created on one device and used on any subsequent device within the same program.

See Also Page 426
gSetWorkspaceLimit
gSetSegMode

gDefineSphericalView

Syntax

C/C++:	**void gDefineSphericalView**(float xcen, float ycen, float zcen, float rad, float dx, float dy, float dz, float dist);

F90:	**subroutine gDefineSphericalView**(xcen, ycen, zcen, rad, dx, dy, dz, dist) real, intent(in) :: xcen,ycen,zcen,rad real, intent(in) :: dx,dy,dz,dist

Arguments

xcen,ycen,zcen
Position of centre of sphere in space coordinates

rad
Radius of sphere in picture coordinates

dx,dy,dz
Direction of viewing

dist
Perspective distance

Description

The routine gDefineSphericalView() sets up a perspective view of the portion of the world enclosed by the specified sphere. The direction of viewing is defined by the vector (**dx,dy,dz**) directed towards the centre of the sphere. The viewpoint and position of the view plane are calculated so that the image of the sphere fits as closely as possible into the current window - i.e. the diameter of the sphere when projected onto the view plane equals the minimum window dimension. The window dimensions are defined by the viewport limits unless a user-defined window is currently specified (see gSetWindow2D(), gSetWindow3D() or gSetWindowMode()).

If either **dist** or **rad** is zero or all of **dx**, **dy** and **dz** are zero, an error message is output and no further action is taken.

See Also

Page 388
gSetWindow2D
gSetWindow3D
gSetWindowMode

gDefineTexture

Syntax

C/C++:	**void gDefineTexture**(int level, int xgrid, int ygrid, int border, int nbyte, int *pixbuf);

F90:	**subroutine gDefineTexture**(level, xgrid, ygrid, border, nbyte, pixbuf) integer, intent(in) :: level,xgrid,ygrid,nbyte,border,pixbuf(*)

Arguments

level
Texture map detail level

0 = base level
>0 = mipmap reduction level

xgrid, ygrid
Dimension of pixel array

border
Border switch

GOFF =	No border supplied
GON=	Border of 1 pixel on each side supplied with data

nbyte
Number of relevant bytes in texture map data (default = 3)

pixbuf
Pixel array used to define texture

Description

The routine gDefineTexture() defines the current texture map, of which only one can exist at any defined level.

The texture map detail level is set in **level**, where a value of zero implies that the base level (largest) texture map is being defined. Levels greater than zero can be used to define smaller mipmapped images for the same texture. Each level must be smaller in both directions by a power of 2 than the preceding level.

The dimensions of the texture map are defined in **xgrid** and **ygrid**. Both must be 2**n (or 2**n+2 if **border** is switched on) but not necessarily equal. The maximum size of a texture map is limited to the equivalent of 1024x1024 pixels.

The value of **nbyte** is in the range 1-4 with the following interpretation of data supplied in **pixbuf**

nbyte	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
1	1	-	-	Luminance
2	1	-	Alpha	Luminance
3(Default)	1	Red	Green	Blue
4	Alpha OR 1	Red	Green	Blue

The data supplied in **pixbuf** comprises **xgrid** * **ygrid** integer values in one of four forms; a) an array of colour index values with RGB settings being interpreted from the current GINO colour table, b) an array of 24bit packed RGB triplets as returned by the function gTrueCol, c) external image data read in by the routine gGetImageFile() or d) manually constructed data. For types a) - c) **nbyte** should be set to 3, for type d) bit 24 of all array values should be set to 1.

See Also

Page 346
gGetImageFile
gSetTextureMappingMode

gDeleteEventQueue

Syntax

C/C++:	**void gDeleteEventQueue**(void);
F90:	**subroutine gDeleteEventQueue**

Arguments

None

Description

On devices where events can be queued for processing by routine gWaitForEvent(), the queue may be deleted by calling gDeleteEventQueue(). All events waiting in the queue will be lost.

See Also

Page 456
gWaitForEvent

gDeleteSeg

Syntax

C/C++:	**void gDeleteSeg**(int nseg);
F90:	**subroutine gDeleteSeg**(nseg) integer, intent(in) :: nseg

Arguments

nseg

Picture segment or segment group number

> 0,	Segment specified by **nseg** is deleted from the display file
= 0,	The dustbin segment
= GALL,	All segments are deleted
< -1,	All segments except those specified by **nseg** are deleted from the display file

Description

The routine gDeleteSeg() is called to delete a picture segment.

It deletes the specified segment from the display file, which may involve redrawing the segment in the background colour in order to remove it from the device. If gDeleteSeg() is called while segment **nseg** is open, then a call to gCloseSeg() is automatically executed.

If gDeleteSeg(GALL) is called and segment 1 exists then it is deleted.

If gDeleteSeg(0) is called, the screen is cleared and any existing picture segments are redisplayed.

If **nseg** specifies a segment group, then all the members of that group are deleted.

See Also Page 427

gDisplayAsciiChar

Syntax

C/C++:	**void gDisplayAsciiChar**(int asc);

F90:	**subroutine gDisplayAsciiChar**(asc) integer, intent(in) :: asc

Arguments

asc
Represents the ASCII code of a character

Description

The routine gDisplayAsciiChar() outputs a single character whose ASCII code is **asc**.

The American Standard Code for Information Interchange (ASCII) is an integer numerical code (from 0 to 127). Each integer represents either an upper or lower case alphabetic character, digit, or special character.

The bottom left-hand corner of the character will be at the current drawing point. The end point is the bottom right-hand corner of that character.

The characters are drawn subject to the font representation, font weight, character size, orientation, italics and underline settings, and are transformed if GINO transformable characters are currently selected.

See Also

Page 137
gSetCharTransformMode
Appendix C - Character font tables

gDisplayInteger

Syntax

C/C++:	**void gDisplayInteger**(int number, int nwidth);

F90:	**subroutine gDisplayInteger(**number, nwidth) integer, intent(in) :: number,nwidth

Arguments

number
Integer value

nwidth
Field width

< 0,	Left-justified
= 0,	No output
> 0,	Right-justified

Description

The routine gDisplayInteger() outputs the integer value number as a decimal character string. The number starts with a minus sign for values less than zero.

If the number occupies less than **nwidth** character positions, the string is padded out with spaces. If the number is longer than **nwidth** characters, the string is filled with asterisks.

For positive values of **nwidth**, the number is right-justified. If **nwidth** is less than zero the number is left-justified. gDisplayInteger() does nothing if **nwidth** is zero.

Field width is limited to 32 character positions. If it exceeds this, it is set to 32 characters and a warning message is output.

The start point of the string is the current drawing position and will be at the bottom left-hand corner of the first character. The end point is the bottom right-hand corner of the string.

The characters are drawn subject to the font representation, font weight, character size, orientation, justification, italics and underline settings and are transformed if GINO transformable characters are currently selected.

See Also

Page 138

gDisplayReal

Syntax

C/C++:

```
void gDisplayRealExponent(float value, int nwidth, int nplace);
void gDisplayRealFixed(float value, int nwidth, int nplace);
void gDisplayRealFloat(float value, int nwidth);
```

F90:

```
subroutine gDisplayRealExponent(value, nwidth, nplace)
subroutine gDisplayRealFixed(value, nwidth, nplace)
subroutine gDisplayRealFloat(value, nwidth)

real, intent(in) :: value
integer, intent(in) :: nwidth,nplace
```

Arguments

value
Real value

nwidth
Field width

< 0,	Left-justified
= 0,	No output
> 0,	Right-justified

nplace
Number of places after the decimal point

Description

The gDisplayReal set of routines outputs the supplied real **value** as a character string in three different floating point formats. Either as a decimal floating-point, a decimal fixed point or a floating point character string. The start point of the string is the current drawing position which will be at the bottom left-hand corner of the numeric field. The end point is the bottom right-hand corner of the numeric field. The characters are drawn subject to the font representation, font weight, character size, orientation, justification, italics and underline settings and are transformed if GINO transformable characters are currently selected.

In all three routines, if the number occupies less than **nwidth** character positions, the string is padded out with spaces. On the other hand, if the number is longer than **nwidth** characters, a string of asterisks is output. For positive values of **nwidth**, the number is right-justified. If **nwidth** is less than zero, the number is left-justified. The field width is limited to 32 character positions. If it exceeds this, it is set to 32 characters and a warning message is output.

In the case of gDisplayRealExponent(), the number consists of a mantissa followed by a decimal exponent. The mantissa consists of a decimal point (preceded by a minus sign if **value** is less than zero) followed by the number of digits specified by **nplace**. If **nplace** is less than zero, its absolute value is used and a warning message is output. The number is rounded in the last decimal place. A zero is output in front of the decimal point if the number is positive and left-justified or if the number is right-justified and there is room for a leading zero. The exponent occupies 4 character positions. The first character is the letter ‘E’ and the remaining characters contain the exponent value, right-justified and preceded by a minus sign for negative exponents. Any gap after the letter ‘E’ is padded out with spaces. If the exponent is too large, the string is filled with asterisks.

In the case of gDisplayRealFixed() the number consists of an integer part (preceded by a minus sign if **value** is less than zero) followed by a decimal point and a fractional part. The number of digits for the fractional part is specified by **nplace**. If **nplace** is less than zero, its absolute value is used and a warning message is output. The number is rounded in the last decimal place. If the integer part is zero, a leading zero is output unless **value** is less than zero and there is only room for the minus sign before the decimal point.

In the case of gDisplayRealFloat() the mantissa consists of a decimal point (preceded by a minus sign if **value** is less than zero) followed by (**nwidth**-6) digits, up to a maximum of 6 digits. If the field width is less than 6, the string is filled with asterisks. The number is rounded in the last decimal place. A zero is output in front of the decimal point if the number is positive and left-justified or if the number is right-justified and there is room for a leading zero. The exponent occupies 4 character positions. The first character is the letter ‘E’ and the remaining characters contain the exponent value, right-justified and preceded by a minus sign for negative exponents. Any gap after the letter ‘E’ is padded with spaces. If the exponent is too large, the string is filled with asterisks.

See Also

Page 138

gDisplayStr

Syntax

C/C++:	**void gDisplayStr**(char string[]);

F90:	**subroutine gDisplayStr**(string) character*(*), intent(in) :: string

Arguments

string
Any character argument (except array name)

Description

The routine gDisplayStr() outputs a character string consisting of any combination of characters in the ASCII set and any special characters permitted by the system. Strings should be terminated by *. otherwise the number of characters is limited to the total length of **string** up to a maximum of 255 characters.

The following escape sequences offer control of various functions:

*L or *l- shift into lower case

*U or *u- shift into upper case

*Fnnn or *fnnn temporary change to font nnn where nnn = 000 to 199 (see gSetCharFont())

*N or *n- move to next line (see gStartTextBlock(), gMoveToNextLine())

*E or *e- activate current exponent position and size settings (see gSetStrExponent())

*I or *i- activate current index position and size settings (see gSetStrExponent())

*O or *o- position next character over previous character at exponent size setting

*S or *s- start underscore (see gSetStrUnderscore())

*W or *w- increase current font weight by 3 (see gSetFontWeight())

*\- set italic angle to -15 degrees (see gSetItalicAngle())

*|- reset italic angle to that on entry

*/- set italic angle to +15 degrees (see gSetItalicAngle())

*A or *a- set current position on base line as an alignment position
(used to switch exponent, index positioning and character size off and
also resets underline and weight)

*B or *b- move back to last align position on base line

*:- displays umlaut form of character if followed by aouAOU

*:S- displays German sz character if available in current font

** means output *

The escape character * can be set to any other character by the gSetEscapeChar() routine.

The characters are drawn using the current font with its weight and representation set by the most recent calls to gSetFontWeight() and gSetFontForm() respectively. If the *F escape sequence is used this will make a temporary change to the current font for the output of this string only. Two further facilities are available for controlling the font.

*FS sets the GINO font to the temporary string font.

*FR restores the font which was current when the string routine was called.

If *N is used, the routine gStartTextBlock() must have been called to set up a text block start position, otherwise a warning message is output and the next line is positioned below the end of the previous string.

Multiple use of *E or *I adjusts the position above or below the base line for each occurrence.

The start and end position and direction of the string are determined by the current setting of character justification (see gSetStrJustify()).

Hardware or software characters are output depending on the current setting of gSetHardChars(), gSetMixedChars() or gSetSoftChars() and they are drawn subject to the current size, orientation, italics and underline. These settings may not be exactly matched for hardware characters. If GINO transformable characters are currently selected (using gSetCharTransformMode()), the string will be output using vectors, in the current line style subject to the current transformation and viewing setting.

All character codes outside the range 0 to 255 are output as ASCII code 47, a '/'. In addition, if software characters are being output then all character codes outside the range 25-127 are also output as a '/'. The exception being the codes representing the a,o, u umlaut and beta characters according to the characters code of the current implementation (DOS or ISO). Thus the following ASCII codes are mapped onto the special characters provided in several GINO software fonts:

DOS	ISO	Char
132	228	ä
148	246	ö
129	252	ü
225	223	ß
142	196	Ä
153	214	Ö
154	220	Ü

See Also

Page 138
gPrintf
gMoveToNextLine
gSetStrExponent
gSetCharFont
gSetStrJustify
gSetStrUnderscore
gSetFontForm
gSetFontWeight
gSetItalicAngle
gSetEscapeChar
gSetCharTransformMode

gDisplayStrPolyline

Syntax

C/C++:	**void gDisplayStrPolylineBy2D**(int npts, GPOINT *points, char string[]); **void gDisplayStrPolylineTo2D**(int npts, GPOINT *points, char string[]);

F90:	**subroutine gDisplayStrPolylineBy2D**(npts, points, string) **subroutine gDisplayStrPolylineTo2D**(npts, points, string) integer, intent(in) :: npts type (GPOINT), intent(in) :: points(*) character*(*), intent(in) :: string

Arguments

npts
Number of vector increments specified

points
Array of points specifying a series of either relative or absolute 2D positions, which define a polyline along which the string will be drawn

string
Character string

Description

The routines gDisplayStrPolylineBy2D() and gDisplayStrPolylineTo2D() display a character string along a polyline represented by the points contained in the array **points** by adjusting the angle of each character in the string. The polyline should be defined as a series of either relative or absolute points depending on the routine used. The relative points used by gDisplayStrPolylineBy2D() start at the current position and pass through **npts** vector increments.

Strings should be terminated by *. otherwise the number of characters is limited to the total length of **string** up to a maximum of 256 characters. For example, if string is declared as char [10] then the string will always be terminated after 10 characters, if no intervening *. is encountered.

The characters are drawn subject to the font representation, font weight, character size, italics and underline settings. If transformations are switched on the string is output using transformed characters so that the string follows the transformed polyline.

The string is justified on the polyline according to the current setting of gSetStrJustify(). The character string may contain any of the GINO escape sequences described under gDisplayStr(). If the length of the character string is longer than the length of the polyline, the string is truncated.

See Also

Page 159
gSetStrJustify
gDisplayStr

gDragSeg

Syntax

C/C++:	**void gDragSeg**(int nseg);

F90:	**subroutine gDragSeg**(nseg) integer, intent(in) :: nseg

Arguments

nseg
Picture segment number or segment group number

> 0,	Drag picture segment or segment group **nseg**
= GALL,	Drag all picture segments
< -1,	Drag all segments except segment **nseg** or segment group **nseg**

Description

The routine gDragSeg() causes the specified picture segment to follow the position of the cursor until a key is pressed. The position of the picture segment anchor and the integer identifier of the key are obtained by calling the routine gGetEventRecord().

Where software emulation of segments is being performed, the cursor is positioned at the segment origin (if possible) and the user can move the cursor to the required position. The segment does not follow the cursor but is erased from its current position and redrawn at the new position when the cursor key is pressed.

See Also

Page 440, 456
gGetEventRecord

gDrawAkima

Syntax

C/C++:	**void gDrawAkimaBy2D**(int npts, GPOINT *points, int beg, int fin); **void gDrawAkimaTo2D**(int npts, GPOINT *points, int beg, int fin);

F90:	**subroutine gDrawAkimaBy2D**(npts, points, beg, fin) **subroutine gDrawAkimaTo2D**(npts, points, beg, fin) integer, intent(in) :: npts,beg,fin type (GPOINT), intent(in) :: points(*)

Arguments

npts
Number of points

points
Array specifying series of relative or absolute points through which the curve is drawn

beg
End conditions for start of curve

= GXPOINT,	Direction of curve calculated using extra point
= GNONE,	No end conditions
= GANGLE,	Direction of curve defined by angle

fin
End conditions for end of curve (as for start of curve)

Description

The routines gDrawAkimaBy2D() and gDrawAkimaTo2D() draw a piecewise smooth cubic curve through the specified number of relative or absolute coordinate points. The matching end slopes of the curve are computed using an averaging method due to Akima. In the case of gDrawAkimaBy2D(), the curve starts at the current drawing position and is drawn through **npts** series of vector increments. In the case of gDrawAkimaTo2D() the curve starts at the first element in **points** and is drawn through the intervening **npts** points. The number of sub-increments is controlled by the arc routine gSetArcIncrement().

The start and end directions of the curve can be separately controlled by setting **beg** and **fin** to non-zero values. If **beg** or **fin** is set to GANGLE, the direction of the curve is specified directly in terms of cosine and sine values. If **beg** or **fin** is set to GXPOINT, the direction of the curve is calculated so that the curve would pass through an extra specified point as if it was extended to include that point. These angles and/or extra points defining the curve end conditions are set up using gSetCurveAttribs2D(). Note that, the curve drawing routines update the current curve end conditions so that the start and end angles match those of the curve that has been drawn. The current curve end conditions can be enquired by calling gEnqCurveAttribs2D().

See Also

Page 93, 285
gEnqCurveAttribs2D
gSetArcIncrement
gSetCurveAttribs2D

gDrawArc

Syntax

C/C++:

void gDrawArcBy2D(float dxc, float dyc, float dxe, float dye, int sense);
void gDrawArcBy3D(float dxc, float dyc, float dzc,float dxe, float dye, float dze,float dxt, float dyt, float dzt);
void gDrawArcTo2D(float xc, float yc, float xe, float ye, int sense);
void gDrawArcTo3D(float xc, float yc, float zc,float xe, float ye, float ze, float dxt, float dyt, float dzt);

F90:

subroutine gDrawArcBy2D(dxc, dyc, dxe, dye, sense)
subroutine gDrawArcBy3D(dxc, dyc, dzc, dxe, dye, dze, dxt, dyt, dzt)
subroutine gDrawArcTo2D(xc, yc, xe, ye, sense)
subroutine gDrawArcTo3D(xc, yc, zc, xe, ye, ze, dxt, dyt, dzt)

real, intent(in) :: dxc,dyc,dzc,dxe,dye,dze,dxt,dyt,dzt
real, intent(in) :: xc,yc,zc,xe,ye,ze
integer, intent(in) :: sense

Arguments

dxc,dyc,dzc,xc,yc,zc
Position of the arc centre defined as either relative to the current drawing position or an absolute position

dxe,dye,dze,xe,ye,ze
A relative or absolute position defining a vector from the arc's centre on which the arc will terminate

sense
Direction of drawing

= GANTICLOCKWISE,	The arc is drawn positively
= GCLOCKWISE,	The arc is drawn negatively

dxt,dyt,dzt
Direction vector defining the starting direction or plane of a 3D arc

Description

These routines draw a two or three dimensional circular arc using relative or absolute coordinates to define its centre and end positions. Where relative positions are given, these are relative to the current drawing position.

The drawing of all arcs start at the current drawing position and have a radius equal to the distance from the start to the specified centre position. The arc is terminated at a point that lies on a line from the specified arc centre through the specified end point.

For 2D arcs, the direction of drawing is given by **sense**, but for 3D arcs, the start direction is indicated by the direction vector (**dxt,dyt,dzt**) . If the start, centre and end points are collinear the direction vector gives the plane of the arc, otherwise it merely indicates whether the major or minor arc is required. If the direction vector is parallel to this same line, an error message is output and no arc is output.

If transformations are being used, it is essential that the current drawing position is set by a routine using absolute coordinates (e.g. gMoveTo2D/3D()) after the transformation is changed and before gDrawArcTo2D/3D() is called.

If the output device cannot support hardware arcs or if software arcs have been requested by calling gSetArcMode(), the arc is drawn as a series of straight vectors. However, sufficient vectors are provided to give the curve a smooth appearance.

See Also

Page 86
Page 284
gSetArcMode

gDrawBezier

Syntax

C/C++:

void gDrawBezierBy2D(int npts, GPOINT *points2);
void gDrawBezierBy3D(int npts, GPOINT3 *points3);
void gDrawBeziero2D(int npts, GPOINT *points2);
void gDrawBezierTo3D(int npts, GPOINT3 *points3);

F90:

subroutine gDrawBezierBy2D(npts, points2)
subroutine gDrawBezierBy3D(npts, points3)
subroutine gDrawBezierTo2D(npts, points2)
subroutine gDrawBezierTo3D(npts, points3)

integer, intent(in) :: npts
type (GPOINT), intent(in) :: points2(*)
type (GPOINT3), intent(in) :: points3(*)

Arguments

npts
Number of coordinate points specified

points2, points3
Array specifying series of relative or absolute points in space coordinates through which the curve is drawn

Description

The routines gDrawBezierBy2D(), gDrawBezierBy3D(), gDrawBezierTo2D()and gDrawBezierTo3D() draw a smooth curve through the specified number of relative or absolute coordinate points. In the case of gDrawBezierBy2D() and gDrawBezierBy3D(), the curve starts at the current drawing position and is drawn through **npts** series of vector increments. In the case of gDrawBezierTo2D() and gDrawBezierTo3D() the curve starts at the first element in **points** and is drawn through the intervening **npts** points.

The number of sub-increments is controlled by the arc routine gSetArcIncrement(), where the total number of straight-line segments used to draw the curve is defined as the number of data points supplied * number of increments. At least 2 points are required to define a Bezier curve.

See Also

Page 101
Page 290
gSetArcIncrement

gDrawBezierSphere

Syntax

C/C++:	**void gDrawBezierSphere**(float xp, float yp, float zp, float radius, ...);

F90:	**subroutine gDrawBezierSphere**(xp, yp, zp, radius, gURot, gVRot, gWRot, gUComp, gVComp) real, intent(in) :: xp,yp,zp,radius real, optional, intent(in) :: gURot, gVRot, gWRot integer, optional, intent(in) :: gUComp, gVComp

Arguments

xp, yp, zp
Origin (centre) of sphere

radius
Radius of sphere

Optional Args

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gUComp ,gVComp
Optional object complexity (default 10 x 10)

Description

The routine gDrawBezierSphere() draws a 'solid' sphere primitive constructed of outward facing facets according to the specified complexity. The sphere is generated from eight Bezier patches centred at the origin (**xp**, **yp**, **zp**), with the defined **radius**.

The arguments **gURot**, **gVRot**, **gWRot** specify optional rotations about the object's local axes system. Whilst these will not alter the visual appearance of a coloured sphere, it will affect the appearance of a textured sphere as the texture origin will lie in a different position with relation to the global axes.

The arguments **gUComp** and **gVComp** define the object's complexity in its U (circumference) and V (height) axes respectively. These values determine the number of divisions (facets) in either direction and therefore, the objects smoothness.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 323
gDrawFacet

gDrawBezierSurface

Syntax

C/C++: **void gDrawBezierSurface**(int nx, int ny, GPOINT3 *mesh, ...);

F90: **subroutine gDrawBezierSurface**(nx, ny, mesh, gUComp, gVComp)

```
integer, intent(in) :: nx,ny
type (GPOINT3), intent(in) :: mesh(nx,ny)

integer, optional, intent(in) :: gUComp, gVComp
```

Arguments

nx, ny
Number of points in mesh

mesh
Two dimensions array of 3D mesh points

Optional Args

gUComp ,gVComp
Optional object complexity (default 10 x 10)

Description

The routine gDrawBezierSurface() draws a surface based on an interpolation of the supplied mesh.

The arguments **gUComp** and **gVComp** define the object's complexity in its U and V axes respectively. These values determine the number of divisions (facets) in either direction and therefore the object's smoothness.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 317
gDrawFacet

gDrawBezierVolume

Syntax

C/C++: **void gDrawBezierVolume**(float xp, float yp, float zp, int npts, GPOINT *points, ...);

F90: **subroutine gDrawBezierVolume**(xp, yp, zp, npts, points, gURot, gVRot, gWRot, gUComp, gVComp)

```
real, intent(in) :: xp,yp,zp
integer, intent(in) :: npts
type (GPOINT), intent(in) :: points(*)

real, optional, intent(in) :: gURot, gVRot, gWRot
integer, optional, intent(in) :: gUComp,gVComp
```

Arguments

xp, yp, zp
Origin (bottom, centre) of volume

npts
Number of control points in outline

points
Array containing two dimensional outline control points

Optional Args

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gUComp, gVComp
Optional object complexity (default 10)

Description

The routine gDrawBezierVolume() draws a 'solid' volume of rotation constructed of outward facing facets according to the specified complexity. The volume is generated by rotating an interpolated curve, based on the supplied control points contained in the **points** array, about a vertical axis which passes through the origin (**xp**, **yp**, **zp**).

An alternative orientation can be specified using either up to three local axes rotations (**gURot**, **gVRot**, **gWRot**) or by using an absolute/relative vector, **gVVec** from the object's origin. In the latter case a local rotation about the object's vertical axis can also be added in **gVRot**.

The arguments **gUComp**and **gVComp** define the object's complexity about its U and V axes. This values determine the number of divisions (facets) in these directions and therefore, the objects smoothness.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 324
gDrawFacet

gDrawBox

Syntax

C/C++: **void gDrawBox**(float xp, float yp, float zp, ...);

F90: **subroutine gDrawBox**(xp, yp, zp, gUDim, gVDim, gWDim, gURot, gVRot, gWRot, gUVec, gVVec, gWVec, gAbs, gUComp, gVComp, gWComp)

```
real, intent(in) :: xp,yp,zp

real, optional, intent(in) :: gUDim, gVDim, gWDim
real, optional, intent(in) :: gURot, gVRot, gWRot
type (GPOINT3), optional, intent(in) :: gUVec, gVVec, gWVec
integer, optional, intent(in) :: gAbs
integer, optional, intent(in) :: gUComp, gVComp, gWComp
```

Arguments

xp, yp, zp
Origin (bottom, left, back corner) of box

Optional Args

gUDim, gVDim, gWDim
Optional box dimensions (default 1.0,1.0,1.0)

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gUVec, gVVec, gWVec
Optional edge vectors

gAbs
Optional edge vector direction flag

= GABSOLUTE,	Absolute edge vectors
= GRELATIVE,	Relative edge vectors

gUComp ,gVComp, gWComp
Optional object complexity (default 1 x 1 x 1)

Description

The routine gDrawBox() draws a 'solid' box primitive of the specified dimensions by generating outward facing facets for each face. The box is positioned at the specified origin (**xp**, **yp**, **zp**), with a default unit dimension extending in a positive direction along each of its three local axes.

An alternative dimension/orientation can be specified using either a) up to three dimension values (**gUDim**, **gVDim**, **gWDim**) and up to three local axes rotations (**gURot**, **gVRot**, **gWRot**) or b) as three mutually perpendicular edge vectors (**gUVec**, **gVVec**, **gWVec**) which may be absolute or relative to the object's origin.

By default, each face is constructed from a single facet, but using the optional arguments **gUComp**, **gVComp** and/or **gWComp** the two faces in each of the three local axes may be sub-divided into multiple facets for greater accuracy of lighting and texturing. Thus if **gUComp** is set to 2, then the two faces along the X/U axes of the box will be divided into 2x2 (4) facets.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 307
gDrawFacet

gDrawCellArray

Syntax

C/C++:	**void gDrawCellArray**(float x1, float y1, float x2, float y2, int npixx, int npixy, int isx, int isy, int idx, int idy, int pixbuf[]);

F90:	**subroutine gDrawCellArray**(x1, y1, x2, y2, npixx, npixy, isx, isy, idx, idy, pixbuf) real, intent(in) :: x1,y1,x2,y2 integer, intent(in) :: npixx,npixy,isx,isy,idx,idy,pixbuf(*)

Arguments

x1,y1
Bottom-left corner of image in current units

x2,y2
Top-right corner of image in current units

npixx,npixy
Dimension of image data array

isx,isy
The start X position and Y position of a sub-array

idx,idy
The X and Y dimensions of a sub-array

pixbuf
The array containing image colour information

Description

The routine gDrawCellArray() draws a scaled image to fit the transformed corner points which are specified in current drawing units. The final image is drawn in a rectangle with edges parallel to the drawing area as represented by the transformed points.

Image data is passed through a pointer to an integer array **pixbuf** dimensioned (**npixx**,**npixy**). Where the whole of this array is to be displayed, the user should set **isx** and **isy** to 1 and **idx** and **idy** to be the same as **npixx** and **npixy**. Alternatively a portion of the array can be displayed (still at the anchor position **ix**,**iy**) by setting the values of **isx** and **isy** to the offsets from the start of the **pixbuf** and **idx**,**idy** to the dimensions of the sub-array.

The image data may consist of packed or unpacked colour indices or in the case of true colour devices, 24bit true colour RGB values.

See Also

Page 191
gDefinePixelPacking
gSetColourInfo

gDrawCone

Syntax

C/C++: **void gDrawCone**(float xp, float yp, float zp, float rad1, float rad2, ...);

F90: **subroutine gDrawCone**(xp, yp, zp, rad1, rad2, gHeight, gURot, gVRot, gWRot, gVVec, gAbs, gUComp, gVComp)

```
real, intent(in) :: xp,yp,zp,rad1,rad2

real, optional, intent(in) :: gHeight
real, optional, intent(in) :: gURot, gVRot, gWRot
type (GPOINT3), optional, intent(in) :: gVVec
integer, optional, intent(in) :: gAbs
integer, optional, intent(in) :: gUComp, gVComp
```

Arguments

xp, yp, zp
Origin (bottom, centre) of cone

rad1, rad2
Bottom and top radii of cone

Optional Args

gHeight
Optional cone height (default 1.0)

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gVVec
Optional heigh/orientation vector

gAbs
Optional height/orientation vector direction flag

= GABSOLUTE,	Absolute height/orientation vector
= GRELATIVE,	Relative height/orientation vector

gUComp ,gVComp
Optional object complexity (default 10 x 10)

Description

The routine gDrawCone() draws a 'solid' cone primitive constructed of outward facing facets according to the specified complexity. The cone is centred at the origin (**xp**, **yp**, **zp**), with the defined bottom and top radii (**rad1** and **rad2**) and a default unit height extending in a positive direction along the local V (vertical) axes.

An alternative height/orientation can be specified using either a height dimension (in **gHeight**) and up to three local axes rotations (**gURot**, **gVRot**, **gWRot**) or an absolute/relative vector, **gVVec** from the base of the object. In the latter case a local rotation about the object's vertical axis can also be added in **gVRot**.

The arguments **gUComp** and **gVComp** define the object's complexity in its U (circumference) and V (height) axes respectively. These values determine the number of divisions (facets) in either direction and in the case of the circumferential value, the objects smoothness. Setting **gUComp** to 6 will define a 6 sided cone for example.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 309
gDrawFacet

gDrawCube

Syntax

C/C++:
```
void gDrawCube(float xp, float yp, float zp, float dim, ...);
```

F90:
```
subroutine gDrawCube(xp, yp, zp, dim ,gURot, gVRot, gWRot)
real, intent(in) :: xp,yp,zp,dim

real,optional, intent(in) :: gURot, gVRot, gWRot
integer, optional, intent(in) :: gUComp, gVComp, gWComp
```

Arguments

xp, yp, zp
Origin (bottom, left, back corner) of cube

dim
Cube dimension

Optional Args ***gURot ,gVRot, gWRot***
Optional rotations about local axes (default 0.0,0.0,0.0)

gUComp ,gVComp, gWComp
Optional object complexity (default 1 x 1 x 1)

Description The routine gDrawCube() draws a 'solid' cube primitive of the specified dimension by generating six equally sized facets for each face. The box is positioned at the specified origin (**xp**, **yp**, **zp**) extending in a positive direction along each of its three local axes.

The arguments **gURot**, **gVRot**, **gWRot** specify optional rotations about the object's local axes system.

By default, each face is constructed from a single facet, but using the optional arguments **gUComp**, **gVComp** and/or **gWComp** the two faces in each of the three local axes may be sub-divided into multiple facets for greater accuracy of lighting and texturing. Thus if **gUComp** is set to 2, then the two faces along the X/U axes of the cube will be divided into 2x2 (4) facets.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also Page 307
gDrawFacet

gDrawCurve

Syntax

C/C++:
```
void gDrawCurveBy2D(int npts, GPOINT *points, int beg, int fin);
void gDrawCurveTo2D(int npts, GPOINT *points, int beg, int fin);
```

F90:
```
subroutine gDrawCurveBy2D(npts, points, beg, fin)
subroutine gDrawCurveTo2D(npts, points, beg, fin)

integer, intent(in) :: npts,beg,fin
type (GPOINT), intent(in) :: points(*)
```

Arguments

npts
Number of points

points
Array specifying series of relative or absolute points through which the curve is drawn

beg
End conditions for start of curve

= GXPOINT,	Direction of curve calculated using extra point
= GNONE,	No end conditions
= GANGLE,	Direction of curve defined by angle

fin
End conditions for end of curve (as for start of curve)

Description

The routines gDrawCurveBy2D() and gDrawCurveTo2D() draw a smooth curve through the specified number of relative or absolute coordinate points. In the case of gDrawCurveBy2D(), the curve starts at the current drawing position and is drawn through **npts** series of vector increments. In the case of gDrawCurveTo2D() the curve starts at the first element in **points** and is drawn through the intervening **npts** points. The number of sub-increments is controlled by the current arc tolerance as set by gSetArcTolerance().

The start and end directions of the curve can be separately controlled by setting **beg** and **fin** to non-zero values. If **beg** or **fin** is set to GANGLE, the direction of the curve is specified directly in terms of cosine and sine values. If **beg** or **fin** is set to GXPOINT, the direction of the curve is calculated so that the curve would pass through an extra specified point as if it was extended to include that point. These angles and/or extra points defining the curve end conditions are set up using gSetCurveAttribs2D(). Note that, the curve drawing routines update the current curve end conditions so that the start and end angles match those of the curve that has been drawn. The current curve end conditions can be enquired by calling gEnqCurveAttribs2D().

See Also

Page 93
gEnqCurveAttribs2D
gSetArcTolerance
gSetCurveAttribs2D

gDrawCylinder

Syntax

C/C++: **void gDrawCylinde**r(float xp, float yp, float zp, float radius, ...);

F90: **subroutine gDrawCylinder**(xp, yp, zp, radius, gHeight, gURot, gVRot, gWRot, gVVec, gAbs, gUComp, gVComp)

```
real, intent(in) :: xp,yp,zp,radius

real, optional, intent(in) :: gHeight
real, optional, intent(in) :: gURot, gVRot, gWRot
type (GPOINT3), optional, intent(in) :: gVVec
integer, optional, intent(in) :: gAbs
integer, optional, intent(in) :: gUComp, gVComp
```

Arguments

xp, yp, zp
Origin (bottom, centre) of cylinder

radius
Radius of cylinder

Optional Args

gHeight
Optional cylinder height (default 1.0)

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gVVec
Optional height vector

gAbs
Optional height vector direction flag

= GABSOLUTE,	Absolute height vector
= GRELATIVE,	Relative height vector

gUComp ,gVComp
Optional object complexity (default 10 x 10)

Description

The routine gDrawCylinder() draws a 'solid' cylinder primitive constructed of outward facing facets according to the specified complexity. The cylinder is centred at the origin (**xp**, **yp**, **zp**), with the defined **radius** and a default unit height extending in a positive direction along the local V (vertical) axes.

An alternative height/orientation can be specified using either a height dimension (in **gHeight**) and up to three local axes rotations (**gURot**, **gVRot**, **gWRot**) or an absolute/relative vector, **gVVec** from the base of the object. In the latter case a local rotation about the object's vertical axis can also be added in **gVRot**.

The arguments **gUComp** and **gVComp** define the object's complexity in its U (circumference) and V (height) axes respectively. These values determine the number of divisions (facets) in either direction and in the case of the circumferential value, the objects smoothness. Setting **gUComp** to 6 will define a 6 sided cylinder for example.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 309
gDrawFacet

gDrawFacet

Syntax

C/C++:
```
void gDrawFacet(int npts, GPOINT3 *points, ...);
```

F90:
```
subroutine gDrawFacet(npts, points, gNormals, gTextCoords, gColours)

integer, intent(in) :: npts
type (GPOINT3), intent(in) :: points(*)

type (GPOINT3), optional, intent(in) :: gNormals(*), gTextCoords(*)
integer, optional, intent(in) :: gColours(*)
```

Arguments

npts
Number of vertices in facet

points
Array specifying series of absolute points that define the facet boundary

Optional Args

gNormals
Optional array specifying normal vectors at each vertex

gTextCoords
Optional array specifying texture coordinate values at each vertex

gColours
Optional integer array specifying colours at each vertex

Description

The routine gDrawFacet() displays a single facet (polygon) containing **npts** vertices held in the array **points** of type GPOINT3. The facet is drawn according to the current modelling and/or viewing transformations subject to the current facet offset as set by gSetFacetOffsetMode(). The facet is drawn either solid or as a boundary according to the current facet filling style as set by gSetFacetFillStyle().

The facet colour is determined by the current material property, which may be set (using gSetMaterialIndex()) to match the current line drawing colour (gSetLineColour()), or according to specific colour attributes that are affected by the current lighting conditions. Alternatively if the optional **gColours** array is used, the facet is coloured using graduations of the colours specified at each vertex in this array (these are not affected by any lighting conditions).

The optional arrays **gNormals** and **gTextCoords** can be used to specify non-planar normals and/or texture coordinates when creating smooth or texture mapped surfaces. Both arrays are of type GPOINT3.

See Also

Page 296
gSetFacetFillStyle
gSetFacetOffsetMode
gSetLineColour
gSetMaterialIndex

gDrawLine

Syntax

C/C++:

```
void gDrawLineBy2D(float dx, float dy);
void gDrawLineBy3D(float dx, float dy, float dz);
void gDrawLineTo2D(float x, float y);
void gDrawLineTo3D(float x, float y, float z);
```

F90:

```
subroutine gDrawLineBy2D(dx, dy)
subroutine gDrawLineBy3D(dx, dy, dz)
subroutine gDrawLineTo2D(x, y)
subroutine gDrawLineTo3D(x, y, z)

real, intent(in) :: dx,dy,dz
real, intent(in) :: x,y,z
```

Arguments

dx,dy,dz
Coordinate increments (in the current units) from the current drawing position to the end point

x,y,z
The absolute coordinates (with respect to the current axes and in the current units) of the end point

Description The gDrawLine set of routines draws a straight line from the current drawing position to the specified end point using relative or absolute coordinates. Both 2D and 3D routines are provided.

Where a mixture of 2D and 3D routines are used, the current Z coordinate is assumed to be constant during the drawing of a 2D vector (the default being 0.0).

The current drawing position is updated to be the end of the line drawn by these routines.

See Also Page 80
Page 280

gDrawMarker

Syntax

C/C++:	**void gDrawMarker**(int nsym);

F90:	**subroutine gDrawMarker**(nsym) integer, intent(in) :: nsym

Arguments

nsym
Symbol number

= GDOT,	Dot
= GUP,	Upwards triangle
= GDOWN,	Downwards triangle
= GPLUS,	Plus sign
= GCROSS,	Cross
= GBOX,	Small square box
= GDIAMOND,	Diamond
= GCIRCLE,	Small circle
= GSTAR,	Star or asterisk
9 - 23,	Optional hardware symbol
> 23,	Character from font table

Description

The routine gDrawMarker() draws either a dot, one of the eight standard symbols, a hardware symbol or any one of the characters from the GINO font tables.

Symbols are drawn independent of the current GINO font, but symbol numbers can be calculated using the following formulae:

FONT*1000 + <ACSII CODE>

The symbol number is shown in the bottom left corner of the character box in Appendix C.

All symbols are drawn centred at the current drawing position with the current position remaining at the same position after the symbol is drawn.

If untransformed characters are currently set then the appearance of symbols is controlled by the current character size only. If software transformed characters are set (using gSetCharTransformMode()) then the symbol is controlled by all the character attributes (size, angle and italic angle).

The whole of the symbol must fall within the window limits for it to be output.

See Also

Page 161
gSetCharTransformMode
Appendix C - Character Font Tables

gDrawPixel

Syntax

C/C++:	**void gDrawPixel**(int ix, int iy, int pix); **void gDrawPixelArea**(int ix, int iy, int npixx, int npixy, int isx, int isy, int idx, int idy, int pixbuf[]);

F90:	**subroutine gDrawPixel**(ix, iy, pix) **subroutine gDrawPixelArea**(ix, iy, npixx, npixy, isx, isy, idx, idy, pixbuf) integer, intent(in) :: ix,iy,pix integer, intent(in) :: npixx,npixy,isx,isy,idx,idy,pixbuf(*)

Arguments

ix,iy
Pixel (anchor) position (relative to top left corner)

pix
Pixel colour information for single pixel

npixx,npixy
Dimension of pixel data array

isx,isy
The start X position and Y position of a sub-array

idx,idy
The X and Y dimensions of a sub-array

pixbuf
The array containing pixel colour information

Description

The routines gDrawPixel() and gDrawPixelArea() draws a single pixel or a rectangular pixel area using the colour information passed by the user. The pixel (area) is displayed with reference to the anchor position specified by the position (**ix,iy**), noting that the pixel coordinate system has its origin as the top left corner of the device with the first pixel position being referenced as (0,0).

In both routines, the colour information may consist of colour indices or 24bit true colour values depending on the colour mode of the currently nominated device and as set by the routine gSetColourInfo().

In the case of gDrawPixelArea(), the pixel information is passed through a pointer to an integer array **pixbuf** dimensioned (**npixx**,**npixy**). Where the whole of this array is to be displayed, the user should set **isx** and **isy** to 1 and **idx** and **idy** to be the same as **npixx** and **npixy**. Alternatively a portion of the array can be displayed (still at the anchor position **ix**,**iy**) by setting the values of **isx** and **isy** to the offsets from the start of the **pixbuf** array and **idx**,**idy** to the dimensions of the sub-array.

The routine gDrawPixelArea() will extract the pixel data from the **pixbuf** array according to the specification set by gDefinePixelPacking(). The default is for one pixel value to be extracted from one word of **pixbuf**. The pixel rectangle will be clipped to the device limits if these limits are exceeded. The pixel array will also be subject to the current pixel transformation as set by gSetPixelTransform().

See Also

Page 191
gDefinePixelPacking
gSetColourInfo
gSetPixelTransform

gDrawPolygonBound

Syntax

C/C++:	**void gDrawPolygonBound**(int line);

F90:	**subroutine gDrawPolygonBound**(line) integer, intent(in) :: line

Arguments

line

Line style index

= GCURRENT,	Current line style
= 1 - 256,	Line style index
> 256,	Current line style

Description

The routine gDrawPolygonBound() draws the boundaries of all the polygons currently defined.

All polygon edges are drawn. This includes the closing edge and any edges that were defined with invisible moves. **line** specifies the line style for drawing the boundaries. The current line style remains unaffected and the current position is restored to that prior to calling gDrawPolygonBound().

See Also

Page 251
gSelectPolygons

gDrawPolyline

Syntax

C/C++:
void gDrawPolylineBy2D(int npts, GPOINT *points2);
void gDrawPolylineBy3D(int npts, GPOINT3 *points3);
void gDrawPolylineTo2D(int npts, GPOINT *points2);
void gDrawPolylineTo3D(int npts, GPOINT3 *points3);

F90:
subroutine gDrawPolylineBy2D(npts, points2)
subroutine gDrawPolylineBy3D(npts, points3)
subroutine gDrawPolylineTo2D(npts, points2)
subroutine gDrawPolylineTo3D(npts, points3)

integer, intent(in) :: npts
type (GPOINT), intent(in) :: points2(*)
type (GPOINT3), intent(in) :: points3(*)

Arguments

npts
Number of lines to be drawn

points2,points3
Array holding relative or absolute coordinates specifying the end points of lines to be drawn with respect to the current axes and in the current units

Description

The gDrawPolyline set of routines draws a polyline in 2D or 3D using either relative or absolute coordinates. In the case of gDrawPolylineBy2D/3D(), the polyline starts at the current drawing position and is drawn through **npts** series of vector increments. In the case of gDrawPolylineTo2D/3D() the polyline starts at the current drawing position and draws straight lines to each of the **npts** points in **points2** or **points3**.

The current drawing position is updated to be the end of the last vector drawn by these routines.

See Also

Page 82
Page 280

gDrawPolylineSet

Syntax

C/C++:
void gDrawPolylineSet2D(int npol, GPOLYGON *polylines2);
void gDrawPolylinetSet3D(int npol, GPOLYGON3 *polylines3);

F90:
subroutine gDrawPolylineSet2D(npol, polylines2)
subroutine gDrawPolylineSet3D(npol, polylines3)

integer, intent(in) :: npol
type (GPOLYGON),intent(in) :: polylines2(*)
type (GPOLYGON3),intent(in) :: polylines3(*)

Arguments

npol
Number of polylines in polyline set

polylines2,polylines3
Array of 2D or 3D polyline structures to be drawn

Description

The routines gDrawPolylineSet2D() and gDrawPolylineSet3D() draw a set of polygons. Each polygon structure consists of a number of vertices and a pointer to an array of 2D or 3D points. Each polygon is complete within itself and will automatically be closed if not defined as such. Coordinates are absolute and have no relation to the current drawing position.

These routines can handle up to 2048 points and if the total number of points in the polyline set exceeds this, an error message is generated and no output is done.

The current position is restored to that prior to calling either routine.

See Also

Page 85, 283
gFillPolygonSet2D
gFillPolygonSet3D

gDrawPolymarker

Syntax

C/C++:

void gDrawPolymarkerBy2D(int npts, GPOINT *points2, int nsym);
void gDrawPolymarkerBy3D(int npts, GPOINT3 *points3, int nsym);
void gDrawPolymarkerTo2D(int npts, GPOINT *points2, int nsym);
void gDrawPolymarkerTo3D(int npts, GPOINT3 *points3, int nsym);

F90:

subroutine gDrawPolymarkerBy2D(npts, points2, nsym)
subroutine gDrawPolymarkerBy3D(npts, points3, nsym)
subroutine gDrawPolymarkerTo2D(npts, points2, nsym)
subroutine gDrawPolymarkerTo3D(npts, points3, nsym)

```
integer, intent(in) :: npts,nsym
type (GPOINT), intent(in) :: points2(*)
type (GPOINT3), intent(in) :: points3(*)
```

Arguments

npts
Number of symbols to be drawn

points2,points3
Array holding relative or absolute coordinates specifying the points at which symbols are to be drawn (with respect to the current axes and in the current units)

nsym
Symbol number (see gDrawMarker())

Description

The gDrawPolymarker set of routines are used to draw **npts** symbols at points specified by a series of relative or absolute coordinate points held in the 2D array **points2** or the 3D array **points3**. Coordinate points specified by gDrawPolyMarkerBy2D() and gDrawPolyMarkerBy3D() are relative to the current drawing position.

The current drawing position is updated to be the centre of the last symbol drawn by these routines.

See Also Page 97, 163, 287
gDrawMarker

gDrawRect3D

Syntax

C/C++:	**void gDrawRect3D**(float xmin, float xmax, float ymin, float ymax, float zmin, float zmax);

F90:	**subroutine gDrawRect3D**(xmin, xmax, ymin, ymax, zmin, zmax) real, intent(in) :: xmin, xmax, ymin, ymax, zmin, zmax

Arguments

xmin, xmax, ymin, ymax, zmin, zmax
Object limits

Description

The routine gDrawRect3D() draws a ‘solid’ rectangular parallelepiped with its edges parallel to and aligned along the current axes, according to the specified limits.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also Page 307
gDrawFacet

gDrawRuledBezierSurface

Syntax

C/C++:	**void gDrawRuledBezierSurface**(int np1, GPOINT3 *points1, int np2, GPOINT3 *points2, ...);

F90:	**subroutine gDrawRuledBezierSurface**(np1, points1, np2, points2, gUComp, gVComp) integer, intent(in) :: np1,np2 type (GPOINT3), intent(in) :: points1(*),points2(*) integer, optional, intent(in) :: gUComp, gVComp

Arguments

np1
Number of control points

points1
Array of 3D control points

np2
Number of control points

points2
Array of 3D control points

Optional Args ***gUComp ,gVComp***
Optional object complexity (default 10 x 10)

Description The routine gDrawRuledBezierSurface() draws a surface between a curve based on an interpolation of the first set of control points in **points1** and curve based on an interpolation of the second set of control points in **points2**. Where one curve has a different number of control points, the one with the smaller number is elevated to the number in the larger.

The arguments **gUComp** and **gVComp** define the object's complexity in its U (curve1/curve2) and V axes respectively. These values determine the number of divisions (facets) in either direction and therefore the object's smoothness.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also Page 322
gDrawFacet

gDrawSeg

Syntax

C/C++:	**void gDrawSeg**(int nseg);

F90:	**subroutine gDrawSeg**(nseg) integer, intent(in) :: nseg

Arguments ***nseg***
Picture segment or segment group number

> 0,	Draw segment(s) specified by segment **nseg**
= GALL,	Draw all segments
< -1,	Draw all segments except those specified by **nseg**

Description The routine gDrawSeg() is used to redraw an existing segment from the segment display file. gDrawSeg() can be used to redisplay a segment or group which has been 'damaged' by other segment operations using background-erase.

gDrawSeg() does not redraw invisible segments.

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(0), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also Page 433
gSetSegMode

gDrawShadedPolylineTo3D

Syntax

C/C++: **void gDrawShadedPolylineTo3D**(int npts, GPOINT3 *points, GPOINT3 *normals, ...);

F90: **subroutine gDrawShadedPolylineTo3D**(npts, points, normals, gTextCoords,)

```
integer, intent(in) :: npts
type (GPOINT3), intent(in) :: points(*), normals(*)

type (GPOINT3), optional, intent(in) :: gTextCoords(*)
```

Arguments

npts
Number of points in the supplied arrays

points
Array holding absolute coordinates specifying the vertices of a polyline to be drawn with respect to the current axes and in the current units

normals
Array specifying normal vectors at each vertex

Optional Args

gTextCoords
Optional array specifying texture coordinate values at each vertex

Description

The gDrawShadedPolylineTo3D routine draws a polyline in 3D using absolute coordinates subject to lighting and shading conditions. The polyline starts at the first point in the supplied array and and draws straight lines to each of the remaining **npts-1** points in **points**.

The polyline is drawn using the current material attributes and shaded according to the current lighting conditions using the normlas supplied in the array **normals**. The current texture map may also be applied to the polyline, if texture mapping is switched on, using the texture coordinates supplied in the optional array **gTextCoords**, which if not passed are set to zero for each vertex.

The current drawing position is not updated to be the end of the last vector drawn by this routine.

See Also

Page 307
gSetMaterialIndex
gSetTextureMappingMode

gDrawSphere

Syntax

C/C++:	**void gDrawSphere**(float xp, float yp, float zp, float radius, ...);

F90:	**subroutine gDrawSphere**(xp, yp, zp, radius, gURot, gVRot, gWRot, gUComp, gVComp) real, intent(in) :: xp,yp,zp,radius real, optional, intent(in) :: gURot, gVRot, gWRot integer, optional, intent(in) :: gUComp, gVComp

Arguments

xp, yp, zp
Origin (centre) of sphere

radius
Radius of sphere

Optional Args

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gUComp ,gVComp
Optional object complexity (default 10 x 10)

Description

The routine gDrawSphere() draws a 'solid' sphere primitive constructed of outward facing facets according to the specified complexity. The sphere is centred at the origin (**xp**, **yp**, **zp**), with the defined **radius**.

The arguments **gURot**, **gVRot**, **gWRot** specify optional rotations about the object's local axes system. Whilst these will not alter the visual appearance of a coloured sphere, it will affect the appearance of a textured sphere as the texture origin will lie in a different position with relation to the global axes.

The arguments **gUComp** and **gVComp** define the object's complexity in its U (circumference) and V (height) axes respectively. These values determine the number of divisions (facets) in either direction and therefore, the objects smoothness.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 310
gDrawFacet

gDrawSpline

Syntax

C/C++:

void gDrawSplineBy2D(int npts, GPOINT *points2, int beg, int fin);
void gDrawSplineBy3D(int npts, GPOINT3 *points3, int beg, int fin);
void gDrawSplineTo2D(int npts, GPOINT *points2, int beg, int fin);
void gDrawSplineTo3D(int npts, GPOINT3 *points3, int beg, int fin);

F90:

subroutine gDrawSplineBy2D(npts, points2, beg, fin)
subroutine gDrawSplineBy3D(npts, points3, beg, fin)
subroutine gDrawSplineTo2D(npts, points2, beg, fin)
subroutine gDrawSplineTo3D(npts, points3, beg, fin)

integer, intent(in) :: npts,beg,fin
type (GPOINT), intent(in) :: points(*)
type (GPOINT3), intent(in) :: points3(*)

Arguments

npts
Number of coordinate points specified

points2, points3
Array specifying series of relative or absolute points in space coordinates through which the curve is drawn

beg
End conditions for start of a spline curve

= GXPOINT,	Direction of curve calculated using extra point
= GNONE,	No end conditions
= GANGLE,	Direction of curve defined by slope derivatives

fin
End conditions for end of a spline curve (as for **beg**)

Description

The routines gDrawSplineBy2D(), gDrawSplineBy3D(), gDrawSplineTo2D()and gDrawSplineTo3D() draw a smooth spline curve through the specified number of relative or absolute coordinate points. In the case of gDrawSplineBy2D() and gDrawSplineBy3D(), the curve starts at the current drawing position and is drawn through **npts** series of vector increments. In the case of gDrawSplineTo2D() and gDrawSplineTo3D() the curve starts at the first element in **points** and is drawn through the intervening **npts** points.

The number of sub-increments is controlled by the arc routine gSetArcIncrement() and the spline tension is controlled by the routine gSetSplineTension(). At least 3 points are required to define a spline curve.

The start and end directions of the spline curve can be separately controlled by setting **beg** and **fin** to non-zero values. If **beg** or **fin** is set to GANGLE, the direction of the curve is specified directly in terms of slope derivatives. If **beg** or **fin** is set to GXPOINT, the direction of the curve is calculated so that the curve would pass through an extra specified point as if it was extended to include that point. These slopes and/or extra points defining the curve end conditions are set up using gSetCurveAttribs2D() or gSetCurveAttribs3D(). Note that, the spline curve drawing routines update the current curve end conditions so that the start and end slopes match those of the curve that has been drawn. The current curve end conditions can be enquired by calling gEnqCurveAttribs2D() or gEnqCurveAttribs3D().

See Also

Page 98
Page 288
gEnqCurveAttribs2D
gEnqCurveAttribs3D
gSetArcIncrement
gSetCurveAttribs2D
gSetCurveAttribs3D
gSetSplineTension

gDrawSplineSurface

Syntax

C/C++: **void gDrawSplineSurface**(int nx, int ny, GPOINT3 *mesh, ...);

F90: **subroutine gDrawSplineSurface**(nx, ny, mesh, gUComp, gVComp)

```
integer, intent(in) :: nx,ny
type (GPOINT3), intent(in) :: mesh(nx,ny)

integer, optional, intent(in) :: gUComp, gVComp
```

Arguments

nx, ny
Number of points in mesh

mesh
Two dimensions array of 3D mesh points

Optional Args

gUComp ,gVComp
Optional object complexity (default 4*nx and 4*ny)

Description

The routine gDrawSplineSurface() draws a surface based on an interpolation of the supplied mesh such that the surface passes through all the data points supplied.

The arguments **gUComp** and **gVComp** define the object's complexity in its U and V axes respectively. These values determine the number of divisions (facets) in either direction and therefore the object's smoothness. The actual complexity is, however, always rounded down to a complete multiple of the number of data points on the mesh in either direction so that the surface passes through all the data points.

The tension of the spline surface is controlled by the tension routine gSetSplineTension() and the routine takes special account of data that is closed in the U direction, by ensuring a smooth join between the start and end of the surface.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also Page 313
gDrawFacet
gSetSplineTension

gDrawSweptBezierSurface

Syntax

C/C++:	**void gDrawSweptBezierSurface**(int np1, GPOINT3 *points1, int np2, GPOINT3 *points2, ...);

F90:	**subroutine gDrawSweptBezierSurface**(np1, points1, np2, points2, gUComp, gVComp) integer, intent(in) :: np1,np2 type (GPOINT3), intent(in) :: points1(*),points2(*) integer, optional, intent(in) :: gUComp, gVComp

Arguments

np1
Number of control points

points1
Array of 3D control points

np2
Number of control points

points2
Array of 3D control points

Optional Args

gUComp ,gVComp
Optional object complexity (default 10 x 10)

Description

The routine gDrawSweptBezierSurface() draws a surface based on an interpolation of the first set of control points in **points1** extruded along a curve based on an interpolation of the second set of control points in **points2**.

The arguments **gUComp** and **gVComp** define the object's complexity in its U (curve1) and V (curve2) axes respectively. These values determine the number of divisions (facets) in either direction and therefore the object's smoothness.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also Page 321
gDrawFacet

gDrawTabulatedBezierSurface

Syntax

C/C++: **void gDrawTabulatedBezierSurface**(int np, GPOINT3 *points, GPOINT3 vector, ...);

F90: **subroutine gDrawTabulatedBezierSurface**(np, points, vector, gUComp, gVComp)

```
integer, intent(in) :: np
type (GPOINT3), intent(in) :: points(*),vector

integer, optional, intent(in) :: gUComp, gVComp
```

Arguments

np
Number of control points

points
Array of 3D control points

vector
Extrusion vector

Optional Args

gUComp ,gVComp
Optional object complexity (default 10 x 10)

Description

The routine gDrawTabulatedBezierSurface() draws a surface based on an interpolation of the control **points** extruded along the supplied **vector**.

The arguments **gUComp** and **gVComp** define the object's complexity in its U (control curve) and V (extrusion vector) axes respectively. These values determine the number of divisions (facets) in either direction and therefore the object's smoothness.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 320
gDrawFacet

gDrawVolume

Syntax

C/C++: **void gDrawVolume**(float xp, float yp, float zp, int npts, GPOINT *points, ...);

F90: **subroutine gDrawVolume**(xp, yp, zp, npts, points, gVVec, gAbs, gURot, gVRot, gWRot, gUComp)

```
real, intent(in) :: xp,yp,zp
integer, intent(in) :: npts
type (GPOINT), intent(in) :: points(*)

type (GPOINT3), optional, intent(in) :: gVVec
integer, optional, intent(in) :: gAbs
real, optional, intent(in) :: gURot, gVRot, gWRot
integer, optional, intent(in) :: gUComp
```

Arguments

xp, yp, zp
Origin (bottom, centre) of volume

npts
Number of points in outline

points
Array containing outline vertices

Optional Args

gVVec
Optional orientation vector

gAbs
Optional orientation vector direction flag

= GABSOLUTE,	Absolute vector
= GRELATIVE,	Relative vector

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gUComp
Optional object complexity (default 10)

Description

The routine gDrawVolume() draws a ‘solid’ volume of rotation constructed of outward facing facets according to the specified complexity. The volume is generated by rotating the set of outline vertices contained in the **points** array, about a vertical axis which passes through the origin (**xp**, **yp**, **zp**).

An alternative orientation can be specified using either up to three local axes rotations (**gURot**, **gVRot**, **gWRot**) or by using an absolute/relative vector, **gVVec** from the object’s origin. In the latter case a local rotation about the object’s vertical axis can also be added in **gVRot**.

The argument **gUComp** defines the object's complexity about its U axis (circumference). This value determines the number of divisions (facets) in this direction and therefore, the objects smoothness. The number of facets in the vertical (V axis) direction is determined by **npts**.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also

Page 311
gDrawFacet

gDrawWedge

Syntax

C/C++: **void gDrawWedge**(float xp, float yp, float zp, ...);

F90: **subroutine gDrawWedge**(xp, yp, zp, gUDim, gVDim, gWDim, gURot, gVRot, gWRot, gUVec, gVVec, gWVec, gAbs)

```
real, intent(in) :: xp,yp,zp

real, optional, intent(in) :: gUDim, gVDim, gWDim
real, optional, intent(in) :: gURot, gVRot, gWRot
type (GPOINT3), optional, intent(in) :: gUVec, gVVec, gWVec
integer, optional, intent(in) :: gAbs
```

Arguments

xp, yp, zp
Origin (bottom, left, back corner) of wedge

Optional Args

gUDim, gVDim, gWDim
Optional wedge dimensions (default 1.0,1.0,1.0)

gURot ,gVRot, gWRot
Optional rotations about local axes (default 0.0,0.0,0.0)

gUVec, gVVec, gWVec
Optional edge vectors

gAbs
Optional edge vector direction flag

= GABSOLUTE,	Absolute edge vectors
= GRELATIVE,	Relative edge vectors

Description

The routine gDrawWedge() draws a 'solid' wedge primitive of the specified dimensions by generating outward facing facets for each face. The wedge is positioned at the specified origin (**xp**, **yp**, **zp**), with a default unit dimension extending in a positive direction along each of its three local axes. The wedge is cut from top left to bottom right.

An alternative dimension/orientation can be specified using either a) up to three dimension values (**gUDim**, **gVDim**, **gWDim**) and up to three local axes rotations (**gURot**, **gVRot**, **gWRot**) or b) as three mutually perpendicular edge vectors (**gUVec**, **gVVec**, **gWVec**) which may be absolute or relative to the object's origin.

The facets are drawn in the current facet colour/material according to the current lighting/shading environment and subject to the current modelling/viewing transformations.

See Also Page 309
gDrawFacet

gDummy

Syntax

C/C++:	**void gDummy**(void);

F90:	**subroutine gDummy**

Arguments None

Description A call to gDummy() nominates a notional device which does not produce any graphics output.

Used in conjunction with gDebug() or gSetTracerMode() it is a useful aid to program development.

It is called by GINO if the user fails to nominate a device. This is to allow proper initialization of GINO.

See Also Page 42
gDebug
gSetTracerMode

gElevateBezier

Syntax

C/C++:	**void gElevateBezier2D**(int *npts, GPOINT *points2); **void gElevateBezier3D**(int *npts, GPOINT3 *points3);

F90:	**subroutine gElevateBezier2D**(npts, points2) **subroutine gElevateBezier3D**(npts, points3) **integer, intent(inout) :: npts** **type (GPOINT), intent(inout) :: points2(*)** **type (GPOINT3), intent(inout) :: points3(*)**

Arguments ***npts***
Number of points in Bezier curve, incrented by one on return.

points2,points3
Array holding absolute coordinates of Bezier curve control points.

Description The gElevateBezier set of routines takes a set of Bezier curve control points in 2D or 3D and generates a new set, with an additional control point, that represents the same curve.

Note that the value passed in **npts** will be incremented by one on return and the arrays **points2** or **points3** must be large enough to contain the extra control point that is returned in these arrays.

See Also

Page 103
Page 290
gReduceBezier2D
gReduceBezier3D

gEditSeg

Syntax

C/C++:	**void gEditSeg2D**(int nseg, int tag, GMAT2D t2, int sw); **void gEditSeg3D**(int nseg, int tag, GMAT3D t3, int sw);

F90:	**subroutine gEditSeg2D**(nseg, tag, t2, sw) **subroutine gEditSeg3D**(nseg, tag, t3, sw) integer, intent(in) :: nseg,tag real, intent(in) :: t2(6),t3(16) integer, intent(in) :: sw

Arguments

nseg
Picture segment number

tag
Segment tag identifier

t2,t3
2D or 3D transformation matrix

sw
Transformation mode switch

= GAPPLY,	Apply matrix according to current transformation mode
= GREPLACE,	Replace current transformation matrix

Description

The routines gEditSeg2D() and gEditSeg3D() replace the modelling transformation matrix that immediately follows the identifier **tag** in segment **nseg**, with matrix **t2** or **t3**.

The argument **sw** determines the way in which the matrix will affect the modelling transformation when the segment is redrawn. If **sw** = GAPPLY then the new transformation matrix **t2/t3** will be added to the existing modelling transformation matrix according to the current transformation mode (see gSetTransformMode()). If **sw** = GREPLACE then the matrix **t2/t3** will replace the existing modelling transformation with the new matrix.

If the segment **tag** occurs more than once in the segment, all matrices following the tag will be edited.

If the identifier **tag** does not exist within the segment a warning message is output but if the tag is not followed by a transformation matrix, no action is taken.

See Also Page 436
gInsertSegTag
gSetTransformMode

gEndBatchUpdate

Syntax

C/C++:	**void gEndBatchUpdate**(void);

F90:	**subroutine gEndBatchUpdate**

Arguments None

Description This routine is used to end a batch of modifications started by gStartBatchUpdate(). The modifications are output to the display to reflect the changes.

See Also Page 50
gStartBatchUpdate

gEndPolygon

Syntax

C/C++:	**void gEndPolygon**(void);

F90:	**subroutine gEndPolygon**

Arguments None

Description A polygon is closed by a call to gEndPolygon(). At this point the current polygon identifier is assigned to the polygon. No more vertices can be added to the polygon after a call to gEndPolygon().

See Also Page 247

gEnqArcState

Syntax

C/C++:	**void gEnqArcState**(int *sw, int *nincs, float *tol);

F90:	**subroutine gEnqArcState**(sw, nincs, tol) integer, intent(out) :: sw,nincs real, intent(out) :: tol

Arguments ***sw***
State of software arc switch

= GHARD,	Hardware arcs
= GSOFT,	Software arcs

nincs
Number of increments per full circle

tol
Arc tolerance

Description

The routine gEnqArcState() returns values in **sw**,**nincs** and **tol** as set up by the most recent calls to gSetArcMode(), gSetArcIncrement() and gSetArcTolerance() respectively. If gSetArcMode() or gSetArcTolerance() have not been called, the appropriate arguments are set to the defaults, which depend on the resolution of the output device. If gSetArcIncrement() has not been called, **nincs** is returned as 0.

See Also

Page 91
gSetArcIncrement
gSetArcMode
gSetArcTolerance

gEnqBrokenLine

Syntax

C/C++:	**void gEnqBrokenLine**(int brk);

F90:	**subroutine gEnqBrokenLine**(brk) integer, intent(out) :: brk

Arguments

brk
Current broken line type

= GSOLID,	Solid
= 1 - 256,	User defined or device dependent
> 256,	Solid or device dependent

Description

The routine gEnqBrokenLine() returns the currently requested broken line type. The current line type may be set by calling gSetBrokenLine(). The default for the broken line type is solid, which corresponds to a call to gSetBrokenLine(0).

If **brk** is returned set to zero, the current line type is solid. If **brk** is in the range 1 to 256, the parameters that define the current line type may be enquired by calling gEnqBrokenLineStyle().

See Also

Page 116
gSetBrokenLine
gEnqBrokenLineStyle

gEnqBrokenLineStyle

Syntax

C/C++:	**void gEnqBrokenLineStyle**(int brk, GBRKSTY *rep);

F90:	**subroutine gEnqBrokenLineStyle**(brk, rep) integer, intent(in) :: brk type (GBRKSTY), intent(out) :: rep

Arguments

brk
Broken line type

= 1 to 256 only

rep.mode
Line mode

= GSOLID
= GCONTDASH
= GCONTCHAIN
= GDISCONTDASH
= GDISCONTCHAIN

rep.repeat
Repeat length in current units

rep.dash
Dash length in current units

rep.dot
Dot length in current units (for chained lines)

Description

The routine gEnqBrokenLineStyle() returns the line style parameters **rep.mode**, **rep.repeat**, **rep.dash** and **rep.dot** associated with one of the broken line types 1 to 256. If **brk** is outside this range, an error message is output and all the values are returned set to zero. In the absence of any calls to gDefineBrokenLineStyle(), gEnqBrokenLineStyle() returns the default values for the line type parameters.

See Also

Page 124
gDefineBrokenLineStyle

gEnqCharAttribs

Syntax

C/C++:	**void gEnqCharAttribs**(GCHASTY *rep);

F90:	**subroutine gEnqCharAttribs**(rep) type (GCHASTY), intent(out) :: rep

Arguments

rep.type
Character output type

= 1,	Hardware (gSetHardChars(),gSetMixedChars() or gSetHardCharSize())
= 2,	Pseudo hardware (gSetHardChars(),gSetMixedChars() or gSetHardCharSize())
= 3,	Software untransformable (gSetSoftChars())
= 4,	Software transformable (gSetCharTransformMode(GON))

rep.width
Character width in current units

rep.height
Character height in current units

rep.size
Hardware character size

rep.slant
Italic angle in degrees

rep.angle
String angle in degrees

Description

The routine gEnqCharAttribs() returns information about the current character settings in the structure **rep**. The structure, which is of type GCHASTY contains the following elements:

If **rep.size** is returned set to -1, a call to gSetHardCharSize() is not current and the working character specification is returned in **rep.width**, **rep.height**, **rep.slant** and **rep.angle**. This may differ slightly from the requested specification (see gSetMixedChars() and gSetHardChars()).

If **rep.size** is greater than or equal to zero, gSetHardCharSize() has set the current character size and orientation. The size is returned in **rep.width** and **rep.height**, **rep.angle** will be set to 0.0 or 90.0 and **rep.slant** will be set to zero. If **rep.size** is zero, **rep.width** and **rep.height** return the default character size.

See Also

Page 151
gSetHardCharSize
gSetCharTransformMode
gSetHardChars
gSetMixedChars
gSetSoftChars

gEnqCharTransform

Syntax

C/C++:
```
void gEnqCharTransform(float dx, float dy, GPOINT *point);
```

F90:
```
subroutine gEnqCharTransform(dx, dy, point)

real, intent(in) :: dx,dy
type (GPOINT), intent(out) :: point
```

Arguments ***dx,dy***
Width and height of character string

point
Return the relative end position subject to character angle and italic

Description The routine gEnqCharTransform() transforms a coordinate position according to the current character angle and italic settings.

See Also Page 152
gSetStrAngle
gSetItalicAngle

gEnqClippingMode

Syntax

C/C++:	**void gEnqClippingMode**(int *sw);

F90:	**subroutine gEnqClippingMode**(sw) integer, intent(out) :: sw

Arguments ***sw***
Clipping mode:

= GNOCLIP,	No hardware or software clipping
= GHARD,	Hardware clipping (default)
= GSOFT,	Software clipping

Description The routine gEnqClippingMode() returns the current clipping mode for GINO output as set by the routine gSetClippingMode().

The routine gSetClippingMode() may be used to switch to software clipping or switch all clipping off.

See Also Page 221, 222
gSetClippingMode
gSetViewportClipSwitch
gSetWindow2D
gSetWindow3D

gEnqColourInfo

Syntax

C/C++:	**void gEnqColourInfo**(int *ndc, int *ndt);

F90:	**subroutine gEnqColourInfo**(ndc, ndt) integer, intent(out) :: ndc,ndt

Arguments ***ndc***
Number of colours that can be separately defined or selected

= 0,	Monochrome device
= 1,	Monochrome device with background erase
> 1,	Colour/greyscale device

ndt
Display type, identifying the colour/greyscale capabilities of the device:

= 0,	No colour/greyscale variation (i.e. Monochrome)
= ±1,	Fixed colour/greyscale (i.e. no gDefineRGB() facilities)
= ±2,	Static colour/greyscale (i.e. selection from fixed palette)
= ±3,	Dynamic colour/greyscale
= ±4,	Direct colour/greyscale

N.B. Positive values indicate colour display, negative values indicate greyscale

Description

The routine gEnqColourInfo() returns in **ndc** and **ndt** information about the device's current colour and greyscale capabilities. In general, a device can only display a finite number of colours/greyscales. The number of colours/greyscales that can be defined (see gDefineRGB()) and selected (see gSetLineColour()) is returned in **ndc**. If **ndc** is set to zero, the device cannot display any colours.

ndt returns information about the colour/greyscale capabilities of the device: **ndt**= 0 for monochrome displays; = ± 1 for displays with fixed colours/greyscales which cannot be changed with gDefineRGB(); = ± 2 for displays with a palette but may not provide exactly the colour requested with gDefineRGB(); = ± 3 for dynamic colour/greyscale devices and = ±4 for direct colour/greyscale devices. Negative values indicate greyscale rather than colour display.

See Also

Page 46, 206
gSetLineColour
gDefineRGB

gEnqConfigStatus

Syntax

C/C++: **int gEnqConfigStatus**(...);

F90: **integer function gEnqConfigStatus**(cfgdir)

character*(*), optional, intent(in) :: cfgdir

Arguments

None

Optional Args

cfgdir
Optional location of GINO configuration file

Description

The function gEnqConfigStatus() enables a GINO application to check the status of the GINO Configuration File and take appropriate action if the file is not found or illegal. The routine returns a zero value if the GINO configuration file is found to exist and contains the correct licencing information.

If the routine gEnqConfigStatus() is not called, the configuration file is checked by the first GINO routine used, and if not satisfactory the application will STOP.

Where an application requires a check on the configuration file, gEnqConfigStatus() should therefore be used in place of the call to gOpenGino(), ie. the very first GINO routine called in an application. If gEnqConfigStatus() is called at any other time GINO is re-initialized.

See Also

Page 26, 40
gOpenGino

gEnqCursorAction

Syntax

C/C++: **void gEnqCursorAction**(int *action, int *lverts, GPOINT *points);

F90: **subroutine gEnqCursorAction**(action, lverts, points)

integer, intent(out) :: action,lverts
type (GPOINT), intent(out) :: points(*)

Arguments

action
Cursor action type

= GPOLYLINE,
= GDEFAULT,
= GRUBBERBAND,
= GRUBBERBOX,
= GRUBBERSQUARE,
= GRUBBERELLIPSE,
= GRUBBERCIRCLE,

lverts
Number of vertices if polyline cursor

points
Coordinates of polyline cursor

Description

The routine gEnqCursorAction() returns current setting for the cursor action type as set by gSetCursorAction(). The default cursor action type is for the currently defined cursor shape to follow the cursor or pointer position with no additional action.

If the current type is a user defined polyline cursor (**action** = GPOLYLINE) the coordinates are returned in **points** with the number of vertices returned in **lverts**. **points** should be declared as length 200 to contain the maximum possible number of points that could be returned by this routine. All other cursor action types return **lverts** as zero with no information in **points**.

The availability of cursor action types is hardware dependent and users should refer to the relevant Appendix B document for the current device being used.

See Also

Page 243
gSetCursorAction
Appendix B

gEnqCursorType

Syntax

C/C++:	**void gEnqCursorType**(int *type, int *forcol, int *bakcol);

F90:	**subroutine gEnqCursorType**(type, forcol, bakcol) integer, intent(out) :: type,forcol,bakcol

Arguments

type
Cursor Type

= GDEFAULT,	Default
= GSMALLCROSS,	Small cross
= GLARGECROSS,	Large cross(full screen/window if available)
= GX,	X
= GPOINTER,	Pointer
> 4,	Hardware dependent cursor types

forcol
Foreground colour

bakcol
Background colour

Description

The routine gEnqCursorType() returns the current setting for the cursor type as set by gSetCursorType().

The default cursor type is hardware dependent. When **type** > 4 refer to the appropriate Appendix B document for a list of available cursor types.

See Also

Page 242
gSetCursorType
Appendix B

gEnqCurveAttribs

Syntax

C/C++:	**void gEnqCurveAttribs2D**(float dxbeg, float dybeg, float dxfin, float dyfin, GPOINT *begp2, GPOINT *finp2); **void gEnqCurveAttribs3D**(float dxbeg, float dybeg, float dzbeg, float dxfin, float dyfin, float dzfin, GPOINT3 *begp3, GPOINT3 *finp3);

F90:	**subroutine gEnqCurveAttribs2D**(dxbeg, dybeg, dxfin, dyfin, begp2, finp2) **subroutine gEnqCurveAttribs3D**(dxbeg, dybeg, dzbeg, dxfin, dyfin, dzfin, begp3, finp3) real, intent(out) :: dxbeg,dybeg,dzbeg,dxfin,dyfin,dzfin type (GPOINT), intent(out) :: begp2,finp2 type (GPOINT3), intent(out) :: begp3,finp3

Arguments

dxbeg,dybeg,dzbeg
Slope angle/derivative for start of the curve

dxfin,dyfin,dzfin
Slope angle/derivative for end of the curve

begp2,begp3
Extra point defining start angle of curve

finp2,finp3
Extra point defining end angle of curve

Description

The routine gEnqCurveAttribs2D() and gEnqCurveAttribs3D() return the curve end conditions which have either been set through calling gSetCurveAttribs2D() or gSetCurveAttribs3D() or have been set/updated as a result of the last curve drawing routine.

For the piecewise cubic curves, the slopes are measured in terms of the cosine and sine of the angles at each end, whereas for the spline curves, the slope is measured in terms of actual gradient and therefore will be scaled.

The curve end conditions define the direction of a curve at each end. They will be used in the next curve drawing routine if the curve is to be drawn with specified end conditions and then updated so that the start and end slopes match those of the curve that has been drawn. By enquiring and setting the curve end conditions in between curve drawing routines, it is possible to get curves to merge smoothly.

See Also

Page 100, 289
gDrawAkimaBy2D
gDrawAkimaTo2D
gDrawCurveBy2D
gDrawCurveTo2D
gDrawSplineBy2D
gDrawSplineTo2D
gSetCurveAttribs2D
gSetCurveAttribs3D

gEnqDepthMode

Syntax

C/C++: **void gEnqDepthMode**(int *mode, float *dinit);

F90: **subroutine gEnqDepthMode**(mode, dinit)

```
integer, intent(out) :: mode
real, intent(out) :: dinit
```

Arguments

mode
Depth test mode

= GNEVER,	Never display output
= GLESSTHAN,	Display if depth < value in depth buffer
= GLESSTHANOREQUALTO,	Display if depth <= value in depth buffer

= GEQUALTO,	Display if depth = value in depth buffer
= GNOTEQUALTO,	Display if depth <> value in depth buffer
= GGREATERTHANEQUALTO,	Display if depth >= value in depth buffer
= GGREATERTHAN,	Display if depth > value in depth buffer
= GALWAYS,	Always display output

dinit
Initial depth buffer setting (0.0 - 1.0)

Description This routine returns the current or default settings of the depth buffer operation.

See Also Page 329
gSetDepthMode

gEnqDeviceState

Syntax

C/C++:	**void gEnqDeviceState**(GDEVSTATE *devstate);

F90:	**subroutine gEnqDeviceState**(devstate) type (GDEVSTATE), intent(out) :: devstate

Arguments

devstate.name
Device driver nomination routine name

devstate.ddver
Device driver version number

devstate.maxaux
Maximum number of auxiliary drawing areas

devstate.dddim
Coordinate system handled by driver

= 2,	2D driver
= 3,	3D driver

devstate.ndevty
Device type

<0,	Metafile device (JPEG, PNG, BMP etc.)
1 - 99,	Graphics terminal
100 - 199,	Plotter (on-line)
200 - 299,	Plotter (on-line)
300 - 399,	Windowing Device
400 - 499,	GINOMENU GUI device (no GINO events)

devstate.ndev
Device output file pointer/unit

devstate.ntype
Device output type

devstate.xmm
Number of millimetres in 1 current unit

devstate.xdu
Number of device units in 1 current unit

devstate.unitsc
Device unit in millimetres

devstate.atrib
Device attribute flag

devstate.ahard
Arc generation flag

= 0,	Hardware arcs not available
= 1,	Hardware arcs available

devstate.arctol
Arc tolerance

devstate.cwt
Line attributes entries flag

= 0,	Driver used PENSEL entry
= 1,	Driver used LINCOL, LINWID, PENTYP and PENSET

devstate.nbrk
Number of hardware broken line types

devstate.nlend
Number of line end types

devstate.thick
Thick line generation flag

= 0,	Thick lines done in driver
= 1,	Thick lines generated by lines drawn in direction of line
= 2,	Thick lines generated by horizontal/vertical lines
= 3,	Thick lines generated by hardware area fill

devstate.chard
Character generation flag

= 1,	Hardware characters available
= 2,	Hardware characters not available

devstate.nchard
Number of hardware character sizes

devstate.cangm
Hardware character angle multiple

= 0,	Only horizontal
= 1	All angles can be drawn
> 1,	Only angles at this multiple (eg. 90 degrees)

devstate.rectfi
Rectangle filling flag

= 0,	No rectangle filling available
= 1,	Rectangle filling available

devstate.npolmx
Maximum number of polygons that can be filled

= -1,	No limitation
= 0,	No polygon filling
= 1,	Single polygon only
> 1,	Maximum polygon limit

devstate.nvermx
Maximum number of points in filled polygons

devstate.ndcmax
Maximum number of colours/pens available

= 0,	Monochrome device
= 1,	Monochrome device with background erase
> 1,	Colour/greyscale device

devstate.ndtmax
Display colour type

= 0,	No colour/greyscale variation (i.e. Monochrome)
± 1,	Fixed colour/greyscale (i.e. no rgb facilities)
± 2,	Static colour/greyscale (i.e. selection from fixed palette)
± 3,	Pseudo colour/greyscale
± 4,	Direct colour/greyscale

devstate.whard
Hardware windowing flag

= 0,	No hardware windowing
= 1,	Device can do hardware windowing

devstate.mhard
Hardware masking flag

= 0,	No hardware masking
= 1,	Device can do hardware masking

devstate.nsegmx
Maximum permitted segment number

devstate.ncurtp
Number of CURSOR types

devstate.nevetp
Number of EVENT types

devstate.nquemx
Maximum queue length

devstate.dialog
Dialogue area flag

= 0,	No dialogue facilities
= 1,	Device has dialogue facilities

devstate.xpixel
Horiz. pixel size in current units

devstate.ypixel
Vert. pixel size in current units

devstate.npixdp
Maximum pixel depth

devstate.nxpix
Number of pixels in horizontal device limit

devstate.nypix
Number of pixels in vertical device limit

Description

The routine gEnqDeviceState() returns information relating to the currently nominated device in the structure **devstate**.

The information returned by gEnqDeviceState() is set by the device driver and in most cases cannot be changed by a GINO program after device initialization.

See Also

Page 43
gEnqColourInfo
gEnqMaxDrawingLimits

gEnqDrawingLimits

Syntax

C/C++: **void gEnqDrawingLimits**(GDIM *dim, int *type);

F90: **subroutine gEnqDrawingLimits**(dim, type)

type (GDIM), intent(out) :: dim
integer, intent(out) :: type

Arguments

dim
Paper dimensions in current units

type
Paper type

Description

The routine gEnqDrawingLimits() returns **dim.xpap, dim.ypap** of the currently selected paper or workstation window limits and **type** as the current paper type as defined in the most recent call to gSetDrawingLimits(). If gSetDrawingLimits() has not been called, the default values are returned.

The routine gEnqMaxDrawingLimits() may be used to find the maximum available limits.

See Also Page 45, 272
gSetDrawingLimits
gEnqMaxDrawingLimits

gEnqEscapeChar

Syntax

C/C++:	**void gEnqEscapeChar**(char *cha);

F90:	**subroutine gEnqEscapeChar**(cha) character*(*), intent(out) :: cha

Arguments

cha
Escape character

Description

The routine gEnqEscapeChar() returns the current string escape character. The default escape character is * (see gSetEscapeChar()).

See Also

Page 158
gSetEscapeChar

gEnqFacetFillStyle

Syntax

C/C++:	**void gEnqFacetFillStyle**(int *fill);

F90:	**subroutine gEnqFacetFillStyle**(fill) integer, intent(out) :: fill

Arguments

fill
Facet fill style

= GHOLLOW,	Boundary only
= GSOLID,	Solid fill

Description

The routine gEnqFacetFillStyle() returns the current facet filling style.

See Also

Page 302
gSetFacetFillStyle

gEnqFacetMaterialProps

Syntax

C/C++:	**void gEnqFacetMaterialProps**(int face, int *amb, int *diff, int *spec, int *emit, float *shine, float *trans);

F90:	**subroutine gEnqFacetMaterialProps**(face, amb, diff, spec, emit, shine, trans) integer, intent(in) :: face integer, intent(out) :: amb,diff,spec,emit real, intent(out) :: shine,trans

Arguments

face
Facet face

= GFRONT,	Enquire material properties for front face
= GBACK,	Enquire material properties for back face

amb
Ambient reflection colour

diff
Diffuse reflection colour

spec
Specular reflection colour

emit
Emission colour

shine
Specular concentration (shininess) as percentage

trans
Translucence (filtering) (0.0-1.0)

Description

The routine gEnqFacetMaterialProps() returns the current facet material properties as set by the routine gSetFacetMaterialProps().

See Also

Page 342
gSetFacetMaterialProps

gEnqFacetOffsetMode

Syntax

C/C++:	**void gEnqFacetOffsetMode**(int *mode);

F90:	**subroutine gEnqFacetOffsetMode**(mode) integer, intent(out) :: mode

Arguments ***mode***
Facet offset mode

= GOFF,	No facet offset set
= GBOUNDARYAWAY,	Shift boundary away from viewpoint
= GBOUNDARYNEAR,	Shift boundary towards the viewpoint
= GINTERIORAWAY,	Shift interior away from viewpoint
= GINTERIORNEAR,	Shift interior towards the viewpoint

Description The routine gEnqFacetOffsetMode() returns the current facet offset mode as set by gSetFacetOffsetMode().

See Also Page 303
gSetFacetOffsetMode

gEnqFog

Syntax

C/C++: **void gEnqFog**(GFOGATT *attribs);

F90: **subroutine gEnqFog**(attribs)

type (GFOGATT), intent(out) :: attribs

Arguments ***attribs.mode***
Fog mode

= GNONE,	No fog
= GLINEAR,	Linear for depth-cueing
= GEXP1,	Exponential for cloud and heavy fog
= GEXP2,	Exponential for smoke and weather haze

attribs.colour
Fog colour (index or 24bit true colour value)

attribs.density
Fog density for exponential modes (default = 0.0025)

attribs.start
Start depth for linear fog (default = 0.0)

attribs.end
End depth for linear fog (default = 1.0)

Description The routine gEnqFog() returns the current fog settings.

See Also Page 338
gDefineFog

gEnqFontStyle

Syntax

C/C++:	**void gEnqFontStyle**(int *font, GFNTFILSTY *style, int *weight, int *space, int *rep);

F90:	**subroutine gEnqFontStyle**(font, style, weight, space, rep) integer, intent(out) :: font type (GFNTFILSTY), intent(out) :: style integer, intent(out) :: weight,space,rep

Arguments

font
Current font number

style
Font filling style definition

weight
Font weight

< 0,	Font thinning factor
= 0,	Normal font weight
> 0,	Font weighting factor

space
Font spacing

= 0,	Normal font spacing
= 1,	Force equal spacing

rep
Font representation

= 0,	for normal font representation
= 1,	Display as font 0
= 2,	Display as boxes (type 1)
= 3,	Display as font 0 and boxes (type 1)
= 4,	Display as boxes (type 2)
= 5,	Display as font 0 and boxes (type 2)
= 6,	Display as boxes (type 3)
= 7,	Display as font 0 and boxes (type 3)

Description

The routine gEnqFontStyle() returns the current settings of the font attributes. The following list shows each parameter's description, and the routine in which it is set.

Parameter	Description	Set By
font	font number	gSetCharFont
style	Setting for the font fill style	gSetFontFillStyle
weight	Font weight	gSetFontWeight
space	Font spacing	gSetFontSpacing
rep	Software font representation	gSetFontForm

See Also

Page 147
gSetCharFont
gSetFontForm
gSetFontSpacing
gSetFontFillStyle
gSetFontWeight

gEnqGinoState

Syntax

C/C++:	**void gEnqGinoState**(GLIBSTATE *gstate);

F90:	**subroutine gEnqGinoState**(gstate) type (GLIBSTATE), intent(out) :: gstate

Arguments

gstate.gino
GINO State

= 1,	GINO initialized
= 2,	Device nominated
= 3,	Device initialized and drawing units defined
= 4,	Picture started and drawing limits defined
= 5,	Segment open and drawing started

gstate.graf
GINOGRAF state

= 0,	Library not initialized
= 1,	Library initialized

gstate.surf
GINOSURF state

= 0,	Library not initialized
= 1,	Library initialized

gstate.menu
GINOMENU state

= 0,	Library not initialized
= 1,	Library initialized

Description

The routine gEnqGinoState() returns the state of GINO and its associated libraries.

The variable **gstate.gino** gives the state of GINO, in terms of device nomination and whether drawing has started. All GINO routines either require GINO to be in a particular state before operation or set GINO to a particular state after operation. These are classed as follows:

State	Require GINO in State	Move GINO to state
1	Set or enqure GINO Attribute	Close driver
2	Device Qualification or device enquiry	Device nomination
3	Set device attribute table	Clear picture
4	Set Device attribute	Close segment
5	Drawing Routine	Open segment or drawing

Note that all drawing is placed in the dummy segment zero even if segments are not active. If segments are active the currently opened segment can be obtained through the routine gEnqOpenSeg().

The state of GINO's associated libraries can be obtained from the remaining three elements of the GLIBSTATE structure.

See Also Page 26, 50
gEnqOpenSeg

gEnqHardFontList

Syntax

C/C++: **void gEnqHardFontList**(int list[], int n, int *count);

F90: **subroutine gEnqHardFontList**(list, n, count)

integer, intent(in) :: n
integer, intent(out) :: list(*), count

Arguments

list
Array containing font numbers

n
Number of elements in **list**

count
Number of font numbers available

Description

The routine gEnqHardFontList() returns a list of hardware font numbers that the current device has available. The numbers returned pertain to the font numbers specified in the routine gSetCharFont().

The user should set **n** to be the size of the array **list** and no more than **n** font numbers will be returned. The actual number of font numbers available is returned in **count** whether this is larger or smaller than **n**.

See Also Page 147
gSetCharFont

gEnqHatchStyle

Syntax

C/C++: **void gEnqHatchStyle**(int fill, GHATSTY *rep);

F90: **subroutine gEnqHatchStyle**(fill, rep)

integer, intent(in) :: fill
type (GHATSTY), intent(out) :: rep

Arguments

fill
Hatch style index

rep
Hatch style representation

Description

The routine gEnqHatchStyle() returns the values defined for a particular entry in the table of hatch styles in the structure **rep**. The elements are initially defined by GINO but may subsequently have been redefined by calls to gDefineHatchStyle(). The elements of the hatch style structure are discussed under the routine gDefineHatchStyle().

See Also

Page 172, 182
gDefineHatchStyle

gEnqHLS

Syntax

C/C++: **void gEnqHLS**(int col, GHLSSTY *hls);

F90: **subroutine gEnqHLS**(col, hls)

integer, intent(in) :: col
type (GHLSSTY), intent(out) :: hls

Arguments

col
Colour index

< 0,	Dummy definition
= GBACKGROUND,	Background colour
> 0,	Index up to device capability

hls
Hue, lightness and saturation values

Description

The routine gEnqHLS() returns the hue, lightness and saturation values identified by **col**. The colour may have been defined by a routine other than gDefineHLS(), in which case the colour coordinates are converted to HLS. If **col** is less than zero, the colour values returned are those last specified by any of the colour definition routines.

The default values are returned by gEnqHLS() until the colour identified by **col** is redefined by any call to the colour definition routines.

The default values correspond to as many of the standard GINO colours as the device can implement. For plotters and other devices that do not cater for variable colours or if **col** is out of range for the device, gEnqHLS() returns **hls.hue**, **hls.light** and **hls.sat** set to zero.

See Also

Page 216
gDefineHLS
gDefineHSV
gDefineRGB

gEnqHSV

Syntax

C/C++: **void gEnqHSV**(int col, GHSVSTY *hsv);

F90: **subroutine gEnqHSV**(col, hsv)

integer, intent(in) :: col
type (GHSVSTY), intent(out) :: hsv

Arguments

col
Colour index

< 0,	Dummy definition
= GBACKGROUND,	Background colour
> 0,	Index up to device capability

hsv
Hue, saturation and value settings

Description

The routine gEnqHSV() returns the hue, saturation and value settings identified by **col**. The colour may have been defined by a routine other than gDefineHSV(), in which case the colour coordinates are converted to HSV. If **col** is less than zero, the colour values returned are those last specified by any of the colour definition routines.

The default values are returned by gEnqHSV() until the colour identified by **col** is redefined by any call to the colour definition routines.

The default values correspond to as many of the standard GINO colours as the device can implement. For plotters and other devices that do not cater for variable colours or if **col** is out of range for the device, gEnqHSV() returns **hsv.hue**, **hsv.sat** and **hsv.value** set to zero.

See Also

Page 214
gDefineHLS
gDefineHSV
gDefineRGB

gEnqImageFile

Syntax

C/C++:	**void gEnqImageFile**(char file[], int *type, int *ixgrid, int *iygrid, int *nbpp, int *ncols);

F90:	**subroutine gEnqImageFile**(file, type, ixgrid, iygrid, nbpp, ncols) character*(*), intent(in) :: file integer, intent(out) :: type,ixgrid,iygrid,nbpp,ncols

Arguments

file
Image file name

type
Image type

= 0,	Unknown image file type
= GBMPFILE,	Uncompressed Windows Bitmap file (BMP)
= GXWDFILE,	X Windows Dump format (XWD)
= GICOFILE,	Windows Icon file (ICO)
= GJPEGFILE,	JPEG image format (JPG)
= GPNGFILE,	Portable Network Graphics file (PNG)

ixgrid
Image width in pixels

iygrid
Image height in pixels

nbpp
Number of bits per pixel

ncols
Number of colours in image file colour table

Description

The routine gEnqImageFile() can be used to interrogate an image file to determine the image type and a number of its attributes prior to reading in the image file itself.

Five image file types are recognised by this routine (matching those that can be interpreted by the routine gGetImageFile()) and the attributes returned are as described above.

See Also

Page 74
gGetImageFile

gEnqImpAttribs

Syntax

C/C++:	**void gEnqImpAttribs**(GIMPLEMENTATION *impl);

F90:	**subroutine gEnqImpAttribs**(gstate) type (GIMPLEMENTATION), intent(out) :: impl

Arguments

impl.rmin
Minimum real value for machine/implementation

impl.rmax
Maximum real value for machine/implementation

impl.rsmall
Minimum real value > 0.0 for machine/implementation

impl.rsig
Small significant real number for machine/implementation

impl.imin
Minimum integer value for machine/implementation

impl.imax
Maximum integer value for machine/implementation

impl.nipr
Number of integer words per real word

impl.nfmax
Number of bytes per integer word

impl.nbits
Number of bits per machine word

impl.nbmask
Integer word with all bits set to 1

impl.iwtres
Smallest unit of time in milliseconds for generating time delay

impl.nbyter
Number of bytes per unit in unformatted file record length specification

impl.nfumin
Lowest valid Fortran unit number

impl.nfumax
Higest valid Fortran unit number

impl.ndevdf
Default Fortran unit number used for device output to file

impl.nsavdf
Default Fortran unit number used for SAVDRA device output

impl.nfdinp
Fortran unit used for command input

impl.nfdout
Fortran unit used for command output

impl.nfertr
Fortran unit used for error and trace output

impl.nfmess
Fortran unit used to read GINO error message file

impl.nfsdf
Fortran unit used for Software Display File

impl.nffont
Fortran unit used for reading GINO font file

impl.nficon
Fortran unit used for reading GINOMENU icon file

impl.nfstat
Fortran unit used for GINO-F state stack file

impl.nfimpl
Implementation number

impl.nflice
GINO license number

impl.iso
Character set flag (0=ANSI/DOS, 1=ISO)

impl.dsep
Pathname directory separator character ('\' for PC, '/' for Unix, '.' for VMS)

Description The routine gEnqImpAttribs() returns the settings of various implementation dependent variables. Where a Fortran file unit is returned set to -1, this means that the particular unit has not been opened and any available unit will be used when required.

See Also Appendix A

gEnqKeyState

Syntax

C/C++:	**int gEnqKeyState**(int key);

F90:	**integer function gEnqKeyState**(key) integer, intent(in) :: key

Arguments

key
key code

Description

The function gEnqKeyState() returns the status of a keyboard key at the time the function is called. The function returns 0 (zero) if the key is not pressed and 1 if it is.

The possible values of **key** are documented under gGetEventRecord() with the following additional special key codes:

key	Special Key Description
-1	Left mouse button
-2	Middle mouse button
-3	Right mouse button
-20	Shift
-21	Ctrl
-22	Alt
-30	Caps lock
-31	Num lock
-32	Scroll lock

This routine is only supported on windowing devices.

See Also

Page 457
gGetEventRecord

gEnqLastErrors

Syntax

C/C++: **void gEnqLastErrors**(int list[], int n, int *count);

F90: **subroutine gEnqLastErrors**(list, n, count)

integer, intent(in) :: n
integer, intent(out) :: list(*),count

Arguments

list
Integer array returning error and warning numbers

n
Number of error and warning numbers to return

= 0, Nothing returned in **list**

count
Number of errors logged since last call to gSetMaxErrorLimit()

Description GINO keeps a count of all errors. The count is reset whenever gSetMaxErrorLimit() is called. gEnqLastErrors() returns the current count in **count**. GINO also stores the 12 most recent error and warning numbers. Error numbers are positive, warning numbers are negative. gEnqLastErrors() returns **n** error and warning numbers. All numbers beyond those currently recorded are set to zero. The numbers returned in array **list** start with the most recent error or warning.

If **n** is less than or equal to zero, no numbers are returned in **list**. If **n** is less than zero, a warning message is output.

See Also Page 30
gSetMaxErrorLimit

gEnqLightAttribs

Syntax

C/C++: **void gEnqLightAttribs**(int light, GLITATT *att);

F90: **subroutine gEnqLightAttribs**(light, att)

integer, intent(in) :: light
type (GLITATT), intent(out) :: att

Arguments

light
Light source number (1-8)

att.state
Light state

= GON, Light is switched on
= GOFF, Light is switched off

att.type
Light type

= GAMBIENT
= GDIRECTIONAL
= GPOINTLIGHT
= GSPOTLIGHT

att.col
Light colour

att.dir
Direction vector

att.pos
Position

att.att1
Constant attenuation factor

att.att2
Linear attenuation factor

att.conc
Spot light concentration

att.spang
Spot light spread angle in degrees

att.spec
Specular colour component

Description The routine gEnqLightAttribs() returns the current attributes of the specified light source. The information is returned in a structure of type GLITATT, which contains values set by gDefineLightSource() and gSetLightSwitch().

See Also Page 333
gDefineLightSource
gSetLightSwitch

gEnqLineColour

Syntax

C/C++:	**void gEnqLineColour**(int *col);

F90:	**subroutine gEnqLineColour**(col) integer, intent(out) :: col

Arguments

col
Current colour index

Description The routine gEnqLineColour() returns the currently requested colour index set by gSetLineColour(). If **col** falls outside the range of colours of the device, the pen colour will be different from the requested colour index. The pen colour is determined by calling gEnqSelectedPen().

See Also Page 117
gSetLineColour
gEnqSelectedPen

gEnqLineEnd

Syntax

C/C++:	**void gEnqLineEnd**(int *end);

F90:	**subroutine gEnqLineEnd**(end) integer, intent(out) :: end

Arguments

end
Current line end type

= GNONE,	No ends
= GSQUARE,	Square ends
= GROUND,	Round ends
> 2,	No ends or device dependent

Description The routine gEnqLineEnd() returns the currently requested line end type. The default for the line end type is 'no ends'. The line end type may be changed by calling gSetLineEnd().

See Also Page 120
gSetLineEnd

gEnqLineStyle

Syntax

C/C++: **void gEnqLineStyle**(int line, GLINSTY *rep);

F90: **subroutine gEnqLineStyle**(line, rep)

integer, intent(in) :: line
type (GLINSTY), intent(out) :: rep

Arguments

line
Line style index

= GCURRENT,	Current line style
= 1 - 256,	Stored line style

rep
Line style representation

Description The routine gEnqLineStyle() returns the line attribute values defined by the specified line style. If **line** is zero or out of range, the current line attributes are returned. If **line** is out of range, a warning message is output. Otherwise, one of 256 sets of line attributes, stored in the line style table, is returned. The line style representation elements are discussed under the routine gDefineLineStyle().

See Also Page 129
gSetBrokenLine
gSetLineColour
gSetLineEnd
gSetLineVis
gSetLineWidth
gSetPenType

gEnqLineVis

Syntax

C/C++:	**void gEnqLineVis**(int *vis);

F90:	**subroutine gEnqLineVis**(vis) integer, intent(out) :: vis

Arguments

vis
Current line visibility

= GOFF,	Off
= GON,	On

Description

The routine gEnqLineVis() returns the current setting for the line visibility. The default is for line visibility to be switched on. The line visibility may be changed by calling gSetLineVis().

See Also

Page 116
gSetLineVis

gEnqLineWidth

Syntax

C/C++:	**void gEnqLineWidth**(float *width);

F90:	**subroutine gEnqLineWidth**(width) real, intent(out) :: width

Arguments

width
Current line width in current units

Description

The routine gEnqLineWidth() returns the currently requested line width. The default for the line width is either 0.2mm or the initial thickness for lines generated by the device. The line width may be changed by calling gSetLineWidth().

The actual thickness of lines generated by the device can be enquired by calling gEnqSelectedPen(). If the device is limited in its ability to generate lines of a specified thickness, the value returned by gEnqSelectedPen() will almost certainly differ from the currently requested line width.

See Also

Page 119
gEnqSelectedPen
gSetLineWidth

gEnqLineWidthMode

Syntax

C/C++:	**void gEnqLineWidthMode**(int *sw);

F90:	**subroutine gEnqLineWidthMode**(sw) integer, intent(out) :: sw

Arguments

sw
Thick line generation mode

= GHARDWARE,	Hardware thick line generation
= GMIXWARE,	Mixed hardware/software thick line generation
= GSOFTWARE,	Software thick line generation

Description

The routine gEnqLineWidthMode() returns the current thick line generation mode for lines greater than one device unit wide as set by the routine gSetLineWidthMode().

See Also

Page 119
gSetLineWidthMode

gEnqLineWidthScaling

Syntax

C/C++:	**void gEnqLineWidthScaling**(float *scale);

F90:	**subroutine gEnqLineWidthScaling**(scale) real, intent(out) :: scale

Arguments

scale
Line width scale factor

Description

The routine gEnqLineWidthScaling() returns the line width scale factor set by gSetLineWidthScaling(). The scale factor affects all settings of line width including user settings through gSetLineWidth() and internal settings for character underlining and font weight.

See Also

Page 120
gSetLineWidthScaling

gEnqMaskState

Syntax

C/C++: **void gEnqMaskState**(int *sw, GLIMIT *bounds);

F90: **subroutine gEnqMaskState**(sw, bounds)

integer, intent(out) :: sw
type (GLIMIT), intent(out) :: bounds

Arguments

sw
State of masking switch

= 0,	Masking off
= 1,	Masking on, rectangular limits returned
= 2,	Polygonal masking on, polygon rectangular boundary returned

bounds
Mask bounds

Description

The routine gEnqMaskState() returns the current masking state.

If the mask has been defined using gSetMask2D(), the mask limits are returned in **bounds**. If a polygonal mask has been defined using gSetPolygonMask() this routine only returns the rectangular extent of the defined polygons. The list of polygon identifiers can be obtained using gEnqPolygonMaskList().

If no mask has been defined with gSetMask2D() or gSetPolygonMask(), the masking state will be returned as off, irrespective of the call to gSetMaskMode(), and all its bounds are returned as 0.0.

See Also

Page 225
gSetMask2D
gSetPolygonMask
gEnqPolygonMaskList

gEnqMaterial

Syntax

C/C++: **void gEnqMaterial**(int mat, GMATSTY *rep);

F90: **subroutine gEnqMaterial**(mat, rep)

integer, intent(in) :: mat
type (GMATSTY), intent(out) :: rep

Arguments

mat
Material table index (1-256)

rep.ambient
Ambient reflection coefficient (0.0 - 1.0)

rep.diffuse
Diffuse reflection coefficient (0.0 - 1.0)

rep.specular
Specular reflection coefficient (0.0 - 1.0)

rep.shine
Specular concentration (shininess) (%)

rep.trans
Translucence (filtering) (0.0 - 1.0)

Description The routine gEnqMaterial() returns the material coefficients and values of an entry in the material table. These values are set using the routine gDefineMaterial().

See Also Page 341
gDefineMaterial

gEnqMaterialAttribs

Syntax

C/C++: **void gEnqMaterialAttribs**(int *fcol, int *bcol, int *fmat, int *bmat);

F90: **subroutine gEnqMaterialAttribs**(fcol, bcol, fmat, bmat)
integer, intent(out) :: fcol, bcol, fmat, bmat

Arguments

fcol
Front face material colour

bcol
Back face material colour

fmat
Front face material index

bmat
Back face material index

Description The routine gEnqMaterialAttribs() returns the current settings for the material colour and the material table indices for both front and back faces of facets. These parameters are set by the routines gSetMaterialColour() and gSetMaterialIndex() respectively.

See Also Page 341
gSetMaterialColour
gSetMaterialIndex

gEnqMaxDrawingLimits

Syntax

C/C++:	**void gEnqMaxDrawingLimits**(GDIM *dim);

F90:	**subroutine gEnqMaxDrawingLimits**(dim) type (GDIM), intent(out) :: dim

Arguments

dim
Maximum paper or device limits

Description

The routine gEnqMaxDrawingLimits() returns the maximum paper or workstation window limits for the currently nominated device.

These limits may be different to those returned by gEnqDrawingLimits(), even at device nomination, as gEnqDrawingLimits() always returns the current paper or workstation window limits which may be less than the maximum available on that device.

See Also

Page 46
gEnqDrawingLimits

gEnqMousePos

Syntax

C/C++:	**void gEnqMousePos**(int env, GPIXEL *point);

F90:	**subroutine gEnqMousePos**(env, point) integer, intent(in) :: env type (GPIXEL), intent(out) :: point

Arguments

env
Mouse position environment

GSCREEN	Relative to screen or display area
GDRAWINGAREA	Relative to current drawing area or device window

point
Mouse position in pixels

Description

The routine gEnqMousePos() enquires the current position of the graphics pointer or mouse in the structure **point** relative to the top left corner of the specified environment passed in **env**.

The routine gEnqPosOfPixel() can be used to translate the mouse position relative to the current drawing area or device window (GDRAWINGAREA) to a picture position and the routine gSetMousePos() can be used to set the current mouse position.

See Also

Page 456
gSetMousePos
gEnqPosOfPixel
Appendix B

gEnqNumberOfErrors

Syntax

C/C++:	**void gEnqNumberOfErrors**(int *count);

F90:	**subroutine gEnqNumberOfErrors**(count) integer, intent(out) :: count

Arguments

count

Number of trapped errors and warnings

= -1, Error trapping disabled

Description

The routine gEnqNumberOfErrors() returns the number of errors and warnings generated since the last call to gSetErrorTrap(1). If error trapping is disabled, **count** is returned set to -1. A call to gEnqLastErrors() will return the actual error and warning numbers generated (up to a maximum of 12).

See Also

Page 30
gEnqLastErrors
gSetErrorTrap

gEnqOpenSeg

Syntax

C/C++:	**void gEnqOpenSeg**(int *nseg);

F90:	**subroutine gEnqOpenSeg**(nseg) integer, intent(out) :: nseg

Arguments

nseg

Currently opened segment

Description

The routine gEnqOpenSeg() returns the currently opened segment as set by the routine gOpenSeg(). If no segment is currently open **nseg** returns a value of zero.

See Also

Page 427
gOpenSeg

gEnqPenType

Syntax

C/C++:	**void gEnqPenType**(int *type);

F90:	**subroutine gEnqPenType**(type) integer, intent(out) :: type

Arguments

type

Pen type

= 0, Unspecified

Description

The routine gEnqPenType() returns the currently requested pen type. Most devices will not provide all pen types so quite often the pen type made available will not correspond to the requested pen type. The actual pen type is determined by calling gEnqSelectedPen().

See Also

Page 120
gEnqSelectedPen
gSetPenType

gEnqPicturePos

Syntax

C/C++:	**void gEnqPicturePos**(GPOINT3 *point);

F90:	**subroutine gEnqPicturePos**(point) type (GPOINT3), intent(out) :: point

Arguments

point

The coordinates with respect to the picture axes and in the current units of the current drawing position

Description

The routine gEnqPicturePos() sets the argument **point** to the current drawing position in picture coordinates.

See Also

Page 238, 368

gEnqPixelAttribs

Syntax

C/C++:	**void gEnqPixelAttribs**(int *ori, float *xsca, float *ysca, int *xrep, int *yrep);

F90:	**subroutine gEnqPixelAttribs**(ori, xsca, ysca, xrep, yrep) integer, intent(out) :: ori real, intent(out) :: xsca,ysca integer, intent(out) :: xrep,yrep

Arguments

ori
An integer indicating the orientation of the pixel array

= 0,	None
= 1,	90 degrees anti-clockwise
= 2,	180 degrees anti-clockwise
= 3,	270 degrees anti-clockwise

xsca,ysca
X and Y scaling factors

xrep,yrep
Direction and number of pixels of replication

Description

gEnqPixelAttribs() returns the current pixel scaling and replication attributes as set by the routines gSetPixelTransform() and gSetPixelReplication().

The orientation and scale factors are set by the routine gSetPixelTransform() and the replication values are set by the routine gSetPixelReplication(). A fuller explanation of the returned variables can be found under the appropriate routine.

See Also

Page 202
gSetPixelReplication
gSetPixelTransform

gEnqPixelPacking

Syntax

C/C++:	**void gEnqPixelPacking**(int *nbp, int *nrb, int *npw, int *ndir, int *dir);

F90:	**subroutine gEnqPixelPacking**(nbp, nrb, npw, ndir, dir) integer, intent(out) :: nbp,nrb,npw,ndir,dir

Arguments

nbp
The number of bits per pixel

nrb
The number of relevant bits

npw
The number of pixels per word

ndir
Pixel order within machine word

= +1,	Normal direction
= -1,	Reverse direction

dir
Drawing direction

= 1,	Start top left, access horizontally
= 2,	Start top left, access vertically
= 3,	Start top right, access horizontally
= 4,	Start top right, access vertically
= 5,	Start bottom left, access horizontally
= 6,	Start bottom left, access vertically
= 7,	Start bottom right, access horizontally
= 8,	Start bottom right, access vertically

Description gEnqPixelPacking() returns the users pixel data characteristics as set by gDefinePixelPacking() and used by gDrawPixelArea() and gGetPixelArea(). It defines the form of bit packing or unpacking between the users data storage and the actual device.

See Also Page 202
gDefinePixelPacking
gDrawPixelArea
gGetPixelArea

gEnqPixelPos

Syntax

C/C++: **void gEnqPixelPos**(float xsc, float ysc, GPIXEL *pix);

F90: **subroutine gEnqPixelPos**(xsc, ysc, pix)

```
real, intent(in) :: xsc,ysc
type (GPIXEL), intent(out) :: pix
```

Arguments ***xsc,ysc***
Screen coordinates with respect to the picture axes within the device limits

pix
Returned pixel coordinates of the screen position

Description The routine gEnqPixelPos() returns the pixel coordinates (**pix**) of a screen position (**xsc,ysc**) with respect to the picture axes.

NB: The pixel coordinate system has its origin at the top left corner of the device.

See Also Page 198

gEnqPixelResolution

Syntax

C/C++:	**void gEnqPixelResolution**(int *nxpix, int *nypix);

F90:	**subroutine gEnqPixelResolution**(nxpix, nypix) integer, intent(out) :: nxpix,nypix

Arguments

nxpix
The number of pixels in the horizontal direction

nypix
The number of pixels in the vertical direction

Description

The routine gEnqPixelResolution() returns the pixel resolution for the current device limits.

The range of pixels is from 0 to **nxpix**-1 and 0 to **nypix**-1.

The pixel origin is at the top left of the device limits.

See Also

Page 190

gEnqPointMode

Syntax

C/C++:	**void gEnqPointMode**(int *switch);

F90:	**subroutine gEnqPointMode**(switch) integer, intent(out) :: switch

Arguments

switch
Point storage mode

= GOFF,	Switch point storing off
= GSPACE,	Points stored in space coordinates
= GPICTURE,	Points stored in picture coordinates

Description

The routine gEnqPointMode() returns the current point storage mode.

See Also

Page 104
gSetPointMode

gEnqPolygonList

Syntax

C/C++:	**void gEnqPolygonList**(int list[], int n, int *count);

F90:	**subroutine gEnqPolygonList**(list, n, count) integer, intent(in) :: n integer, intent(out) :: list(*),count

Arguments

list
Integer array returning selected polygon identifiers

n
Number of polygon identifiers to return

= 0, Nothing returned in **list**

count
Number of polygon identifiers in last call to gSelectPolygons()

Description

The routine gEnqPolygonList() returns a list of polygon identifiers that are currently selected for polygon filling with the gFillSelectedPolygons() routine. The polygons are selected using the routine gSelectPolygons().

The array **list** should be declared of length **n**, and no more than that number of identifiers are returned. **count** returns the actual number of polygon identifiers selected with gSelectPolygons() even if this number is greater then **n**. If **count** = 0 then all defined polygons are currently selected for filling.

If **n** is less than or equal to zero, no numbers are returned in **list**. If **n** is less than zero, a warning message is output.

See Also

Page 257
gSelectPolygons

gEnqPolygonMaskList

Syntax

C/C++:	**void gEnqPolygonMaskList**(int list[], int n, int *count);

F90:	**subroutine gEnqPolygonMaskList**(list, n, count) integer, intent(in) :: n integer, intent(out) :: list(*),count

Arguments

list
Integer array returning selected polygon identifiers

n
Number of polygon identifiers to return

= 0, Nothing returned in **list**

count
Number of polygon identifiers in last call to gSetPolygonMask()

Description The routine gEnqPolygonMaskList() returns a list of polygon identifiers that are currently selected for polygon masking. The polygons are selected using the routine gSetPolygonMask().

The array **list** should be declared of length **n**, and no more than that number of identifiers are returned. **count** returns the actual number of polygon identifiers selected with gSetPolygonMask() even if this number is greater then **n**. If **count** = 0 then polygon masking is not being used.

If **n** is less than or equal to zero, no numbers are returned in **list**. If **n** is less than zero, a warning message is output.

See Also Page 266
gSetPolygonMask

gEnqPolygonWindowList

Syntax

C/C++: **void gEnqPolygonWindowList**(int list[], int n, int *count);

F90: **subroutine gEnqPolygonWindowList**(list, n, count)

integer, intent(in) :: n
integer, intent(out) :: list(*),count

Arguments

list
Integer array returning selected polygon identifiers

n
Number of polygon identifiers to return

= 0, Nothing returned in **list**

count
Number of polygon identifiers in last call to gSetPolygonWindow()

Description The routine gEnqPolygonWindowList() returns a list of polygon identifiers that are currently selected for polygon windowing. The polygons are selected using the routine gSetPolygonWindow().

The array **list** should be declared of length **n**, and no more than that number of identifiers are returned. **count** returns the actual number of polygon identifiers selected with gSetPolygonWindow() even if this number is greater then **n**. If **count** = 0 then polygon windowing is not being used.

If **n** is less than or equal to zero, no numbers are returned in **list**. If **n** is less than zero, a warning message is output.

See Also

Page 266
gSetPolygonWindow

gEnqPolygonWorkspace

Syntax

C/C++:	**void gEnqPolygonWorkspace**(int *npoly, int *nvert, int *nfree, int *ident);

F90:	**subroutine gEnqPolygonWorkspace**(npoly, nvert, nfree, ident) integer, intent(out) :: npoly,nvert,nfree,ident

Arguments

npoly
Number of polygons defined so far

nvert
Number of vertices defined so far

nfree
Number of real words still available in the polygon workspace

ident
Current polygon identifier

Description

The routine gEnqPolygonWorkspace() returns information about the storage of polygons. **npoly** and **nvert** together describe the amount of space in gDefinePolygonWorkspace() that has been used. Each polygon requires a header of eight real words and each coordinate pair requires two real words.

Thus:

nw = 8***npoly** +2 ***nvert**+**nfree** = size of polygon workspace (see gDefinePolygonWorkspace())

ident is either 0 or the identifier defined by the last call to gSetPolygonIdent().

npoly, **nvert** and **nfree** are all set to zero if no polygon workspace has been defined, i.e. no call has been made to gDefinePolygonWorkspace().

See Also

Page 250
gDefinePolygonWorkspace
gSetPolygonIdent

gEnqPosOfPixel

Syntax

C/C++:	**void gEnqPosOfPixel**(int ix, int iy, GPOINT *point);

F90:	**subroutine gEnqPosOfPixel**(ix, iy, point) integer, intent(in) :: ix,iy type (GPOINT), intent(out) :: point

Arguments

ix,iy
Pixel position within the device coordinates

point
Returned screen coordinates of pixel position

Description

The routine gEnqPosOfPixel() returns the screen coordinates (**point**) of a pixel position (**ix,iy**).

NB: The pixel coordinate system has its origin at the top left corner of the device.

See Also

Page 198

gEnqQueueLength

Syntax

C/C++:	**void gEnqQueueLength**(int *len);

F90:	**subroutine gEnqQueueLength**(len) integer, intent(out) :: len

Arguments

len
The length of the event queue

= 0,	If no events are waiting
> 0,	If any events are waiting

Description

The routine gEnqQueueLength() is called to enquire whether an event is waiting in the queue to be read. If none, **len** is set to 0. If an event is waiting, **len** is set to the number of events in the queue. In some cases, the device may not know exactly how many events are waiting, in which case, **len** will be set to the minimum.

See Also

Page 456

gEnqRGB

Syntax

C/C++: **void gEnqRGB**(int col, GRGBSTY *rgb);

F90: **subroutine gEnqRGB**(col, rgb)

integer, intent(in) :: col
type (GRGBSTY), intent(out) :: rgb

Arguments

col
Colour index

< 0,	Dummy definition
= GBACKGROUND,	Background colour
> 0,	Index up to device capability

rgb
Red, green, blue colour components

Description

The routine gEnqRGB() returns the red, green, blue values identified by **col**. The colour may have been defined by a routine other than gDefineRGB(), in which case the colour coordinates are converted to RGB. If **col**<0, the colour values returned are those last specified by any of the colour definition routines.

The default values are returned by gEnqRGB() until the colour identified by **col** is redefined by any call to the colour definition routines.

The default values correspond to as many of the standard GINO colours as the device can implement. For plotters and other devices that do not cater for variable colours, or if **col** is out of range for the device, gEnqRGB() returns **rgb.red**, **rgb.green** and **rgb.blue** set to zero.

See Also

Page 212
gDefineHLS
gDefineHSV
gDefineRGB

gEnqSavdraDimension

Syntax

C/C++: **void gEnqSavdraDimension**(GFILE *fp, int *type, GDIM *dim);

F90: **subroutine gEnqSavdraDimension**(unit, type, dim)

integer, intent(in) :: unit
integer, intent(out) :: type
type (GDIM), intent(out) :: dim

Arguments

fp
GINO-C file pointer

unit
Fortran 90 File unit

type
Type of SAVDRA metafile

= 1,	SAVDRA metafile
= 2,	SAVPIC metafile

dim
Paper size of SAVDRA metafile

Description

The routine gEnqSavdraDimension() returns the type and paper limits of a GINO SAVDRA metafile without interpreting the whole file.

The metafile should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

See Also

Page 64
gFopen

gEnqSavdraSegAttribs

Syntax

C/C++: **void gEnqSavdraSegAttribs**(GFILE *fp, int nseg, GPICATT *att);

F90: **subroutine gEnqSavdraSegAttribs**(unit, nseg, att)

integer, intent(in) :: unit,nseg
type (GPICATT), intent(out) :: att

Arguments

fp
GINO-C file pointer

unit
Fortran 90 File unit

nseg
Picture segment number

att.exist
Flag indicating existence of segment

= 0,	Segment **nseg** does not exist
= 1,	Segment **nseg** exists
= 2,	Segment **nseg** exists as a group

att.vis
Visibility status

= GOFF,	invisible (not displayed)
= GON,	visible (displayed)

att.sens
Sensitivity status

= GOFF,	Non hit-sensitive
= GON,	Hit-sensitive

att.mark
'Marked' status

= GUNMARK,	Not marked
= GMARK,	Marked

att.anchor
Picture anchor point (GPOINT3 structure)

Description

The routine gEnqSavdraSegAttribs() returns the segment attributes of segment **nseg** in a GINO SAVDRA metafile. The metafile is scanned until the requested segment is found but no graphical information is interpreted.

The metafile should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

See Also

Page 65
gFopen

gEnqSavdraSegList

Syntax

C/C++: **void gEnqSavdraSegList**(GFILE *fp, int list[], int n, int *count);

F90:
```
subroutine gEnqSavdraSegList((unit, list, n, count)

integer, intent(in) :: unit,n
integer, intent(out) :: list(*),count
```

Arguments

fp
GINO-C file pointer

unit
Fortran 90 File unit

list
Integer array returning segment identifiers

n
Number of segment identifiers to return

= 0,	Nothing returned in **list**

count
Number of segment identifiers contained in the metafile

Description

The routine gEnqSavdraSegList() returns a list of segments contained in the opened SAVDRA metafile. The complete metafile is scanned but no graphical information is interpreted.

The metafile should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

The array **list** should be declared of length **n**, and no more than that number of identifiers are returned. The argument **count** returns the actual number of segment identifiers contained in the file even if this number is greater then **n**. If **count** = 0 then the metafile only contains information in segment zero.

If **n** is less than or equal to zero, no numbers are returned in **list**. If **n** is less than zero, a warning message is output.

See Also

Page 65
gEnqSavdraSegAttribs

gEnqSegAttribs

Syntax

C/C++: **void gEnqSegAttribs**(int nseg, GPICATT *att);

F90: **subroutine gEnqSegAttribs**(nseg, att)

integer, intent(in) :: nseg
type (GPICATT), intent(out) :: att

Arguments

nseg
Picture segment number

att.exist
Flag indicating existence of segment

= 0,	Segment **nseg** does not exist
= 1,	Segment **nseg** exists
= 2,	Segment **nseg** exists as a group

att.vis
Visibility status

= GOFF,	invisible (not displayed)
= GON,	visible (displayed)

att.sens
Sensitivity status

= GOFF,	Non hit-sensitive
= GON,	Hit-sensitive

att.mark
'Marked' status

= GUNMARK, Not marked
= GMARK, Marked

att.anchor
Picture anchor point (GPOINT3 structure)

Description

The routine gEnqSegAttribs() returns the segment attributes of segment **nseg** in the structure **att**.

Where possible this information is obtained from the display file held in the device. If the software display file has been activated using gSetSegMode(), any information not obtainable from the hardware will be supplemented from information held in the software display file. If no display file is being used or the segment doesn't exist all arguments are returned as zero.

Where **nseg** has been defined as a group with gDefineSegGroup(), **att.exist** is returned as 2 and all other arguments are returned as zero.

See Also

Page 432
gDefineSegGroup
gSetSegVis
gSetSegHit
gMarkSeg
gInsertSegRef
gOpenSeg
gSetSegMode

gEnqSegGroup

Syntax

C/C++:	**void gEnqSegGroup**(int ngrp, int *segmin, int *segmax);

F90:	**subroutine gEnqSegGroup**(ngrp, segmin, segmax) integer, intent(in) :: ngrp integer, intent(out) :: segmin,segmax

Arguments

ngrp
Segment group number

segmin
Start segment number

segmax
End segment number

Description

The routine gEnqSegGroup() returns the range of segment numbers defined for segment group **ngrp**. A segment group is defined by calling gDefineSegGroup() and removed by calling gRemoveSegGroup(). **ngrp** should lie within the range defined by a call to gDefineGroupRange(). If gDefineGroupRange() has not been called, the range defaults to 1 to 32767. If **ngrp** is out of range, **segmin** and **segmax** are returned set to -1.

If the specified segment group does not exist, **segmin** and **segmax** are returned set to zero. Otherwise, **segmin** and **segmax** return the range of segment numbers defined for the segment group.

See Also

Page 438
gRemoveSegGroup
gDefineSegGroup
gDefineGroupRange

gEnqSegHit

Syntax

C/C++: **void gEnqSegHit**(int *nseg, float x, float y, float radius);

F90: **subroutine gEnqSegHit**(nseg, x, y, radius)

integer, intent(out) :: nseg
real, intent(in) :: x,y,radius

Arguments

nseg
Picture segment number

= -1, No picture segment in hit area

x,y
Picture coordinates of hit centre

radius
Radius of hit area in current units

Description

The routine gEnqSegHit() simulates a light pen hit in a picture segment. The display buffer is searched for picture segments that are hit sensitive (see gSetSegHit()) and that cross the hit area **radius**. If none are found **nseg** will be set to -1. The hit area is centred at **x,y** and has a radius specified by **radius**. If more than one picture segment crosses the hit area, either the one closest to the hit centre or the next in the display file is chosen.

Some devices that support segments do not support this routine in hardware and software emulation is required.

See Also

Page 439
gSetSegHit
gSetSegMode

gEnqSegTransform

Syntax

C/C++:	**void gEnqSegTransform**(int nseg, float *xsca, float *ysca, float *ang, GPOINT *pos); **void gEnqSegTransform2D**(int nseg, GMAT2D a);

F90:	**subroutine gEnqSegTransform**(nseg, xsca, ysca, ang, pos) **subroutine gEnqSegTransform2D**(nseg,a) integer, intent(in) :: nseg real, intent(out) :: xsca,ysca,ang type (GPOINT), intent(out) :: pos real, intent(out) :: a(6)

Arguments

nseg
Segment name

xsca
Scaling factor for X coordinates

ysca
Scaling factor for Y coordinates

ang
Angle of rotation about anchor point

pos
Received coordinates of current anchor point

a
3 x 2 array to receive the 2-D segment transformation matrix

Description

The routines gEnqSegTransform() and gEnqSegTransform2D() return the current segment transformation for segment **nseg** either as individual transformation elements or as a 2D transformation matrix.

These routines match the equivalent routines to set the segment transformation matrix, however, gEnqSegTransform() can only be used if the segment transformation was set using gSetSegTransform() which also uses separate values for the scale, rotation and position elements. gEnqSegTransfrom2D() can be used whichever routine was used to set the segment transform.

Where possible this information is obtained from the display file held in the device. If the software display file has been activated using gSetSegMode(), any information not obtainable from the hardware will be supplemented from information held in the software display file.

If no display file is being used or the segment doesn't exist, all arguments are returned as zero.

See Also

Page 432
gSetSegMode
gSetSegTransform

gEnqSegWorkspace

Syntax

C/C++:	**void gEnqSegWorkspace**(int *nw, int *nfree);

F90:	**subroutine gEnqSegWorkspace**(nw, nfree) integer, intent(out) :: nw,nfree

Arguments

nw
Number of real words set aside for storing Software Display File

= 0,	Software Display File not defined in memory
>0,	Size of display file

nfree
Number of free words of memory in the Software Display File

Description

The routine gEnqSegWorkspace() returns the number of real words that have been allocated for use by the Software Display File and the amount of free space in that memory. The space is allocated by the routine gDefineSegWorkspace().

gEnqSegWorkspace() will return a value of zero if gDefineSegWorkspace() has not been called, but the Software Display File may still be active in file mode.

See Also

Page 426
gDefineSegWorkspace

gEnqSelectedPen

Syntax

C/C++:	**void gEnqSelectedPen**(int *col, float *width, int *type);

F90:	**subroutine gEnqSelectedPen**(col, width, type) integer, intent(out) :: col,type real, intent(out) :: width

Arguments

col
Colour setting (see gSetLineColour())

width
Pen width in current units (see gSetLineWidth())

type
Pen type (see gSetPenType())

= 0,	No pen types implemented

Description The routine gEnqSelectedPen() returns the physical pen attributes currently implemented by the device. Devices cannot always implement colour, width and pen type exactly as requested and this is always reflected in the values returned by gEnqSelectedPen(). The colour and pen type requested simply may not be available or, for devices like plotters where the three attributes are associated, the exact combination of attributes may not be available. The **width** value returned is the pen width in current units.

Note that in the case of direct colour devices, the return value of **col** will contain a 24bit RGB triplet irrespective of whether the line colour was selected using a colour identifier, a colour table index or a 24bit RGB triplet.

See Also Page 113
gSetLineColour
gSetPenType

gEnqShadingMode

Syntax

C/C++:	**void gEnqShadingMode**(GSHADING *att);
F90:	**subroutine gEnqShadingMode**(att) type (GSHADING), intent(out) :: att

Arguments

att.mode
Shading mode

= GNONE,	No shading (default)
= GFLAT,	Flat shading (using facet normals)
= GGOURAUD,	Smooth (Gouraud) shading (using vertex normals)
= GPHONG,	Phong shading (using interpolated normals)

att.culling
Culling mode

= GOFF,	No culling - Two sided lighting (default)
= GBACK,	Ignore back (clockwise) facing polygons
= GFRONT,	Ignore front (anti-clockwise) facing polygons

att.blending
Blending mode

= GOFF,	Blending disabled (default)
= GON,	Blending enabled

att.winding
Facet winding mode

= GANTICLOCKWISE,	Anti-clockwise winding = front face (default)
= GCLOCKWISE,	Clockwise winding = front face

Description The routine gEnqShadingMode() returns the current lighting and shading settings that affect the display of 3D shaded objects.

See Also Page 327
gSetDepthMode
gSetShadingMode

gEnqSpacePos

Syntax

C/C++:	**void gEnqSpacePos**(GPOINT3 *point);

F90:	**subroutine gEnqSpacePos**(point) type (GPOINT3), intent(out) :: point

Arguments ***point***
The coordinates with respect to the space axes and in the current units of the current drawing position

Description The routine gEnqSpacePos() sets the argument **point** to the current drawing position in space coordinates.

See Also Page 238, 368

gEnqSplineTension

Syntax

C/C++:	**void gEnqSplineTension**(float tension);

F90:	**subroutine gEnqSplineTension**(tension) real, intent(out) :: tension

Arguments ***tension***
Spline curve tension

Description The routine gEnqSplineTension() returns the current setting of the spline curve tension as set by the routine gSetSplineTension().

See Also Page 100, 287
gSetSplineTension

gEnqStrExponent

Syntax

C/C++:	**void gEnqStrExponent**(float *relcw, float *relch, float *posexp, float *posind);

F90:	**subroutine gEnqStrExponent**(relcw, relch, posexp, posind) real, intent(out) :: relcw,relch,posexp,posind

Arguments ***relcw, relch***
Relative character size for exponents and indices

posexp
Relative character position for exponents

posind
Relative character position for indices

Description The routine gEnqStrExponent() returns the current settings for string exponent attributes set by gSetStrExponent().

The default size of string exponents and indices is 0.7 times the current character size, with exponents positioned at 0.6 times the current character height above the base line and indices positioned 0.3 times the current character height below the base line.

See Also Page 155
gSetStrExponent

gEnqStrJustify

Syntax

C/C++:	**void gEnqStrJustify**(int *jus);

F90:	**subroutine gEnqStrJustify**(jus) integer, intent(out) :: jus

Arguments

jus
String justification setting

= GLEFT,	Left-justified
= GCENTRE,	Centre-justified
= GRIGHT,	Right-justified

Description The routine gEnqStrJustify() returns the current string justification setting as set by gSetStrJustify().

The default setting is for strings to be left justified.

See Also Page 153
gSetStrJustify

gEnqStrUnderscore

Syntax

C/C++:	**void gEnqStrUnderscore**(int *und);

F90:	**subroutine gEnqStrUnderscore**(und) integer, intent(out) :: und

Arguments

und
String underscore switch

= GOFF,	Underscore off
= GON,	Underscore on

Description The routine gEnqStrUnderscore() returns the current string underscore switch as set by gSetStrUnderscore().

Underscoring can also be switched on within strings using the *S escape sequence, but gEnqStrUnderscore() is not able to detect this as the switch is switched off at the end of the string in these circumstances.

The default setting is for strings with no underscoring.

See Also Page 152
gSetStrUnderscore

gEnqSysArgs

Syntax

C/C++: void

F90:
```
subroutine gEnqSysArgs(n, args)

integer, intent(inout) :: n
character*(*), intent(out) :: args(*)
```

Arguments

n
Size of args array on entry, number of elements filled on exit

args
Character array containing application name and any command line arguments

Description The system utility gEnqSysArgs() is available to F90 programmers to enquire the application program name and any command line arguments entered when starting the program. The application name is returned in args(1) and arguments in args(2), args(3), etc.
C programmers should use the standard argc and argv parameters to main().

See Also Page 463

gEnqSysDate

Syntax

C/C++:
```
void gEnqSysDate(GDATE *date);
char * gEnqSysDateStr(char datestr[], int slen);
```

F90:
```
subroutine gEnqSysDate(date)
subroutine gEnqSysDateStr(datestr)

type (GDATE), intent(out) :: date
character*(*), intent(out) :: datestr
```

Arguments

date
Current year/month/day

datestr
System date/time string

slen
Length of date string (C/C++ only)

Description

The system utilities gEnqSysDate() and gEnqSysDateStr() return the current system date as either a structure containing year, month and day integer elements or as a string.

The routine gEnqSysDateStr() returns the system date/time as a character string, the format of which is system dependent.

The C/C++ binding of gEnqSysDateStr() returns the date either in the string **datestr** of length **slen** or (if **datestr** is NULL) as a pointer to a character string that has been malloc'ed internally and which should be freed after use with the function free. In the Fortran 90 binding the date is returned in the character argument **datestr.** If this is not large enough to contain the date it will be truncated.

See Also

Page 463
gEnqSysTime

gEnqSysEnviron

Syntax

C/C++: **char * gEnqSysEnviron**(char envnam[], char setting[], int slen);

F90: **subroutine gEnqSysEnviron**(envnam, setting)

character*(*), intent(in) :: envnam
character*(*), intent(out) :: setting

Arguments

envnam
Environment string

setting
Environment setting

slen
Length of string setting (C/C++ only)

Description

The system utility gEnqSysEnviron() can be used to enquire the setting of a system environment variable. The environment variable name is passed in through the string **envnam**, and its setting, if any, is returned in **setting**. If there is no setting of the requested environment variable, a NULL/blank string is returned.

The C/C++ binding returns the environment setting either in the string **setting** of length **slen** or (if **setting** is NULL) as a pointer to a character string that has been malloc'ed internally and which should be freed after use with the function free. In the Fortran 90 binding the environment setting is returned in the character argument **setting.** If this is not large enough to contain the setting it will be truncated.

See Also Page 464

gEnqSysPriority

Syntax

C/C++:	**void gEnqSysPriority**(int *pri);

F90:	**subroutine gEnqSysPriority**(pri) integer, intent(out) :: pri

Arguments

pri
Task priority

= GREALTIME,	Highest possible priority
= GHIGH,	Higher than normal
= GNORMAL,	(default)
= GLOW,	Lower than normal
= GIDLE,	Idle state

Description

The routine gEnqSysPriority() returns the current priority of the GINO application as set by the routine gSetSysPrioirty().

See Also

Page 466
gSetSysPriority

gEnqSysTime

Syntax

C/C++:	**void gEnqSysTime**(GTIME *time);

F90:	**subroutine gEnqSysTime**(time) type (GTIME), intent(out) :: time

Arguments

time.hour, time.min, time.sec, time.millsec
Current hour/minute/seconds/millisecs

Description

The system utility gEnqSysTime() returns the current system time in terms of hours, minutes, seconds and milliseconds.

See Also

Page 463
gEnqSysDate

gEnqSysUsername

Syntax

C/C++:	**char * gEnqSysUsername**(char uname[], int slen);

F90:	**subroutine gEnqSysUsername**(uname) character*(*), intent(out) :: uname

Arguments

uname
Current user name or system id.

slen
Length of user name string (C/C++ only)

Description

The system utility gEnqSysUsername() returns an appropriate user name of the current application. In most multi-user implementations, the login name is returned, but in single user systems, such as PC's, an implementation dependent string is returned.

The C/C++ binding returns the user name either in the string **uname** of length **slen** or (if **uname** is NULL) as a pointer to a character string that has been malloc'ed internally and which should be freed after use with the function free. In the Fortran 90 binding the user name is returned in the character argument **uname.** If this is not large enough to contain the user name it will be truncated.

See Also

Page 464

gEnqTextBlockAttribs

Syntax

C/C++:	**void gEnqTextBlockAttribs**(float *xbeg, float *ybeg, float *drpfac);

F90:	**subroutine gEnqTextBlockAttribs**(xbeg, ybeg, drpfac) real, intent(out) :: xbeg,ybeg,drpfac

Arguments

xbeg,ybeg
Text block start position

drpfac
Inter-line spacing factor

Description

The routine gEnqTextBlockAttribs() returns the current settings for text block attributes as set by gStartTextBlock() and gSetInterlineSpace().

The default inter-line spacing factor is 2.0, but there is no default setting for the text block starting position. If gStartTextBlock() has not been called **xbeg**, **ybeg** are both returned as -9999.0.

See Also Page 155
gStartTextBlock
gSetInterlineSpace

gEnqTextureCoordGeneration

Syntax

C/C++:	**void gEnqTextureCoordGeneration**(int mode, ...);

F90:	**subroutine gEnqTextureCoordGeneration**(mode, gSVec, gTVec) integer, intent(out) :: mode type (GTEXVEC), optional, intent(out) :: gSVec,gTVec

Arguments

mode
Texture coordinate generation mode

= GOFF,	Texture coordinate generation off (default)
= GOBJECT,	Object coordinates used
= GSPHERICAL,	Spherical texture coordinates

Optional Args

gSVec, gTVec
Object coordinate transformation vectors for S and T texture coordinates

Description

The routine gEnqTextureCoordGeneration() returns the current GINO texture coordinate generation mode. Optionally the object coordinate transformation vectors may also be enquired, but these are only relevant when the generation mode is currently set to GOBJECT.

Note that C/C++ users should pass the address of a structure of type GTEXVEC along with the **gSVec** and/or **gTVec** arguments in a similar manner to the use of gSetTextureCoordGeneration.

See Also

Page 351
gSetTextureCoordGeneration

gEnqTextureMappingMode

Syntax

C/C++:	**void gEnqTextureMappingMode**(GTEXATT *att);

F90:	**subroutine gEnqTextureMappingMode**(att) type (GTEXATT), intent(out) :: att

Arguments

att.mode
Texture mapping mode

= GOFF,	Texture mapping is off (default)
= GOVERLAY,	Overlay texture on surface
= GMODULATE,	Modulate texture colours with surface colour
= GBLEND,	Blend texture with constant blend colour

att.blendcol
Blend colour

att.wraps
Texture wrapping switch in S direction

= GREPEAT,	Repeat texture map (default)
= GCLAMP,	Clamp texture map

att.wrapt
Texture wrapping switch in T direction

= GREPEAT,	Repeat texture map (default)
= GCLAMP,	Clamp texture map

att.maxfil
Filter when enlarging texture map

= GNEAREST,	Use nearest texel (default)
= GLINEAR,	Use weighted average of 2x2 texels

att.minfil
Filter when reducing texture map

= GNEAREST,	Use nearest texel (default)
= GLINEAR,	Use weighted average of 2x2 texels
= GNEARESTNEAREST,	Nearest mipmap using nearest texel filter
= GNEARESTLINEAR,	Nearest mipmap using linear texel filter
= GLINEARNEAREST,	Linear interpolate mipmap using nearest texel filter
= GLINEARLINEAR,	Linear interpolate mipmap and linear texel filter

att.bordercol
Texture map border colour

Description The routine gEnqTextureMappingMode() returns the current GINO texture mapping mode and its attributes.

See Also Page 358
gSetTextureMappingMode

gEnqTransformState

Syntax

C/C++:	**void gEnqTransformState**(int *ntran, int *dim, int *mode);

F90:	**subroutine gEnqTransformState**(ntran, dim, mode) integer, intent(out) :: ntran,dim,mode

Arguments

ntran
State of modelling transformation switch

= GOFF,	Transforming off
= GON,	Transforming on

dim
Number of dimensions in use

= GOFF,	Transforming off
= 2,	2-D
= 3,	3-D with no perspective
= -3,	3-D with perspective

mode
State of picture mode switch

= GSPACE,	Space mode
= GPICTURE,	Picture mode

Description The routine gEnqTransformState() returns the current state of transforming.

See Also Page 381

gEnqViewport

Syntax

C/C++:	**void gEnqViewport2D**(GLIMIT *piclim2, GLIMIT *viewlim); **void gEnqViewport3D**(GLIMIT3 *piclim3, GLIMIT *viewlim);

F90:	**subroutine gEnqViewport2D**(piclim2, viewlim) **subroutine gEnqViewport3D**(piclim3, viewlim) type (GLIMIT), intent(out) :: piclim2,viewlim type (GLIMIT3), intent(out) :: piclim3

Arguments ***piclim2, piclim3***
Picture coordinate range

viewlim
Viewport coordinate range in current paper units

Description The routine gEnqViewport2D() returns both the ranges of user picture coordinates and viewport coordinates used in calculating the viewport mapping.

The values returned are either those set by gSetViewport2D() or if gSetViewport2D() has not been called, the default values for both ranges are the same as the current paper limits as set by gSetDrawingLimits() or the device defaults.

See Also Page 51, 221
Page 274
gSetViewport2D
gSetViewport3D

gEnqViewportMode

Syntax

C/C++:	**void gEnqViewportMode**(int *sw);

F90:	**subroutine gEnqViewportMode**(sw) integer, intent(out) :: sw

Arguments

sw
Viewport scaling switch

= GCENTRAL,	Keep aspect ratio and centre in viewport
= GBOTTOMLEFT,	Keep aspect ratio and place at bottom left of viewport
= GDEFORMED,	Allow deformation of picture

Description

The routine gEnqViewportMode() returns the viewport scaling switch as set by gSetViewportMode(). The switch determines whether a viewport transformation set by gSetViewport2D() should keep the aspect ratio of the picture coordinate area or allow deformation.

See Also

Page 50, 220
gSetViewportMode

gEnqViewportState

Syntax

C/C++:	**void gEnqViewportState**(int *swi, int *clp, GLIMIT *limit);

F90:	**subroutine gEnqViewportState**(swi, clp, limit) integer, intent(out) :: swi,clp type (GLIMIT), intent(out) :: limit

Arguments

swi
Viewport scaling switch

= GCENTRAL,	Keep aspect ratio and centre in viewport
= GBOTTOMLEFT,	Keep aspect ratio and place at bottom left of viewport
= GDEFORMED,	Allow deformation of picture

clp
Viewport clipping switch

= GOFF,	Do not clip to viewport limits
= GON,	Clip to viewport limits

limit
Viewport coordinate range in current paper units

Description The routine gEnqViewportState() returns the current viewport state.

The variable **swi** returns the viewport scaling switch as set by gSetViewportMode(). The switch determines whether a viewport transformation set by gSetViewport2D() should keep the aspect ratio of the picture coordinate area or allow deformation.

The variable **clp** returns the setting of the viewport clipping switch as set by gSetViewportClipSwitch(). The default setting of **clp** = GON implies that the setting of a viewport with gSetViewport2D() will also set the clipping limits to the same area, effectively restricting drawing to those viewport limits. Where the viewport clipping switch is set to GOFF, any future viewport setting will act simply as a mapping operation and window limits may extend outside the viewport limits.

The remaining arguments return the actual viewport limits of the current viewport in paper units. This may vary from the requested viewport limits returned by gEnqViewport2D() depending on the current viewport scaling switch (**swi**) but represents the actual limits of the specified picture region on the drawing area.

See Also Page 51, 221
gSetViewport2D
gEnqViewport2D
gSetViewportClipSwitch
gSetViewportMode

gEnqViewTransformMode

Syntax

C/C++:	**void gEnqViewTransformMode**(int *mode);

F90:	**subroutine gEnqViewTransformMode**(mode) integer, intent(out) :: mode

Arguments

mode
Viewing/Transformation mode

= GHARD,	Hardware viewing/transformation mode
= GSOFT,	Software viewing/transformation mode

Description This routine returns the current viewing/transformation mode of the currently nominated device.

See Also Page 371
gSetViewTransformMode

gEnqWindowState

Syntax

C/C++:	**void gEnqWindowState**(int *sw, GLIMIT3 *bounds);

F90:	**subroutine gEnqWindowState**(sw, bounds) integer, intent(out) :: sw type (GLIMIT3), intent(out) :: bounds

Arguments

sw
State of windowing switch

= 0,	Windowing off, viewport limits returned
= 1,	Windowing on, user-defined limits returned
= 2,	Windowing on and set to viewport limits, viewport limits returned
= 3,	Windowing on and set to polygon, polygon boundary returned

bounds
Current window limits

Description

The routine gEnqWindowState() returns the current windowing state.

For a rectangular window set by gSetWindowMode(), gSetWindow2D() or gSetWindow3D(), the window limits are returned in **bounds**. For a polygonal window defined by gSetPolygonWindow() this routine only returns the rectangular extent of the defined polygons. The list of polygon identifiers can be obtained through gEnqPolygonWindowList().

See Also

Page 224, 275
gSetPolygonWindow
gEnqPolygonWindowList
gSetWindow2D
gSetWindow3D
gSetWindowMode

gEnqWorkingDir

Syntax

C/C++:	**char * gEnqWorkingDir**(char directory[], int slen);

F90:	**subroutine gEnqWorkingDir**(directory) character*(*), intent(out) :: directory

Arguments

directory
Current directory

slen
Length of directory string (C/C++ only)

Description The system utility gEnqWorkingDir() returns the current operating directory of the GINO application. This may be the directory in which the application was initiated, or one set by the routine gSetWorkingDir().

The C/C++ binding returns the directory name either in the string **directory** of length **slen** or (if **directory** is NULL) as a pointer to a character string that has been malloc'ed internally and which should be freed after use with the function free. In the Fortran 90 binding the directory name is returned in the character argument **directory.** If this is not large enough to contain the directory name it will be truncated.

If the current working directory cannot be obtained by the system, this function will return a NUL/blank string.

See Also Page 460
gSetWorkingDir

gEnqWorkspaceLimit

Syntax

C/C++:	**void gEnqWorkspaceLimit**(int *n);

F90:	**subroutine gEnqWorkspaceLimit**(nl1, nl2) integer, intent(out) :: nl1,nl2

Arguments

n
Number of real words reserved for use as a workspace area

= 0, No workspace area defined

nl1,nl2
Limits of global workspace area

Description The routine gEnqWorkspaceLimit() returns the size of GINO's workspace area, if it has been defined by gSetWorkspaceLimit(). Otherwise **n** is returned set to zero.

See Also Page 36, 53, 248
gSetWorkspaceLimit

gExecuteSysCommand

Syntax

C/C++:	**int gExecuteSysCommand**(char command[], ...);

F90:	**integer function gExecuteSysCommand**(command, gShow, gSuspend, gHandle) character*(*), intent(in) :: command integer, optional, intnent(in) :: gShow, gSuspend integer, optional, intent(out) :: gHandle

Arguments

command
System command string

Optional Args

gShow
Visible state of Windows process

= GHIDDEN,	Run as background process (no window)
= GNORMAL,	Run in normal window (default)
= GMINIMIZE,	Run in minimized/iconized window
= GMAXIMIZE,	Run in maximized window

gSuspend
Suspension state of calling application

= GON,	Suspend calling application while process runs
= GOFF,	Return control to calling application immediately after starting new process (i.e. Run new process in parallel). This is the default.

gHandle
Handle of new process (available when not suspending calling application)

Description

The system utility gExecuteSysCommand() provides a means to execute a system command or process from within a GINO application. By default, the function will create a new process on the host machine to execute the command and control will be immediately passed back to the GINO application. In a Windows environment, the process will be run in a normal window, and in all enviroments output from the command will be directed to the standard output stream unless re-directed using appropriate operating system commands or symbols.

The optional argument **gShow** may be used in a Windows environment to alter the initial state of the window in which the process is run.

Where the new process is running in parallel with the GINO application, its process handle way be obtained by using the optional output argument **gHandle**. This may then be used in susbsequent system commands or routines to track its progress or halt it if required. The handle is not available if the calling application is suspended whilst the separate process runs.

The routine returns a value which is set to a system dependent non-zero value if the command was illegal or failed for some reason.

See Also

Page 465

gExtendSeg

Syntax

C/C++:	**void gExtendSeg**(int nseg);

F90:	**subroutine gExtendSeg**(nseg) integer, intent(in) :: nseg

Arguments

nseg
Picture segment number

Description The routine gExtendSeg() reopens segment **nseg** so that more drawing may be added to it. If segment **nseg** does not exist, a gOpenSeg(nseg) is invoked.

On metafile output, gExtendSeg(nseg) is equivalent to gOpenSeg(nseg).

See Also Page 427
gOpenSeg

gFclose

Syntax

C/C++:	**int gFclose**(GFILE *fp);

F90:	**integer function gFclose**(unit) integer, intent(in) :: unit

Arguments

fp
GINO-C file pointer

unit
F90 file unit

Description The routine gFclose() closes the GINO file unit that was opened with gFopen(). The routine returns 0 if the close is successful or an error code if not. In the C library this will be EOF (from <stdio.h>).

See Also Page 27, 28
gFopen

gFillPolygon

Syntax

C/C++:	**void gFillPolygonBy2D**(int fill, int line, int inv, int npts, GPOINT *points2); **void gFillPolygonBy3D**(int fill, int line, int inv, int npts, GPOINT3 *points3); **void gFillPolygonTo2D**(int fill, int line, int inv, int npts, GPOINT *points2); **void gFillPolygonTo3D**(int fill, int line, int inv, int npts, GPOINT3 *points3);

F90:	**subroutine gFillPolygonBy2D(**fill ,line, inv, npts, points2) **subroutine gFillPolygonBy3D**(fill, line, inv, npts, points3) **subroutine gFillPolygonTo2D**(fill, line, inv, npts, points2) **subroutine gFillPolygonTo3D**(fill, line, inv, npts, points3) integer, intent(in) :: fill,line,inv,npts type (GPOINT), intent(in) :: points2(*) type (GPOINT3), intent(in) :: points3(*)

Arguments

fill
Fill style index

= GHOLLOW, Draw boundary only

= GSOLID,	Solid fill
= 1 - 256,	Fill style index (hardware) or hatch style index (software)
> 256,	Fill style index (hardware) or solid fill (software)

line
Line style index

= GCURRENT,	Current line style
= 1 - 256,	Line style index
> 256,	Current line style

inv
Inverse fill flag

= GAREA,	Fill polygon
= GINVERSE,	Fill outside polygon (inverse fill)

npts
Number of points in **points** array

points2,points3
2D or 3D coordinate arrays specifying relative or absolute points of polygon boundary

Description

The gFillPolygon set of routines fill a single polygonal area without the need to declare any workspace. Coordinates may be supplied as relative or absolute coordinates in 2 or 3 dimensions. Whether relative or absolute coordinates are used, the complete polygon includes the current drawing position, all the points in the array **points2/3** and an additional point (if necessary) to ensure the last point is the same as the first point. All the points are then clipped to the current clipping limits.

Up to2048 points can be used to define the polygon and if **npts** is greater than this, an error message is generated and no output is done.

fill defines the fill style. The constant GHOLLOW specifies that only the boundary is to be drawn and GSOLID defines solid fill. Values between 1 and 256 point to the entries in the hatch style table (see gDefineHatchStyle()) and thus specifies the hatch style for the fill. Whether an area is filled by software or hardware is determined by gSetFillMode(). If hardware fill is enabled, **fill** may be interpreted in a device dependent way.

By default, hardware fill is attempted. If it is not possible to fill using the device's hardware then GINO defaults to the hatch style specified by **fill**. If **fill** is out of range for software filling, a solid fill is used by default.

line defines the line style used for the boundary or fill style. A value of **line** between 1 and 256 points to an entry in the line style table (see gDefineLineStyle()) and specifies additional attributes for the fill, usually a colour for hardware fill and a line type and colour for hatching. The current line style is left unchanged.

A value for **line** which is zero or greater than 256 specifies the current line style.

A value for **line** less that zero or **fill** less than -1 generates a warning message and its positive value is used.

inv defines whether the inside of the polygon is filled, or if **inv** = GINVERSE, the inverse area is filled up to the current clipping limits.

The current position is restored to that prior to calling one of these routines.

See Also

Page 167, 287
gSetFillMode
gDefineHatchStyle
gDefineLineStyle

gFillPolygonSet

Syntax

C/C++:
void gFillPolygonSet2D(int fill, int line, int inv, int npol, GPOLYGON *polygons2);
void gFillPolygonSet3D(int fill, int line, int inv, int npol, GPOLYGON3 *polygons3);

F90:
subroutine gFillPolygonSet2D(fill, line, inv, npol, polygons2)
subroutine gFillPolygonSet3D(fill, line, inv, npol, polygons3)

integer, intent(in) :: fill,line,inv,npol
type (GPOLYGON),intent(in) :: polygons2(*)
type (GPOLYGON3),intent(in) :: polygons3(*)

Arguments

fill
Fill style index

= GHOLLOW,	Draw boundary only
= GSOLID,	Solid fill
= 1 - 256,	Fill style index (hardware) or hatch style index (software)
> 256,	Fill style index (hardware) or solid fill (software)

line
Line style index

= GCURRENT,	Current line style
= 1 - 256,	Line style index
> 256,	Current line style

inv
Inverse fill flag

= GAREA,	Fill polygon
= GINVERSE,	Fill outside polygon (inverse fill)

npol
Number of polygons in polygon set

polygons2,polygons3
Array of 2D or 3D polygon structures to be filled

Description

The routines gFillPolygonSet2D() and gFillPolygonSet3D() fill a set of polygons according to the specified fill and line styles. Each polygon structure consists of a number of vertices and a pointer to an array of 2D or 3D points. Each polygon is complete within itself and will automatically be closed if not defined as such. Coordinates are absolute and have no relation to the current drawing position.

These routines can handle up to 2048 points and if the total number of points in the polygon set exceeds this, an error message is generated and no output is done.

fill defines the fill style. The constant GHOLLOW specifies that only the boundary is to be drawn and GSOLID defines solid fill. Values between 1 and 256 point to the entries in the hatch style table (see gDefineHatchStyle()) and thus specifies the hatch style for the fill. Whether an area is filled by software or hardware is determined by gSetFillMode(). If hardware fill is enabled, **fill** may be interpreted in a device dependent way. By default, hardware fill is attempted.

If it is not possible to fill using the device's hardware then GINO defaults to the hatch style specified by **fill**. If **fill** is out of range for software filling, a solid fill is used by default.

line defines the line style used for the boundary or fill style. A value of **line** between 1 and 256 points to an entry in the line style table (see gDefineLineStyle()) and specifies additional attributes for the fill, usually a colour for hardware fill and a line type and colour for hatching. The current line style is left unchanged.

A value for **line** which is zero or greater than 256 specifies the current line style. A value for **line** less that zero or **fill** less than -1 generates a warning message and its positive value is used.

inv defines whether the inside of the polygon is filled, or if **inv** = GINVERSE, the inverse area is filled up to the current clipping limits.

The current position is restored to that prior to calling either routine.

See Also

Page 170, 288
gDrawPolylineSet2D
gDrawPolylineSet3D

gFillRect

Syntax

C/C++: **void gFillRect**(int fill, int line, GLIMIT *limit);

F90: **subroutine gFillRect**(fill, line, limit)

```
integer, intent(in) :: fill,line
type (GLIMIT), intent(in) :: limit
```

Arguments

fill
Fill style index

= GHOLLOW,	Draw boundary only
= GSOLID,	Solid fill
= 1 - 256,	Fill style index (hardware) or hatch style index (software)
> 256,	Fill style index (hardware) or solid fill (software)

line
Line style index

= GCURRENT,	Current line style

= 1 - 256,	Line style index
> 256,	Current line style

limit
Coordinates of the rectangle to be filled

Description

The routine gFillRect() fills a rectangle whose corners are transformed according to the current GINO transformation and clipped to the current clipping limits - the result of which may not be a rectangle.

fill defines the fill style. The constant GHOLLOW specifies that only the boundary is to be drawn and GSOLID defines solid fill. Values between 1 and 256 point to the entries in the hatch style table (see gDefineHatchStyle()) and thus specifies the hatch style for the fill. Whether an area is filled by software or hardware is determined by gSetFillMode(). If hardware fill is enabled, **fill** may be interpreted in a device dependent way. By default, hardware fill is attempted. If it is not possible to fill using the device's hardware then GINO defaults to the hatch style specified by **fill**. If **fill** is out of range for software filling, a solid fill is used by default.

line defines the line style used for the boundary or fill style. A value of **line** between 1 and 256 points to an entry in the line style table (see gDefineLineStyle()) and specifies additional attributes for the fill, usually a colour for hardware fill and a line type and colour for hatching. The current line style is left unchanged.

A value for **line** which is zero or greater than 256 specifies the current line style. A value for **line** less that zero or **fill** less than -1 generates a warning message and its positive value is used.

The current position is restored to that prior to calling gFillRect().

See Also

Page 165
gSetFillMode
gDefineHatchStyle
gDefineLineStyle

gFillSelectedPolygons

Syntax

C/C++: **void gFillSelectedPolygons**(int fill, int line, int inv);

F90: **subroutine gFillSelectedPolygons**(fill, line, inv)
integer, intent(in) :: fill,line,inv

Arguments

fill
Fill style

= GHOLLOW,	Draw boundary
= GSOLID,	Solid fill
= 1 - 256,	Fill style index (hardware fill) or hatch style index (software fill)
> 256,	Fill style index (hardware fill) or solid fill (software fill)

line
Line style

= GCURRENT,	Current line style
= 1 - 256,	Line style index
> 256,	Current line style

inv
Inverse fill flag

= GAREA,	Fill polygons
= GINVERSE,	Fill all but polygons (inverse fill)

Description

The routine gFillSelectedPolygons() is used to fill polygons created using the routines gStartPolygon()/gEndPolygon() and selected using gSelectPolygons(). These polygons may consist of any combination of lines, moves, arcs and curves. As polygons may intersect there is a need for a more precise definition of the areas that should be filled: an area is filled if it is separated from the background area by an odd number of polygon boundaries. Inverse fill calls for an even number of separating boundaries and by definition always fills the background area (the continuous area that surrounds all polygons).

Although polygons may be defined anywhere in picture space, the fill actually generated will be clipped to the current clipping limits.

fill defines the fill style. The constant GHOLLOW specifies that only the boundary is to be drawn and GSOLID defines solid fill. Values between 1 and 256 point to the entries in the hatch style table (see gDefineHatchStyle()) and thus specifies the hatch style for the fill. Whether an area is filled by software or hardware is determined by gSetFillMode(). If hardware fill is enabled, **fill** may be interpreted in a device dependent way. By default, hardware fill is attempted. If it is not possible to fill using the device's hardware then GINO defaults to the hatch style specified by **fill**. If **fill** is out of range for software filling, a solid fill is used by default.

line defines the line style used for the boundary or fill style. A value of **line** between 1 and 256 points to an entry in the line style table (see gDefineLineStyle()) and specifies additional attributes for the fill, usually a colour for hardware fill and a line type and colour for hatching. The current line style is left unchanged.

A value for **line** which is zero or greater than 256 specifies the current line style. A value for **line** less that zero or **fill** less than -1 generates a warning message and its positive value is used.

The current position is restored to that prior to calling gFillSelectedPolygons().

See Also

Page 257
gStartPolygon
gEndPolygon
gSelectPolygons
gSetFillMode
gDefineHatchStyle
gDefineLineStyle

gFitCharStr

Syntax

C/C++:	**void gFitCharStr**(char string[], float x1, float y1, float x2, float y2, int fit);

F90:	**subroutine gFitCharStr**(string, x1, y1, x2, y2, fit) character*(*), intent(in) :: string real, intent(in) :: x1,y1,x2,y2 integer, intent(in) :: fit

Arguments

string
Character string

x1,y1
The absolute coordinates of the start point of fitting line

x2,y2
The absolute coordinates of the end point of fitting line

fit
Fitting type

= GB2P,	Adjust string angle only (default)
= GSIZE,	Adjust character size and angle

Description

The routine gFitCharStr() outputs a character string so that it is aligned along an arbitrary line joining the two specified points.

Strings should be terminated by *. otherwise the number of characters is limited to the total length of **string** up to a maximum of 256 characters. For example, if string is declared as char [10] then the string will always be terminated after 10 characters, if no intervening *. is encountered.

The characters are drawn subject to the font representation, font weight, italics and underline settings. The string is justified according to the current setting of gSetStrJustify(). Left-justified strings start at the position **x1,y1**, centre-justified strings are positioned mid-way between the specified coordinates, and right-justified strings end at the position **x2,y2**.

If **fit** = GB2P the string angle is adjusted so that the string lies along the arbitrary fitting line but without attempting to fit it to the length of the line. When **fit** = GSIZE the character size is adjusted as well as the string angle so that the string fits between the two points. If the string contains more than one line (separated by the *N escape sequence) the character size is based on the longest line in the string.

Any adjustments in the character angle or size are reset to their current values after the string is output.

The character string may contain any of the GINO escape sequences described under gDisplayStr().

If **fit** is out of range a warning message is output and **fit** is set to GB2P.

See Also Page 158
gSetStrJustify
gDisplayStr

gFlushGraphics

Syntax

C/C++:	**void gFlushGraphics**(void);

F90:	**subroutine gFlushGraphics**

Arguments None

Description In order to reduce the communications traffic between a graphics program and a graphics device, GINO and some device drivers store information in a buffer, ready to send it to the device in bursts. This may have the effect that the display is not up-to-date with the sequence of GINO calls within the program.

The routine gFlushGraphics() ensures all graphics information held in all internal buffers within GINO and the current graphics device is sent to the display.

See Also Page 49

gFopen

Syntax

C/C++:	**GFILE *gFopen**(char name[], char cmode[]);

F90:	**integer function gFopen**(name, fmode) character*(*), intent(in) :: name integer, intent(in) :: fmode

Arguments

name
File name

cmode
File open mode for C/C++

= r,	Open for reading only
= w,	Open for writing
= r+,	Open for update (reading and writing)
= w+,	Open for writing

fmode
File open mode for F90

= GREADONLY,	Open for reading only
= GWRITE,	Open for writing

= GUPDATE,	Open for update (reading and writing)
= GSCRATCH,	Open scratch file (removed at gFclose())

Description

The routine gFopen() is used to open a GINO file unit for use with routines associated with file I/O operations. If successful the routine returns a pointer of type GFILE in the C/C++ version or an integer file unit in the F90 version which is required by these routines.

The file is opened with the name passed in the character string **name** and with one of the above specified modes.

If an unspecified mode is given, or the file cannot be opened the routine returns the NULL pointer (from <stdio.h>) for the C/C++ version or 0 (zero) for the F90 version.

The file can be closed with the routine gFclose().

See Also

Page 27, 28
gFclose

gGenerateView

Syntax

C/C++:	**void gGenerateView**(void);

F90:	**subroutine gGenerateView**

Arguments

None

Description

The routine gGenerateView() generates a new modelling transformation by combining the data specified by the viewing routines with the current modelling transformation. When in space mode (the default) the current modelling transformation is post-multiplied by the viewing parameters, while in picture mode (see gSetTransformMode()) the current modelling transformation is pre-multiplied by the viewing parameters. Users are referred to the routine gUpdateView() which sets the same viewing parameters on the current output without modifying the modelling transformation matrix.

The default view is a parallel view pointing along the negative z-axis with the view centre at the origin. It can be redefined by calls to gDefinePerspView(), gDefineSphericalView() or gDefineParallelView() and modified by calls to gSetViewEyeDistance(), gMoveViewCentre(), gViewShift() or gViewTurn(). The default view can be restored by calling gInitView(). If the current view has perspective defined, it may also be modified by calls to gSetViewPlaneDistance() or gViewRotate().

Unless gPosViewCentre() is called the view centre is projected onto the centre of the current window limits. The window limits are set to the viewport limits unless a user-defined window is currently specified (see gSetWindow2D(), gSetWindow3D() or gSetWindowMode()). If gSetViewUpDirection() is not called, gGenerateView() attempts to project the 3-D y-axis parallel to the screen y-axis, or if the 3-D y-axis is parallel to the view direction, then the 3-D x-axis is projected parallel to the screen x-axis.

See Also

Page 399
gMoveViewCentre
gInitView
gDefineParallelView
gSetViewEyeDistance
gDefinePerspView
gPosViewCentre
gViewRotate
gViewShift
gDefineSphericalView
gViewTurn
gSetViewUpDirection
gSetViewPlaneDistance
gSetWindow
gSetWindowMode
gUpdateView

gGetCGMElement

Syntax

C/C++	**void gGetCGMElement**(int element);

F90:	**subroutine gGetCGMElement**(element) integer, intent(out) :: element

Arguments

element
CGM element identifier

Description

When interpreting a CGM metafile element by element, gGetCGMElement() is used to find the identifier of the next element in the file. This element may then be interpreted or skipped using the routines gInterpretCGMElement() and gSkipCGMElement() respectively.

The CGM file must be opened using the routine gOpenCGMFile().

See Also

Page 70
gOpenCGMFile
gInterpretCGMElement
gSkipCGMElement

gGetCursorEvent

Syntax

C/C++:	**void gGetCursorEvent**(int *key, GPOINT *point);

F90:	**subroutine gGetCursorEvent**(key, point) integer, intent(out) :: key type (GPOINT), intent(out) :: point

Arguments

key
ASCII key code

= 0, no cursor input

point
Cursor position in picture coordinates

Description

The routine gGetCursorEvent() turns on the graphics cursor and waits until a key is pressed (or some other suitable trigger). If the call to gGetCursorEvent() is successful, the cursor or pointer position is returned in **point** and the integer ASCII code of the key pressed is returned in **key** (Refer to the specification of gGetEventRecord() for the possible return values of **key**). If the call to gGetCursorEvent() is unsuccessful, **key** and **point** are returned set to zero.

The position of the cursor or pointer when it is turned on can be specified with a call to gSetCursorPos() or gSetMousePos(). The default for this is the centre of device limits. If the call to gGetCursorEvent() is successful, the cursor start position is set to **point**. If the device does not support cursor input, an error message is output.

The cursor or pointer shape displayed when gGetCursorEvent() is called can be set using the routine gSetCursorType(). In addition some devices offer the display of rubber bands, boxes and circles while the cursor is being moved. These options can be set with the routine gSetCursorAction().

Further information on the implement providing the cursor position and key can be obtained by calling gGetEventRecord().

Appendix B should be consulted to see whether the device supports cursor input, positioning, shapes or actions.

See Also

Page 241
gSetCursorAction
gSetCursorPos
gSetCursorType
gGetEventRecord
Appendix B

gGetDirList

Syntax

C/C++:
```
int gGetDirList(int *n, char *names[], int flen);
```

F90:
```
integer function gGetDirList(n, names)

integer, intent(inout) :: n
character*(*), intent(out) :: names(*)
```

Arguments

n
Input - Size of **names** array Output - Number of entries in **names**

names
List of file names

flen
Maximum length of filename including terminating NULL (C/C++ only)

Description

The system utility gGetDirList() returns a simple list of file names in the current directory. The input value of the argument **n** specifies the maximum number of files to be returned (ie. indicating the size of the **names** array), but on return, **n** will be set to the actual number of files returned which may be less than or equal to the input value.

The function returns a non zero value if the operation was not successful for any reason and zero if successful.

In the C/C++ binding, space must be allocated for the required number of file names prior to calling this routine. This is achieved by declaring an array of length **n** of character pointers (the address of which is passed to this function) and allocating space for each file name using malloc, the address of which is placed in the character array. The space allocated for each file name should be equal to **flen** * sizeof(char). In both bindings, if the individual file names are larger than the width of the character array, the names will be truncated.

A more comprehensive utility is provided by gGetFullDirList() if more information is required about files and their attributes.

See Also

Page 461
gGetFullDirList

gGetDrawing

Syntax

C/C++:	**void gGetDrawing**(GFILE *fp, int nseg, int mode, int paper);

F90:	**subroutine gGetDrawing**(unit, nseg, mode, paper) integer, intent(in) :: unit,nseg,mode,paper

Arguments

fp
GINO-C file pointer

unit
Fortran 90 file unit

nseg
Picture segment number

> 0,	Read specified picture segment
= GALL,	Read all picture segments

mode
Relationship between the size of an object in the metafile and its size on a particular device:

= GABSOLUTE,	The absolute size of an object is the same as that specified in the initial metafile generating program regardless of the current drawing units
= GMAPPED,	The positive quadrant of the metafile drawing area is mapped into the current drawing area (window or viewport limits)

= GTRANSFORMED,	The coordinates of an object on interpretation are the same as those when the file was created, but subject to the current GINO transformation
= GWHOLE,	All four quadrants (positive and negative) of the metafile drawing area are mapped into the current drawing area (window or device limits)

paper
The paper size to be used:

= GPROGRAM,	The paper size defined in the processing program is used for each drawing interpreted
= GMETAFILE,	The paper size given at the start of each drawing in the metafile is used

Description

The routine gGetDrawing() interprets a metafile produced by the gSavdra() generator. The output from gGetDrawing() will correspond exactly with the output from the program that created the file. gGetDrawing() restores all the settings of the line and character attributes that were current when each picture segment was opened.

gGetDrawing() can also interpret files produced by gSavpic(). However, gSavpic() does not record the line and character attributes when each segment is opened, so this information cannot be restored by gGetDrawing(). A message is output by gGetDrawing() to warn the user about this.

The metafile should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

If **mode** is GMAPPED or GWHOLE, the current transformation is ignored. If **mode** is out of range, GABSOLUTE is assumed.

See Also

Page 65
gFopen
gGetPicture
Appendix B

gGetEventRecord

Syntax

C/C++:
```
void gGetEventRecord(int intype, GEVEREC *everec);
```

F90:
```
subroutine gGetEventRecord(intype, everec)

integer, intent(in) :: intype
type (GEVEREC), intent(out) :: everec
```

Arguments

intype
Event type

= GNULL,	Null event type
= GKEYPRESS,	Key or mouse button press (key)
= GSEGMENT,	Picture segment number (nseg)

= GSEGMENTANDKEY,	Picture segment number and key/mouse button (key,nseg)
= GLOCATOR,	Screen position and key/mouse button press (key,pos)
= GSTRING,	Text string (nargs,iargs)
= GREALS,	String of real values (nargs,args)
= GINTEGERS,	String of integer values (nargs,iargs)
= GMOVEMENT,	Pointer, mouse or tablet movement (pos)
= GKEYRELEASE,	Key or mouse button release (key,pos)
= GRESIZE,	Window resize event (nargs,args)
= GPOINTERLEAVING,	Pointer leaving window (pos)
= GPOINTERENTERING,	Pointer entering window (pos)

everec.key
ASCII key code

= 0,	Undefined

everec.impkey
Identifier of implement on which key was pressed

= -1,	Undefined

everec.impdat
Identifier of implement from which data was supplied

= -1,	Undefined

everec.nseg
Picture segment number

= -1,	Undefined

everec.pos
Screen position in picture coordinates

= (0.0,0.0),	Undefined

everec.nargs
Number of data values in args or iargs

= 0,	No data

everec.args[80]
A real array of data

everec.iargs[80]
An integer array of data

Description

The routine gGetEventRecord() will return appropriate event data after a call to gWaitForEvent(), gDragSeg() or gGetCursorEvent(). **intype** specifies the type of event data that should be returned.

After a call to gWaitForEvent(), **intype** should be set to the value returned by that routine. After a call to gDragSeg() or gGetCursorEvent(), **intype** should be set to GLOCATOR. If **intype** is set to GNULL or is out of range, the event record is set for the null data type, which corresponds to the values 0,-1,-1,-1,0.0,0.0 and 0. If **intype** is out of range, an error message is output.

Each event data type defines some but not all of the return parameters as indicated above. Any undefined parameters will be set to the value corresponding to the null data type.

everec.key returns the key/button pressed or released corresponding to the following table of keyboard/mouse keys:

key	Keyboard key
0-127	Standard 7-bit ASCII character
76,77,82	Left/Middle/Right mouse button
513-532	Function keys 1-20 (i.e. function key n +512)
533-536	Left/Right/Up/Down arrow key
537,538	Page Up/Down or Prior/Next
539	Insert
540,541	Home/End
542	Shift TAB
544	Numerical Key Pad /
545	Numerical Key Pad *
546	Numerical Key Pad -
547	Numerical Key Pad +
548	Numerical Key Pad 'Enter'
549	Numerical Key Pad .
550-559	Numerical Key Pad 0-9
560	Print Screen
561	Pause
562	Select
563	Execute
564	Help

The following keyboard combinations are also returned in **everec.key**, but where the required combination is not catered for it is suggested that the function gEnqKeyState() is used.

key	Keyboard Key
n+64	Shift + special key > 511 (i.e. Shift F1 = 577)
n-64	Ctrl + special key > 511 (i.e. Ctrl F1 = 449)
n+1024	Alt + any key (i.e. Alt P = 1104, Alt F1 - 1537)

If no key was typed, **everec.key**= 0.

everec.impkey and **everec.impdat** are set to the implement types of the devices that supplied the key and data associated with the event. If no key was typed, or no data was supplied, the identifier(s) is returned as -1.

Implement types are as follows:-

```
0      Screen
100    Keyboard
200    Function Box
300    Light Pen
400    Joystick/arrow keys/thumb wheel/mouse
500    Tablet/Digitizer
600    Valuator
```

everec.nseg is set to the picture segment number concerned in the event. If no picture segment is relevant, **everec.nseg** is set to -1. **everec.pos** is the picture coordinates of the position being returned and are set to -1.E-6 if no position is being returned.

All further information is stored in array **everec.args** or **everec.iargs**, the variable **everec.nargs** giving the number of elements relevant to this event. If **everec.nargs** is set to zero, there are no data values returned in **everec.nargs** or **everec.iargs**, and no attempt should be made to access any element of these data arrays. For a text string, **everec.iargs** contains the ASCII codes of the string entered and for event GRESIZE, the first two elements of **everec.args** contain the width and height of the resized window in picture coordinates.

See Also

Page 451
gDragSeg
gEnqKeyState
gWaitForEvent

gGetFullDirList

Syntax

C/C++:	**int gGetFullDirList**(char pattern[], int *n, char *names[], int types[], int dates[], int sizes[], int flen);

F90:	**integer function gGetFullDirList**(pattern, n, names, types, dates, sizes) character*(*), intent(in) :: pattern integer, intent(inout) :: n character*(*), intent(out) :: names(*) integer, intent(out) :: types(*),dates(*),sizes(*)

Arguments

pattern
File search pattern

n
Input - Size of attribute arrays (No. of files required) Output - Number of entries in attribute arrays (No. of files returned)

names
List of file names

types
List of file types

dates
List of file creation/modification dates (packed)

sizes
List of file sizes (in bytes)

flen
Maximum length of filename including terminating NULL (C/C++ only)

Description

The function gGetFullDirList() returns the names of the files and their attributes that match the file search pattern passed in the character string **pattern**. **pattern** may contain any legal search pattern containing directory name and/or wild characters that are permitted in the implementation being used. If **pattern** is a blank string, all files in the current directory are treated as matching the pattern. The input value of the argument **n** specifies the maximum number of files to be returned (ie. indicating the size of the return arrays **names**, **types**, **dates** and **sizes**), but on return, **n** will be set to the actual number of files returned which may be less than or equal to the input value.

In the C/C++ binding, space must be allocated for the required number of file names prior to calling this function. This is achieved by declaring an array of length **n** of character pointers (the address of which is passed to this function) and allocating space for each file name using malloc, the address of which is placed in the character array. The space allocated for each file name should be equal to **flen** * sizeof(char). In both bindings, if the individual file names are larger than the width of the character array, the names will be truncated.

The list of file names is returned in **names**, each being a simple name without preceding directory name. Their associated file types and sizes are returned in the corresponding elements of the **types**, **dates** and **sizes** arrays. Each entry of the **types** array contains a bit pattern representing the following information about the file:

Bit	Information
1	Read-only
2	Hidden
3	System
4	Archive
5	Directory

Thus if the **types** element =0 , the corresponding file is a normal file, but if the element contains 17, the file is a read-only directory etc.

Each element in the **dates** array contains an integer value representing the date and/or time the corresponding file was last modified or created. The value is in an implementation dependent format and should only be used for sorting purposes without unpacking. The system utility routine gReturnDirDate() can be used to unpack the date and time information into its separate components.

The routine returns a non zero value if the operation was not successful for any reason and zero if successful.

See Also

Page 461
gReturnDirDate
gGetDirList

gGetImageFile

Syntax

C/C++:	**void gGetImageFile**(int type, char file[], int coldef, int coloff, int collim, int *ixgrid, int *iygrid, int psiz, int pix[]);

F90:	**subroutine gGetImageFile**(type, file, coldef, coloff, collim, ixgrid, iygrid, psiz, pix) integer, intent(in) :: type character*(*), intent(in) :: file integer, intent(in) :: coldef,coloff,collim,psiz integer, intent(out) :: ixgrid,iygrid,pix(*)

Arguments

type
Image type

= GBMPFILE,	Uncompressed Windows Bitmap file (BMP)
= GXWDFILE,	X Windows Dump format (XWD)
= GICOFILE,	Windows Icon file (ICO)
= GJPEGFILE,	JPEG image format (JPG)
= GPNGFILE,	Portable Network Graphics file (PNG)

file
Image file name

coldef
Colour definition flag

= 0,	Skip colour definition table - use image data only
= 1,	Read and interpret image file making appropriate GINO colour table changes
= 2	Map image to (nearest) GINO colour indices

coloff
Colour offset

collim
Colour index limit

= 0,	Current device maximum colour index (see gSetColourInfo())

ixgrid
Image width in pixels

iygrid
Image height in pixels

psiz
Size of image array **pix**

pix
Image array

Description

The routine gGetImageFile() will read an image file of the specified type into a pixel array for subsequent GINO processing or output using the image routines. The user must set the size of the pixel array **pix** in **psiz** which should be large enough to hold any image read in by this routine. The size of the image read is returned in the variables **ixgrid** and **iygrid**.

The image file may contain 1,2,4 or 8bit colour indices (optionally preceded with a list of colour definitions) or consist of 24bit RGB triplets. The data is interpreted according to the setting of the arguments **coldef**, **coloff** and **collim**, taking into account the colour mode of the currently nominated device.

For **coldef**=0, any colour index definition data (in an indexed image file) is ignored and the image data is read into the image array as it is stored (adding the specified offset in **coloff**). This mode is not permitted when reading 24bit images on colour index devices (monochrome or static or dynamic) as the data would represent an illegal colour index.

For **coldef**=1, the image integrity is maintained as far as is possible taking into account the colour facilities of the current device. When reading an image on an indexed device, the GINO colour table is modified, within the range of **coloff** and **collim**, to match the colours required in the image file and the appropriate index is placed in the image array. On true colour devices, a 24bit RGB triplet is placed in the image array for each pixel read.

For **coldef**=2, the image colours, whether indexed or 24bit, are mapped onto the nearest colours in the existing GINO colour table and the resulting index is placed in the image array. No modifications are made to the GINO colour table. This mode is useful when using an indexed device, but can seriously limit the range of colours in a 24bit image.

Image file size and attributes can be enquired using the routine gEnqImageFile() before using gGetImageFile().

See Also

Page 74, 346
gDrawCellArray
gDrawPixelArea
gEnqImageFile

gGetPicture

Syntax

C/C++:	**void gGetPicture**(GFILE *fp, int nseg);

F90:	**subroutine gGetPicture**(unit, nseg) integer, intent(in) :: unit,nseg

Arguments

fp
GINO-C file pointer

unit
Fortran 90 file unit

nseg
Picture segment number

> 0,	Read specified picture segment
= GALL,	Read all picture segments

Description

The routine gGetPicture() interprets a metafile produced by the gSavdra() or gSavpic() generator. Only the information contained within each picture segment is output. gGetPicture() ignores any information that records the settings of the line and character attributes that were current when each picture segment was opened.

The metafile should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

A call to gGetPicture() does not affect any of the current settings (line attributes, character attributes, transformation, etc).

See Also

Page 67
gFopen
gGetDrawing
Appendix B

gGetPixel

Syntax

C/C++:	**void gGetPixel**(int ix, int iy, int pix); **void gGetPixelArea**(int ix, int iy, int npixx, int npixy, int isc, int isr, int idx, int idy, int pixbuf[]);

F90:	**subroutine gGetPixel**(ix ,iy, pix) **subroutine gGetPixelArea**(ix, iy, npixx, npixy, isc, isr, idx, idy, pixbuf) integer, intent(in) :: ix,iy,npixx,npixy,isc,isr,idx,idy integer, intent(out) :: pix,pixbuf(*)

Arguments

ix,iy
Pixel position of the top left corner of the pixel array to be read

pix
Pixel information for single pixel

npixx,npixy
Number of X and Y pixel values to be stored in index array

isc,isr
The start column and row of a sub-array

idx,idy
The X and Y dimensions of a sub-array

pixbuf
The array name in which the data is to be stored

Description

The routines gGetPixel() and gGetPixelArea() reads a single pixel or a rectangular pixel area from a devices display. The pixel (area) is read with reference to the anchor position specified by the position (**ix,iy**), noting that the pixel coordinate system has its origin at the top left corner of the device with the first pixel position being referenced as (0,0).

In both routines, the colour information returned may consist of colour indices or 24bit true colour values depending on the colour mode of the currently nominated device and as set by the routine gSetColourInfo().

In the case of gGetPixelArea(), the pixel information is passed through a pointer to an integer array **pixbuf** dimensioned (**npixx**,**npixy**). Where the whole of this array is to be retrieved, the user should set **isx** and **isy** to 1 and **idx** and **idy** to be the same as **npixx** and **npixy**. Alternatively a portion of the array can be replaced with the area on the screen (still at the anchor position **ix**,**iy**) by setting the values of **isx** and **isy** to the offsets from the start of the **pixbuf** and **idx**,**idy** to the dimensions of the sub-array.

The routine gGetPixelArea() will pack the pixel data into the **pixbuf** array according to the specification set by gDefinePixelPacking(). The default is for one pixel value to be placed into one word of **pixbuf**. The pixel rectangle will be clipped to the device limits if these limits are exceeded. The pixel array will also be subject to the current pixel transformation as set by gSetPixelTransform().

See Also

Page 191
gDefinePixelPacking
gSetColourInfo

gGetRand

Syntax

C/C++:	**void gGetRand**(float *rand);

F90:	**subroutine gGetRand**(rand) real, intent(out) :: rand

Arguments

rand
Returned random number in range 0.0 to 1.0

Description

The system utility gGetRand() can be used to return a non repeating sequence of random numbers in the range 0.0 to 1.0.

If an application requires a set sequence of random numbers, a seed may be set using the routine gSetRandSeed(). By default a date/time seed is set at GINO initialization ensuring a different set of random numbers on each run, if gSetRandSeed() is not called.

See Also

Page 466
gSetRandSeed

gGetTransform

Syntax

C/C++:	**void gGetTransform2D**(GMAT2D a2); **void gGetTransform3D**(GMAT3D a3);

F90:	**subroutine gGetTransform2D**(a2) **subroutine gGetTransform3D**(a3) real, intent(out) :: a2(6) real, intent(out) :: a3(16)

Arguments

a2,a3
2D or 3D transformation matrix

Description

The routines gGetTransform2D() or gGetTransform3D() are used to store a copy of the current modelling transformation in the argument **a2** or **a3**.

Where gGetTransform2D() is used only the 2-D part of current modelling transformation is stored in the array **a2**.

See Also

Page 377

gGetView

Syntax

C/C++:	**void gGetViewParams**(GMATV vdata): **void gGetViewState**(GVIEWSTATE vstate);

F90:	**subroutine gGetViewParams**(vdata) **subroutine gGetViewState**(vstate) real, intent(out) :: vdata(15) type (GVIEWSTATE), intent(out) :: vstate

Arguments

vdata
Viewing data parameters

vstate.mode
View mode

= 0,	No view defined
= 1,	Perspective view defined (gDefineSphericalView() or gDefinePerspView())
= 2,	Parallel view defined (gDefineParallelView())

vstate.cflag
View centre flag

= 0,	No view centre defined

= 1, Default view centre defined
= 2, User defined view centre (gPosViewCentre())

vstate.upflag
View up direction flag

= 0, Default view up vector (0.0,1.0,0.0)
= 1, User defined view up vector (gSetViewUpDirection())

vstate.dir
View direction vector

vstate.centre
View centre

vstate.dist
Perspective viewing distance

vstate.shift
View shift

vstate.upvec
View up direction vector

Description

The routines gGetViewParams() and gGetViewState() return all the viewing parameters in either the array **vdata** or the structure **vstate**. The viewing parameters can be reset to the values stored in either structure by calling the appropriate setting routine gSetViewParams() or gSetViewState().

The current eye position can be calculated using the following formulae:

xeye = vstate.centre.x - vstate.dist * vstate.dir.x
yeye = vstate.centre.y - vstate.dist * vstate.dir.y
zeye = vstate.centre.z - vstate.dist * vstate.dir.z

Note that for parallel viws (mode = 2) vstate.dist will equal 0.0, so any arbitary value for dist can be substituted to calculate a normal eye position.

See Also

Page 416
gSetViewParams
gSetViewState

gInitView

Syntax

C/C++:	**void gInitView**(void);
F90:	**subroutine gInitView**

Arguments

None

Description

The routine gInitView() resets the viewing parameters to their initial default settings and initializes the viewing transformation matrix. It does not affect the modelling transformation matrix. The default settings are as follows:

Parallel view

View direction = (0.0,0.0,-1.0)

View centre = (0.0,0.0,0.0)

View centre position = centre of current window limits

View UP direction = (0.0,1.0,0.0)

The effect of a call to gInitView() can also be achieved by calling gSetTransform(GRESET) but this has the additional effect of initializing the modelling transformation matrix as well.

See Also

Page 399
gSetTransform

gInsertSegRef

Syntax

C/C++:	**void gInsertSegRef**(int nseg);

F90:	**subroutine gInsertSegRef**(nseg) integer, intent(in) :: nseg

Arguments

nseg
Picture segment number

Description

The routine gInsertSegRef() inserts a reference to segment **nseg** into the currently opened segment.

A reference may be made to a segment which does not yet exist, in which case the reference is ignored until the referenced segments are created and an operation is carried out on the segment, e.g. gMoveSegTo2D(), gMoveSegBy2D(), gDrawSeg(), gSetSegVis().

A referenced segment will inherit line and character attributes from its parent until specifically set within that segment. All current attributes are, however, saved at the beginning of a reference and restored once the referenced segment has been drawn.

A segment cannot reference itself but may reference another segment up to a depth of 10 references. GINO also checks for recursive references when a segment structure is traversed as long as it is holding a copy of the display file.

gInsertSegRef() may not operate on some devices when a hardware display file is available if this function is not provided. A call to gSetSegMode(GSOFTWARE) may therefore be required to force software emulation of hierarchical segment structures.

See Also

Page 435
gMoveSegBy2D
gDrawSeg
gMoveSegTo2D
gSetSegVis
gSetSegMode

gInsertSegTag

Syntax

C/C++:	**void gInsertSegTag**(int tag);

F90:	**subroutine gInsertSegTag**(tag) integer, intent(in) :: tag

Arguments

tag
Identifier

Description

The routine gInsertSegTag() is called to set a user supplied identifier in a picture segment to facilitate later editing.

It can be called anywhere within a segment but can only usefully be used immediately prior to a routine which changes or sets the modelling transformation matrix. In this way the matrix can be replaced using the routines gEditSeg2D() or gEditSeg3D().

See Also

Page 436
gEditSeg2D
gEditSeg3D

gInterpolateData

Syntax

C/C++:	**int gInterpolateData2D**(int nopt, float ptint, int npts, GPOINT *points2, int nptout, float *ptout1); **int gInterpolateData3D**(int nopt, float ptint, int npts, GPOINT3 *points3, int nptout, float *ptout1, float *ptout2);

F90:	**integer function gInterpolateData2D**(nopt, ptint, npts, points2, nptout, ptout1) **integer function gInterpolateData3D**(nopt, ptint, npts, points3, nptout, ptout1, ptout2) integer, intent(in) :: nopt,npts,nptout real, intent(in) :: ptint type (GPOINT),intent(in) :: points2*) type (GPOINT3),intent(in) :: points3(*) real, intent(out) :: ptout1,ptout2

Arguments

nopt
Interpolation value data type

= GXDATA,	Date value to to interpreted as X data
= GYDATA,	Date value to to interpreted as Y data
= GZDATA,	Date value to to interpreted as Z data

ptint
Data value to be interpolated

npts
Number of data points in either points2 or points3 arrays

points2,points3
Array of 2D or 3D data points

nptout
Size of output arrays ptout1 (and ptout2)

ptout1, ptout2
Intersection values from interpolation

Description

The functions gInterpolateData2D() and gInterpolateData3D() can be used to interpolate a single value against either a 2D or 3D array of data points. The input data points may be supplied from user supplied data or from GINO's internal points storage facilities enabling interpolation of previously drawn lines, arcs or curves.

Interpolation may be carried out in X, Y or Z (where the 3D function is used) according to the setting of **nopt** and value passed in **ptint**. The data on which the interpolation takes place is passed in the arrays **points2** or **points3** with **npts** specifying the number of points supplied.

As there are no restrictions on the form of the input data, the resulting interpolation may supply zero, one or more intersections and this data is returned in the arrays **ptout1** and **ptout2** (where the 3D function is used). Users should set the size of these arrays using the input argument **nptout**, with the actual number of intersections being returned by the function itself. No more than **nptout** intersections will be returned however, even though there may be more depending on the supplied data.

Interpolation is carried out using linear interpolation.

See Also

Page 107
Page 294
gReturnInternalPoints2D
gReturnInternalPoints3D

gInterpretCGMElement

Syntax

C/C++:	**void gInterpretCGMElement**(int element);

F90:	**subroutine gInterpretCGMElement**(element) integer, intent(in) :: element

Arguments

element
CGM element identifier

Description

When interpreting a CGM metafile element by element and having obtained the next element identifier using gGetCGMElement(), this may be interpreted with this routine.

The element may produce graphical output or change the state of the currently nominated device depending on the element. The CGM file must be opened using the routine gOpenCGMFile().

See Also

Page 70
gOpenCGMFile
gGetCGMElement

gMarkSeg

Syntax

C/C++:	**void gMarkSeg**(int nseg, int mark);

F90:	**subroutine gMarkSeg**(nseg, mark) integer, intent(in) :: nseg,mark

Arguments

nseg
Picture segment or segment group number

> 0,	Change 'marked' status of picture segment specified by **nseg**
= GALL,	Change 'marked' status of all picture segments
< -1,	Change 'marked' status of all segments except those specified by **nseg**

mark
'Marked' status

= GUNMARK,	Stop marking (default)
= GMARK,	Mark the segment

Description

The routine gMarkSeg() controls the highlighting of the specified picture segment.

When using hardware segments, highlighting or marking may be achieved by a brightening up or flashing of a picture segment. When using software emulation, marking is achieved by redrawing the selected segment or group with the 'marking colour' index as set by gSetSegMarkColour(). When switching marking off, the segment or group is redrawn in its defined colours.

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(GHARDWARE), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also

Page 431
gSetSegMarkColour

gModifyTransform

Syntax

C/C++:	**void gModifyTransform2D**(GMAT2D a2); **void gModifyTransform3D**(GMAT3D a3);

F90:	**subroutine gModifyTransform2D**(a2) **subroutine gModifyTransform3D**(a3) real, intent(in) :: a2(6),a3(16)

Arguments

a2,a3
2D or 3D transformation matrix

Description

The routine gModifyTransform2D() or gModifyTransform3D() modifies the current modelling transformation by that passed in the specified matrix according to the current transformation mode.

The GINO modelling transformation is multiplied by the 3x2 matrix **a**. When in space mode (the default) the current transformation is post-multiplied by **a**, while in picture mode (see gSetTransformMode()) the current transformation is pre-multiplied by **a**.

If transforming is not on it is switched on and the modelling transformation is set to the unit matrix, prior to multiplication by **a**.

See Also

Page 377
gSetTransformMode

gModifyView

Syntax

C/C++:	**void gModifyView**(GMAT3D a3);

F90:	**subroutine gModifyView**(a3) real, intent(in) :: a3(16)

Arguments

a3
3D transformation matrix

Description

The routine gModifyView() modifies the current viewing matrix by the specified 4x4 transformation.matrix. The viewing matrix is that constructed from the viewing parameters after the routine gUpdateView() has been called, therefore this routine should be called after gUpdateView() to have the correct effect.

This routine is used for the construction of shadows in conjunction with the routine gCreatePlanarShadowMatrx().

See Also

Page 342, 417
gCreatePlanarShadowMatrix
gUpdateView

gMove

Syntax

C/C++:	**void gMoveBy2D**(float dx, float dy); **void gMoveBy3D**(float dx, float dy, float dz); **void gMoveTo2D**(float x, float y); **void gMoveTo3D**(float x, float y, float z);

F90:	**subroutine gMoveBy2D**(dx, dy) **subroutine gMoveBy3D**(dx, dy, dz) **subroutine gMoveTo2D**(x, y) **subroutine gMoveTo3D**(x, y, z) real, intent(in) :: dx,dy,dz real, intent(in) :: x,y,z

Arguments

dx,dy,dz
Coordinate increments (in current units) from the current drawing position to the required position

x,y,z
Absolute coordinate of the new current drawing position (in current units)

Description

The gMove set of routines moves the current drawing position to the specified end point in terms of a relative or an absolute coordinate. The 2D routines do not alter the current Z position which by default is set at 0.0.

Moves are not output to the device until the next visible item is drawn.

See Also

Page 80
Page 279

gMoveSeg

Syntax

C/C++:	**void gMoveSegBy2D**(int nseg, float dx, float dy); **void gMoveSegTo2D**(int nseg, float x, float y);

F90:	**subroutine gMoveSegBy2D**(nseg, dx, dy) **subroutine gMoveSegTo2D**(nseg, x, y) integer, intent(in) :: nseg real, intent(in) :: dx,dy real, intent(in) :: x,y

Arguments

nseg
Picture segment or segment group number

> 0, Reposition segment(s) specified by segment **nseg**

= -1,	Reposition all segments
< -1,	Reposition all segments except those specified by **nseg**

dx,dy
Increment by which picture segment is repositioned

Description

The routines gMoveSegBy2D() and gMoveSegTo2D() are used to reposition and redraw the specified picture segment(s). The new position, specified in terms of relative or absolute coordinates, represents the new position of the anchor of the specified segment (or the first segment of the group if **nseg** is negative).

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(GHARDWARE), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also

Page 431
gSetSegMode

gMoveToNextLine

Syntax

C/C++:	**void gMoveToNextLine**(void);
F90:	**subroutine gMoveToNextLine**

Arguments

None

Description

The routine gMoveToNextLine() (or *N escape sequence) moves the current position to the beginning of the next line in a text block. The start position of a text block is defined using gStartTextBlock(). The start of each subsequent line is positioned according to the current inter-line spacing factor which is set by gSetInterlineSpace(). The action of gMoveToNextLine() (or *N) can be likened to sending a carriage-return and line feed where the margin is set by gStartTextBlock().

The complete text block is subject to the current setting of the string angle set by gSetStrAngle(). Thus each line is positioned ‘below’ the preceding line such that the block structure is maintained.

If the transforming of characters is switched on with gSetCharTransformMode(), the movement to the next line is also subject to the current transformation.

If gStartTextBlock() has not been called, a warning message is output and the current pen position is moved by the current inter-line spacing factor as if only a line-feed had been printed.

See Also Page 154
gStartTextBlock
gSetInterlineSpace
gSetStrJustify
gDisplayStr
gSetCharTransformMode

gMoveViewCentre

Syntax

C/C++:	**void gMoveViewCentre**(float dist);

F90:	**subroutine gMoveViewCentre**(dist) real, intent(in) :: dist

Arguments

dist
Distance to be moved along line of sight

Description

The routine gMoveViewCentre() calculates a new position for the view plane by displacing the view centre a distance **dist** in the direction of viewing. If a perspective view is being modified, the eye position is displaced by the same amount, leaving the perspective distance unchanged.

If **dist** is less than zero, the displacement is in the opposite direction to the direction of viewing.

See Also Page 404

gNewDrawing

Syntax

C/C++:	**void gNewDrawing**(void);

F90:	**subroutine gNewDrawing**

Arguments

None

Description

The routine gNewDrawing() clears the drawing area without removing segments from the display file.

This has different effects on different output devices. On displays the whole drawing area is cleared using the most efficient method. On plotters and printers the paper is ejected or wound-on and a new drawing area or sheet is selected when the next drawing is started.

Any segments held in a display file, either hardware or software, are not deleted but flagged as being invisible. These may be made visible using the gSetSegVis() routine as required.

See Also Page 48, 50, 64
gSetSegVis

gOpenAuxDrawingArea

Syntax

C/C++: **void gOpenAuxDrawingArea**(int ident, char title[], int xp, int yp, int width, int height);

F90: **subroutine gOpenAuxDrawingArea**(ident, title, xp, yp, width, height)

```
integer, intent(in) :: ident
character*(*), intent(in) :: title
integer, intent(in) :: xp,yp,width,height
```

Arguments

ident
Auxiliary drawing area identifier (>1)

title
Auxiliary drawing area title or banner

xp,yp
Position of drawing area relative to display origin

width,height
Width and height of drawing area

Description

The routine gOpenAuxDrawingArea() creates and opens a visible or invisible auxiliary drawing area, where a device has the possibility of multiple drawing areas or display windows.

Auxiliary drawing areas are arranged in pairs, with the initial drawing area consisting of a visible area with identifier zero and an invisible area or backing store with identifier 1. Additional areas may be created such that every visible area automatically has an associated invisible area, but invisible areas may be created without an associated visible one. When gOpenAuxDrawingArea() is called with an even numbered identifier (**ident**) a new visible area or window is created using the remaining arguments to define its title, origin and size together with an associated invisible area of the same size with identifier **ident**+1. If gOpenAuxDrawingArea() is called with an odd numbered identifier, a new invisible area is opened of the specified size (**width** * **height**) ignoring the title and origin settings.

If the identifier is outside the range of possible identifiers for the current device, GINO outputs an error message and no action is taken. The maximum number of areas/windows (in pairs) that can be opened can be obtained through the device enquiry routine gEnqDeviceState(). An error is also generated if a request is made to open a display area that is already open or a request is made to open a visible area where an invisible area of **ident**+1 is already open. You must close a display area using gCloseAuxDrawingArea() before re-opening another one with the same identifier. Some devices may only provide invisible drawing areas (ie. additional screen memory) each of which will have an odd numbered identifier. This restriction is identified by the value of **device.maxaux** returned by gEnqDeviceState() as being negative.

A visible display area cannot be opened which is larger than the maximum drawing area for the current device. Display resources may prevent the opening of a valid drawing area due to lack of memory or network access.

See Also Page 49
gCloseAuxDrawingArea
gEnqDeviceState

gOpenCGMFile

Syntax

C/C++:	**void gOpenCGMFile**(int code, GFILE *fp, int mode, int errlev);

F90:	**subroutine gOpenCGMFile**(code, unit, mode, errlev) integer, intent(in) :: code,unit,mode,errlev

Arguments

code
CGM encoding type

= GCGMCHAR,	Character encoding
= GCGMBINARY,	Binary encoding

fp
GINO-C file pointer

unit
Fortran 90 file unit

mode
Interpretation mode

= GABSOLUTE ,	Metric metafiles are drawn the same size that they were generated
= GMAPPED,	Abstract and Metric metafiles are scaled to fit the current window limits
= GTRANSFORMED,	Same as GABSOLUTE subject to current transformation

errlev
Error checking level

= GOFF,	No error checking
= GFAST,	Fast error checking
= GFULL,	Full error checking

Description

The routine gOpenCGMFile() is used to open a CGM metafile for interpretation element by element.

The required file should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

The metafile may be either character or binary encoded but the interpreter must be informed of which through the **code** argument.

The routines gGetCGMElement(), gInterpretCGMElement() and gSkipCGMElement() are used to get the next element, interpret it, or skip over it respectively. The file is closed with gCloseCGMFile().

Abstract metafiles are drawn such that one VDC unit = one GINO picture unit in GABSOLUTE and GTRANSFROMED modes.

The interpretation may alter the current state of GINO in that the colour table may be changed within a metafile, but all line and character attributes, window and transformation states are restored at the end of a metafile.

See Also Page 70
gFopen

gOpenGino

Syntax

C/C++:	**void gOpenGino**(void)

F90:	**subroutine gOpenGino**

Arguments

None

Description

Calling gOpenGino() initiates the GINO package. Initialization happens automatically on computer systems that zeroize all variables before running a program. Unfortunately, this is not the case for all systems. A call to routine gOpenGino() prior to any other GINO routine will ensure that GINO is properly initialized in all circumstances on any system.

Routine gOpenGino() must never be called after any other GINO routine except gCloseGino(). For example, a call to routine gOpenGino() occurring after a call to a device nomination routine could leave the device in an abnormal state.

See Also Page 25, 39

gOpenSeg

Syntax

C/C++:	**void gOpenSeg**(int nseg);

F90:	**subroutine gOpenSeg**(nseg) integer, intent(in) :: nseg

Arguments

nseg
Picture segment number, 0 - 32767

> 0,	Open picture segment **nseg**
= 0,	Open (or add to) picture segment 0

Description

The routine gOpenSeg() starts a new picture segment.

It must be executed before a picture segment is started. If the user does not call gOpenSeg(), an internal call to gExtendSeg(0) will be made by GINO.

If gOpenSeg() is called from within a picture segment, this segment is automatically terminated.

On a device with picture segments, **nseg** is the number of the picture segment to be opened. If a picture segment with this number (nseg>0) already exists, the old one is deleted and the new one displayed as it is being created.

Segment handling is enabled if the device has the necessary hardware and the device driver has it implemented, or if software emulation has been switched on using gSetSegMode().

If **nseg** = 0, then picture segment 0 is extended. This is the default and is treated as a dustbin segment by GINO.

See Also

Page 64, 426
gCloseSeg
gExtendSeg
gSetSegMode

gPlaySound

Syntax

C/C++:	**void gPlaySound**(int freq, int time);

F90:	**subroutine gPlaySound**(freq, time) integer, intent(in) :: freq,time

Arguments

freq
Frequency

time
Time in milliseconds

Description

The system utility gPlaySound() attempts to sound a note using hardware system sound resources if available. If a note of the specified frequency and/or time is not possible (e.g. under UNIX and OpenVMS), a simple 'BEEP' may be sounded by the system.

Under Windows implementations of GINO, negative values of **freq** can be used to access the different Windows alert sounds. If available, the defined values of these sounds are found in the <windows.h> or <windows.ins> file. Thus:

gPlaySound(-MB_ICONHAND,0)

or

gPlaySound(-16,0)

will play the System Asterisk entry in the [sounds] section of WIN.INI. Other values include:

-MB_ICONQUESTION -32

-MB_ICONEXCLAMATION -48

-MB_ICONASTERISK -64

See Also Page 466

gPolygonHit

Syntax

C/C++:	**void gPolygonHit**(int *ident, float x, float y, float radius);

F90:	**subroutine gPolygonHit**(ident, x, y, radius) integer, intent(out) :: ident real, intent(in) :: x,y,radius

Arguments

ident
Identifier of polygon whose boundary is nearest to hit centre

= -1, No polygon inside hit area

x,y
Picture coordinates of hit centre

radius
Radius of hit area in current units

Description

The routine gPolygonHit() searches the polygon workspace for polygons that overlap the hit area. The identifier of the polygon whose boundary comes closest to the hit centre is returned in **ident**. If no polygon boundaries extend inside the hit area **ident** is set to -1. The hit area is a circular area of radius **radius** whose centre is at **x,y** in picture coordinates. The absolute value of **radius** determines the hit radius.

If gSelectPolygons() has been called, only those polygons currently selected, will be examined.

See Also

Page 261
gSelectPolygons

gPopTransform

Syntax

C/C++:	**void gPopTransform**(void);

F90:	**subroutine gPopTransform**

Arguments

None

Description

The routine gPopTransform() resets the current modelling transformation to the last transformation saved by a call to gPushTransform().

Up to ten different copies of the current modelling transformation may be stored internally by calls to gPushTransform().

Once a copy has been retrieved no further record of that copy is kept.

See Also Page 376
gPushTransform

gPosViewCentre

Syntax

C/C++:	**void gPosViewCentre**(float xp, float yp);

F90:	**subroutine gPosViewCentre**(xp, yp) real, intent(in) :: xp,yp

Arguments

xp,yp
Point in picture coordinates onto which the view centre is projected

Description

The view transformation, when gGenerateView() or gUpdateView() is called, is set up so that the view centre is projected onto the point (**xp,yp**).

See Also

Page 402
gGenerateView
gUpdateView

gPrintf

Syntax

C/C++:	**int gPrintf**(char format, ...);

Arguments

format
Character string

Description

The function gPrintf() writes a formatted string to the currently nominated device through the routine gDisplayStr(). The function returns the number of characters transferred if successful.

This function provides a combination of gDisplayStr() and printf(3) functionality for GINO devices. Strings can therefore contain GINO escape sequences as well as printf escape and formatting functions and users are referred to gDisplayStr() and chapter 3 of their UNIX documentation for further information on this function.

See Also

Page 138
gDisplayStr

gPushTransform

Syntax

C/C++:	**void gPushTransform**(void);

F90:	**subroutine gPushTransform**

Arguments

None

Description The routine gPushTransform() stores a copy of the current modelling transformation.

Up to ten different copies of modelling transformation sequences may be stored simultaneously by successive calls to gPushTransform().

See Also Page 376
gPopTransform

gReduceBezier

Syntax

C/C++:	**void gReduceBezier2D**(int *npts, GPOINT *points2); **void gReduceBezier3D**(int *npts, GPOINT3 *points3);

F90:	**subroutine gReduceBezier2D**(npts, points2) **subroutine gReduceBezier3D**(npts, points3) **integer, intent(inout) :: npts** **type (GPOINT), intent(inout) :: points2(*)** **type (GPOINT3), intent(inout) :: points3(*)**

Arguments ***npts***
Number of points in Bezier curve, decremented by one on return.

points2,points3
Array holding absolute coordinates of Bezier curve control points.

Description The gReduceBezier set of routines takes a set of Bezier curve control points in 2D or 3D and generates a new set, with one less control point, that represents an approximation of the same curve.

Note that the value passed in **npts** will be decremented by one on return and the arrays **points2** or **points3** will contain one less significant control point in them.

See Also Page 103
Page 290
gElevateBezier2D
gElevateBezier3D

gRemoveDir

Syntax

C/C++:	**int gRemoveDir**(char path[]);

F90:	**integer function gRemoveDir**(path) character*(*), intent(in) :: path

Arguments ***path***
Name of directory

Description The system utility gRemoveDir() removes the specified directory from the file store. The directory may may be simple directory name in the current working directory or a full path name.

The function returns an integer value which equal zero if the removal has been successful. A system dependent error code is returned if the removal fails if, for example, access to the parent directory is not permitted.

See Also Page 461
gMakeDir

gRemoveEventType

Syntax

C/C++:	**void gRemoveEventType**(int intype);

F90:	**subroutine gRemoveEventType**(intype) integer, intent(in) :: intype

Arguments

intype
Event type

= GALL,	All event types
= GNULL,	Null event type
= GKEYPRESS,	Key or mouse button press
= GSEGMENT,	Picture segment number
= GSEGMENTANDKEY,	Picture segment number and key/mouse button
= GLOCATOR,	Screen position and key/mouse button press
= GSTRING,	Text string
= GREALS,	String of real value
= GINTEGERS,	String of integer values
= GMOVEMENT,	Pointer, mouse or tablet movement
= GKEYRELEASE,	Key or mouse button release
= GRESIZE,	Window resize event
= GPOINTERLEAVING,	Pointer leaving window
= GPOINTERENTERING,	Pointer entering window
= GMOUSEWHEEL,	Mouse wheel movement

Description The routine gRemoveEventType() deletes an event type from the list set up by gAddEventType().

If **intype** is set to GALL, all event types are removed from the list and gAddEventType() must be called before any further data can be read with the routine gWaitForEvent().

See Also Page 450
gAddEventType

gRemoveFile

Syntax

C/C++:	**int gRemoveFile**(char file[]);

F90:	**integer function gRemoveFile**(file) character*(*), intent(in) :: file

Arguments

file
Name of file

Description

The system utility gRemoveFile() removes the specified file from the file store. The file may may be simple filename in the current working directory or a full path name.

The function returns an integer value which equal zero if the removal has been successful. A system dependent error code is returned if the removal fails if, for example, access to the parent directory is not permitted.

See Also

Page 461
gCopyFile
gRenameFile

gRemoveSegGroup

Syntax

C/C++:	**void gRemoveSegGroup**(int ngrp);

F90:	**subroutine gRemoveSegGroup**(ngrp) integer, intent(in) :: ngrp

Arguments

ngrp
Segment group number

= -1, Remove all segment groups

Description

The routine gRemoveSegGroup() removes segment group **ngrp**. Further references to segment **ngrp** will be treated as references to a single segment. If **ngrp** is set to -1, gRemoveSegGroup() removes all segment groups.

See Also

Page 437
gDefineSegGroup

gRenameFile

Syntax

C/C++:	**int gRenameFile**(char filea[], char fileb[]);

F90:	**integer function gRenameFile**(filea, fileb) character*(*), intent(in) :: filea, fileb

Arguments

filea
Name of existing file

fileb
Name of new file

Description

The system utility gRenameFile() renames an existing file, **filea** as **fileb**. Either or both file names may be simple file names in the current working directory or full path names.

The function returns an integer value which equal zero if the rename has been successful. A system dependent error code is returned if the rename fails.

See Also

Page 461
gCopyFile
gRemoveFile

gRenameSeg

Syntax

C/C++:	**void gRenameSeg**(int nseg, int newseg);

F90:	**subroutine gRenameSeg**(nseg, newseg) integer, intent(in) :: nseg,newseg

Arguments

nseg
Picture segment number

newseg
New number to be given to picture segment **nseg**

Description

The routine gRenameSeg() is called to rename picture segment **nseg** as **newseg**. The result is a single segment with number **newseg**.

If **nseg** = **newseg**, no action is taken.

If **newseg** already exists, it is deleted.

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(GHARDWARE), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also Page 427
gSetSegMode

gRestoreGinoState

Syntax

C/C++:	**void gRestoreGinoState**(int map);

F90:	**subroutine gRestoreGinoState**(map) integer, intent(in) :: map

Arguments

map

Restore mapping mode

= GABSOLUTE,	Restore without mapping to device limits
= GMAPPED,	Map to current device limits (default)

Description

The routine gRestoreGinoState() restores the setting of all GINO's output attributes to the state when the last call to gSaveGinoState() was made. Further details of the attributes stored is documented under gSaveGinoState().

The argument **map** determines the restoration mapping mode. When **map**=GABSOLUTE, the viewport parameters will be restored as they were saved by gSaveGinoState(), and therefore no mapping will occur. When **map**=GMAPPED, the viewport parameters will be mapped to the current device limits so that the same portion of the drawing area will be used on the current device. This mode is useful when restoring a set of GINO attributes onto a different device from which they were saved and which may have different device limits. Any subsequent drawing will then be automatically mapped onto the device without having to change coordinate system.

As implied by the previous paragraph, GINO attributes may be saved across device nominations. The set of attributes are also saved on a stack, so gRestoreGinoState() will always restore the last set of saved attributes.

See Also Page 54
gSaveGinoState

gRestoreTransform

Syntax

C/C++:	**void gRestoreTransform(void);**

F90:	**subroutine gRestoreTransform**

Arguments

None

Description

The routine gRestoreTransform() restores the state of transforming saved by the last call to gSaveTransform().

See Also Page 376
gSaveTransform

gRetrieveSegs

Syntax

C/C++:	**void gRetrieveSegs**(GFILE *fp);

F90:	**subroutine gRetrieveSegs**(unit) integer, intent(in) :: unit

Arguments

fp
GINO-C file pointer to restore Software Display File from

unit
Fortran 90 file unit to restore Software Display File from

Description

The routine gRetrieveSegs() restores the complete contents of an archived Software Display File.

The archived Display File must be created with gArchiveSegs() and should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

The display file is copied into memory or file depending on whether gDefineSegWorkspace() has been called. If gDefineSegWorkspace() has been called but the restored file does not fit into the allocated space an error message is generated and no action is taken.

The successful action of restoring a display file removes all currently stored segments.

See Also

Page 442
gFopen
gArchiveSegs
gSetSegMode

gReturnDirDate

Syntax

C/C++:	**void gReturnDirDate**(int pdate, GDATE *date, GTIME *time);

F90:	**subroutine gReturnDirDate**(pdate, date, time) integer, intent(in) :: pdate type (GDATE), intent(out) :: date type (GTIME), intent(out) :: time

Arguments

pdate
Packed date/time

date.year, date.month, date.day
Year/Month/Day of creation/modification date

time.hour, time.min, time.sec, time.millsec
Hour/Minute/Second/Millisecond of creation/modification time

Description The system utility gReturnDirDate() unpacks date and time information returned by the routine gGetFullDirList(). The packed date and time is associated with the creation or modification date of a file returned by this routine and may be packed in a system dependent way.

See Also Page 463
gGetFullDirList

gReturnInternalPoints

Syntax

C/C++:
int gReturnInternalPoints2D(int nn, GPOINT *points2, int np, GPOLYGON *polyline2, int *npts, int * npol);
int gReturnInternalPoints3D(int nn, GPOINT3 *points3, int np, GPOLYGON3 *polyline3, int *npts, int *npol);

F90:
integer function gReturnInternalPoints2D(nn, points2, np, polyline2, npts, npol)
integer function gReturnInternalPoints3D(nn, points3, np, polyline3, npts, npol)

integer, intent(in) :: nn,np
type (GPOINT), intent(out) :: points2(*)
type (GPOINT3), intent(out) :: points3(*)
type (GPOLYGON), intent(out) :: polyline2(*)
type (GPOLYGON3), intent(out) :: polyline3(*)
integer, intent(out) :: npts,npol

Arguments

nn
Size of points array

points2,points3
Array to contain internal vertices from point storage workspace

np
Size of polyline array

polyline2,polyline3
Array to contain internal vertices from point storage workspace

npts
Size of polyline array, returns number of points stored

npol
Size of polyline array, returns number of polylines stored

Description The functions gReturnInternalPoints2D() and gReturnInternalPoints3D() fill two arrays with information about visible lines that have been drawn while internal point storage is switched on. Vertices are stored in space (untransformed) or picture (transformed) mode according to the current point storage mode as set by gSetPointMode().

The functions return all the vertices in the array **points2/3** together with an array of polylines in the array **polyline2/3** where the vertex pointers in the structures **polyline2/3** point to relevant vertices in the **points2/3** array.

The arguments **nn** and **np** should be set to the size of the respective arrays that have been passed to the routines, and arguments **npts** and **npol** return the number of points and polylines that are in the internal storage workspace. The functions themselves return the actual number of polylines that have been returned in the user supplied arrays which may be less than the number stored if there is not enough room in the supplied arrays. The total number of vertices that can be stored is limited by the amount of point storage workspace that has been allocated by gDefinePointWorkspace().

The returned polyline array can be drawn or filled using the gDrawPolylineSet2D/3D or gFillPolygonSet2D/3D routines.

See Also

Page 104
Page 293
gDefinePointWorkspace
gDrawPolylineSet2D
gDrawPolylineSet3D
gFillPolygonSet2D
gFillPolygonSet3D
gSetPointMode

gReturnPlanarNormal

Syntax

C/C++: **void gReturnPlanarNormal**(int npts, GPOINT3 *points, GPOINT3 *normal);

F90: **subroutine gReturnPlanarNormal**(npts, points, normal)

```
integer, intent(in) :: npts
type (GPOINT3), intent(in) :: points(*)
type (GPOINT3), intent(out) :: normal
```

Arguments

npts
Number of vertices in facet/surface

points
Array specifying series of absolute points that define a facet/surface boundary

normal
Returned planar normal

Description

The routine gReturnPlanarNormal() returns a vector representing the planar normal of the vertices passed in the array **points**. It is assumed that the points in fact lie on a plane, as the routine only uses the first, second and last points passed to calculate the normal vector.

See Also

Page 297

gReturnStrInfo

Syntax

C/C++: **void gReturnStrInfo**(char string[], float *rlen, int *nnl, float *tch, float *sch, int *nesc);

F90: **subroutine gReturnStrInfo**(string, rlen, nnl, tch, sch, nesc)

```
character, intent(in) :: string
real, intent(out) :: rlen,tch,sch
integer, intent(out) :: nnl,nesc
```

Arguments

string
Character string

rlen
Maximum length of string

nnl
Number of lines contained in string

tch
Maximum character height

sch
Maximum height above base line

nesc
Number of non escape characters in string

Description

The routine gReturnStrInfo() returns information concerning the dimensions of a particular character string.

For strings without any *N escape sequences, **rlen** returns the total length of the string taking into account the width of every character in the string and the layout of exponents and indices, and **nnl** will return 1. If the string contains one or more of the *N escape sequences, **rlen** will return the length of the longest line in **string** and **nnl** will return the number of lines.

The length of the string is calculated by adding the width of each character taking into account variable widths of hardware or software proportional fonts. The same calculation is made for the layout of justified strings. If GINO is unable to obtain the widths of characters for hardware fonts the current character width as set by gSetCharSize() is used for every character.

tch measures the total height of the string including exponents, indices, descenders and multiple lines. **sch** measures only that part of the string above the base line (of the first line if there are multiple lines).

nesc returns the number of non escape characters within the string, that is the number of characters that will actually be displayed on output.

See Also

Page 159
gDisplayStr
gRotate

gRotate

Syntax

C/C++:	**void gRotate2D**(float angle); **void gRotate3D**(int axis, float angle);

F90:	**subroutine gRotate2D**(angle) **subroutine gRotate3D**(axis, angle) integer, intent(in) :: axis real, intent(in) :: angle

Arguments

axis

The axis about which the rotation is made

= GXAXIS,	about the X-axis
= GYAXIS,	about the Y-axis
= GZAXIS,	about the Z-axis

angle

The angle in degrees through which rotation takes place

Description

The routines gRotate2D() and gRotate3D() superimposes a rotation of **angle** degrees in the current modelling transformation matrix. In the case of gRotate2D() the angle is with respect to the Z axis, but with gRotate3D() the rotation is about the axis specified in the first argument. Rotation is right-handed in sense with respect to the relevant axis.

In space mode (the default) this angle is specified with respect to the current space axes, while in picture mode (see gSetTransformMode()) the angle is specified with respect to the picture axes.

If transforming is not on before this routine is called then gRotate3D() switches it on.

See Also

Page 228, 360
gSetTransformMode

gSaveGinoState

Syntax

C/C++:	**void gSaveGinoState**(void);

F90:	**subroutine gSaveGinoState**

Arguments

None

Description

The routine gSaveGinoState() saves the current setting of all the GINO attributes which can later be restored using the routine gRestoreGinoState(). The attributes are saved on a stack, so an application may make multiple calls to gSaveGinoState() before calling gRestoreGinoState().

The attributes saved include the following:

Device Limits

Linestyle and broken line tables and current settings

Hatch tables and current settings

Character and font settings

RGB table and current colour settings

Viewport parameters

Transformation and viewing settings

Rectangular/Polygonal clipping and masking limits

See Also

Page 54
gRestoreGinoState

gSaveLineStyle

Syntax

C/C++:	**void gSaveLineStyle**(int line);

F90:	**subroutine gSaveLineStyle**(line) integer, intent(in) :: line

Arguments

line

Line style index

= GCURRENT,	Current line style
= 1 - 256,	Stored line style

Description

The routine gSaveLineStyle() copies the current line attributes into the specified line style. GINO can record up to 256 different line styles. If **line** = GCURRENT, gSaveLineStyle() does nothing. If **line** is out of range, an error message is output and no further action is taken.

See Also

Page 131
gDefineBrokenLineStyle
gSetBrokenLine
gDefineLineStyle

gSaveTransform

Syntax

C/C++:	**void gSaveTransform**(void);

F90:	**subroutine gSaveTransform**

Arguments

None

Description The routine gSaveTransform() makes a copy of the state of transforming. This can subsequently by recalled by gRestoreTransform(). A second call to gSaveTransform() overwrites the information stored by the first call.

See Also Page 376
gRestoreTransform

gScale

Syntax

C/C++:	**void gScale2D**(float sx, float sy); **void gScale3D**(float sx, float sy, float sz);

F90:	**subroutine gScale2D**(sx, sy) **subroutine gScale3D**(sx, sy, sz) real, intent (in) :: sx,sy,sz

Arguments

sx,sy,sz
Scale factors for X,Y and Z coordinates respectively

Description The routines gScale2D() and gScale3D() set independent scale factors for X,Y (and Z) coordinates by modifying the current modelling transformation matrix.

In space mode (the default) this scaling is specified with respect to the current space axes, whilst in picture mode (see gSetTransformMode()) the scaling is specified with respect to the picture axes.

For negative values of **sx**, **sy** and **sz**, the routine mirrors the drawing about the axis by the absolute scale factor **sx**, **sy** or **sz**.

Transforming is switched on if not on already.

See Also Page 229, 363
gSetTransformMode

gSelectDrawingArea

Syntax

C/C++:	**void gSelectDrawingArea**(int ident);

F90:	**subroutine gSelectDrawingArea**(ident) integer, intent(in) :: ident

Arguments

ident
Drawing area identifier

= 0,	Screen
= 1,	Backing store (default)
> 1,	User generated auxiliary drawing area

Description

The routine gSelectDrawingArea() switches drawing to the selected drawing area. The default drawing area for all devices is 1, which represents the visible part of the screen or drawing media. On window devices, this identifier actually represents the backing store (or pixmap) which is automatically copied to the visible window through internal procedures. On such devices, selecting drawing area 0 may provided faster display times, but at the expense of possible display loss by the overlap of windows from other applications. Note that, drawing can be directed to the backing store only, through the operation of the gStartBatchUpdate()/gEndBatchUpdate() routines.

Additional drawing areas may be selected where these have been successfully opened using the routine gOpenAuxDrawingArea(). These auxiliary drawing area may be visible or invisible depending on whether the identifier is even or odd. All (even numbered) visible drawing areas will have an invisible drawing area associated with them (with the same operational criteria as the default drawing area as described above), but (odd numbered) invisible drawing areas may exist on their own. After selection, the current viewport and window limits are altered to match the size of the selected drawing area and the current point is reset to 0,0, but other GINO attributes remain unaltered.

The requested drawing area identifier must be within the permitted range for the currently nominated device and represent a valid opened area. Any invalid identifier will generate an error message and no change will be made to the currently selected area.

See Also

Page 50
gOpenAuxDrawingArea
gEnqDeviceState
gStartBatchUpdate

gSelectPolygons

Syntax

C/C++:	**void gSelectPolygons**(int list[], int n);

F90:	**subroutine gSelectPolygons**(list, n) integer, intent(in) :: list(*),n

Arguments

list

Integer array containing a list of polygon identifiers

n

Number of items in list

= 0,	Delete list
> 0,	Create new list

Description

The routine gSelectPolygons() specifies a current list of polygon identifiers. The list is copied into a workspace. Allowance must be made for this workspace when calling gSetWorkspaceLimit(). If **n**=0 the list is deleted. The list enables polygons to be selected (e.g. for area fill). If no list is currently defined then all polygons are selected.

See Also

Page 254
gSetWorkspaceLimit

gSetAlphaMode

Syntax

C/C++:	**void gSetAlphaMode**(void);

F90:	**subroutine gSetAlphaMode**

Arguments

None

Description

The routine gSetAlphaMode() switches non-windowing graphical devices into alphanumeric mode so that the user may read from or write to them via I/O systems other than GINO.

The routine gSetAlphaMode() forces out any graphic data held in device buffers, stores the current position, and selects character mode.

When the next GINO routine is called the stored current drawing position is restored and drawing may continue.

See Also

Page 50, 51

gSetArcIncrement

Syntax

C/C++:	**void gSetArcIncrement**(int nincs);

F90:	**subroutine gSetArcIncrement**(nincs) integer, intent(in) :: nincs

Arguments

nincs

Arc increments

> 0,	Number of increments to generate full circle
= 0,	Reset to default tolerance

Description

The routine gSetArcIncrement() sets the number of increments to be used to draw arcs unless hardware arcs are being drawn. Arcs are generated by drawing an inscribed polygon which would have **nincs** sides in a full circle. If **nincs** is less than 3 or greater than 32000 the default arc tolerance takes effect. gSetArcIncrement() disables the drawing of hardware arcs.

This routine also controls the number of line segments used between successive points in Akima and spline curves. A default of 15 is used if gSetArcIncrement() is not called or **nincs** is less than 3 or greater than 32000.

See Also

Page 89

gSetArcMode

Syntax

C/C++:	**void gSetArcMode**(int sw);

F90:	**subroutine gSetArcMode**(sw) integer, intent(in) :: sw

Arguments

sw

Arc generation switch

= GHARD,	Hardware arcs (default)
= GSOFT,	Software arcs

Description

The routine gSetArcMode() switches the software generation of arcs on or off.

By default, hardware arcs are produced. When hardware arcs are requested, arcs will be generated by hardware on devices with hardware arc generating facilities. If the device has no hardware arcs, then software arcs will be generated.

Hardware arcs are subject to the current line mode, and can be windowed and transformed. When transforming hardware arcs, if the resulting arc is not circular it will be generated using software.

Routines gSetArcIncrement() and gSetArcTolerance() have no effect when hardware arcs are being drawn.

See Also

Page 89

gSetArcTolerance

Syntax

C/C++:	**void gSetArcTolerance**(float tol);

F90:	**subroutine gSetArcTolerance**(tol) real, intent(in) :: tol

Arguments

tol

Maximum distance in current units between the circumference of a smooth arc and straight line segment generated as a representation of the arc

Description

The routine gSetArcTolerance() controls the smoothness of arcs and standard GINO curves.

It sets the tolerance to which arcs must be drawn if hardware arcs are not being drawn.

GINO will approximate to arcs by drawing an inscribed polygon such that the maximum distance between the required arc and the approximating polygon is **tol**. If **tol** = 0.0 the default value is restored. The arc tolerance is ignored if hardware arcs are being drawn.

See Also

Page 89

gSetBrokenLine

Syntax

C/C++:	**void gSetBrokenLine**(int brk);

F90:	**subroutine gSetBrokenLine**(brk) integer, intent(in) :: brk

Arguments

brk

Broken line type

= GSOLID,	Solid (default)
= GSHORTDASHED,	Short dashed
= GSHORTDOTTED,	Short dotted
= GSHORTCHAINED,	Short chained
= GLONGDASHED,	Long dashed
= GLONGDOTTED,	Long dotted
= GLONGCHAINED,	Long chained
= GDOTTED,	Dotted
= 8 - 256,	User defined or device dependent
> 256,	Solid or device dependent

Description

The routine gSetBrokenLine() selects a broken line type for subsequent graphical output. If **brk** is less than zero, a warning message is output and the absolute value of **brk** is used to set the broken line type.

When **brk** is in the range 1 to 256, a further set of parameters come into effect. These parameters explicitly define the appearance of the line type. Each line type is provided with a set of default parameters which may subsequently be changed by calling gDefineBrokenLineStyle(). Even if the device cannot generate the line type exactly as specified, an equivalent hardware-generated line type may still be selected. Appendix B should be consulted to see what line types the device can provide.

If **brk** is greater than 256, and the device cannot generate the requested line type (see Appendix B), the solid line type will be used for subsequent graphical output.

The effect of a call to gSetBrokenLine() can depend on the capabilities of the device. gSetBrokenLineMode() may be called to force GINO to generate all broken lines, thereby ensuring that they are output correctly on any device.

See Also

Page 116
gDefineBrokenLineStyle
gSetBrokenLineMode
gEnqBrokenLine
Appendix B

gSetBrokenLineMode

Syntax

C/C++:	**void gSetBrokenLineMode**(int sw)**;**

F90:	**subroutine gSetBrokenLineMode**(sw) integer, intent(in) :: sw

Arguments

sw
Software broken line switch

=GHARD,	Hardware-generated if possible (default)
=GSOFT,	Software-generated

Description

The routine gSetBrokenLineMode() allows the user to specify whether or not a device will be allowed to generate broken lines after a call to gSetBrokenLine(). The default, corresponding to a call to gSetBrokenLineMode(GHARD), is for broken lines to be generated if possible by the device.

The appearance of hardware-generated broken lines will depend on the capabilities of the device (see Appendix B). A call to gSetBrokenLineMode(GSOFT) will force GINO to generate all broken lines. This may slow down the generation of broken lines, but it will ensure that they will be output exactly as requested on any device.

See Also

Page 116
gSetBrokenLine
Appendix B

gSetCharFont

Syntax

C/C++:	**void gSetCharFont**(int font);

F90:	**subroutine gSetCharFont**(font) integer, intent(in) :: font

Arguments

font
Font number

= 0,	GDEFAULT
Software Fonts	
= 1,	GRoman_Simplex
= 2,	GRoman_Duplex
= 3,	GRoman_Complex
= 4,	GRoman_Triplex
= 5,	GItalic_Complex
= 6,	GItalic_Triplex
= 7,	GScript_Simplex

= 8,	GScript_Complex
= 9,	GGreek_Simplex
= 10,	GGreek_Complex
= 11,	GGothic_English
= 12,	GGothic_German
= 13,	GGothic_Italian
= 14,	GCyrillic_Complex
= 15,	GSwiss_Solid*
= 16,	GDutch_Solid*
= 17,	GWestern*
= 18,	GComputer*
= 19,	GDisplay*
= 20,	GLatin*
= 21,	GGreek_Font_1
= 22,	GGreek_Font_2
= 23,	GGreek_Font_3*
= 24,	GGreek_Font_4*
= 25,	GGreek_Font_5*

Symbol Fonts

= 70,	GMaths_Symbols*
= 71,	GHershey_Maths_Symbols
= 72,	GHershey_Symbols_1
= 73,	GHershey_Symbols_2
= 74,	GSymbol1_normal*
= 75,	GSymbol1_thick*
= 76,	GSymbol1_filled*
= 77,	GSymbol2_normal*
= 78,	GSymbol2_filled*
= 79,	GGINO_Dingbats*

Hardware and Software Fonts

= 100,	GCourier (hardware only)
= 101,	GHelvetica
= 102,	GTimes
= 103,	GAvant_Garde
= 104,	GLublin_Graph
= 105,	GNew_Century_Schoolbook
= 106,	GSouvenir
= 107,	GPalatino
= 108,	GChancery
> 108,	device specific hardware fonts

All fonts are proportional except 0 and 100
*=Polygon Font

Description

The routine gSetCharFont() sets the current character font to be used by all subsequent character and string output routines. The font may be selected by number or name. The current font can also be selected using the *Fnnn escape sequence within the character string routines.

Font 0 is the default font and is always available in software and on most devices in hardware as well. It is always a fixed pitch font.

Fonts 1 to 69 are reserved for software fonts of which the ones listed are currently available in the GINO font file. They are all proportional fonts.

Fonts 70 to 99 are reserved for symbol fonts of which the ones listed are currently available in the GINO font file. These fonts are primarily provided for use with the gDrawMarker() routine. Additional hardware symbol fonts may be available on the current device.

Fonts 100 to 199 are reserved for fonts of which the ones listed are currently registered. Registered fonts are increasingly available on a number of graphics devices but in some cases only certain sizes are provided. With registered fonts (101-108) a software polygon font is also provided in the GINO-F font file so that if software output is requested or the currently selected device does not support this font, the appearance of text matches approximately that of the requested font. Additional (non-registered) hardware fonts may be provided on the current output device and users should consult the relevant Appendix B documentation for further information on these. There is no software form of non-registered fonts.

The selection of hardware or software font is determined by the current setting of character attributes. Where software fonts are used gSetFontForm() can be used to simplify their representation during program development.

If the length of a string is required for a proportional font it is recommended that the routine gReturnStrInfo() is used as it cannot be calculated by multiplying the number of characters by the character width. Alternatively, gSetFontSpacing() can be used to force any font to be output as if it were fixed pitch.

gSetFontFillStyle() can be used to select different font fill styles (e.g. outline and hatch fill) for hardware fonts and software fonts marked with a *. gSetFontWeight() can be used to alter the weight of all fonts.

The list of hardware fonts that are available on the current output device can be obtained through the gEnqHardFontList() routine. If the font requested is not available, font zero is used. Negative font numbers cause a warning message and font zero is selected.

See Also

Page 141
gDrawMarker
gEnqHardFontList
gSetFontForm
gSetFontSpacing
gSetFontFillStyle
gSetFontWeight
gReturnStrInfo

gSetCharSize

Syntax

C/C++:	**void gSetCharSize**(float width, float height); **void gSetCharSizePoint**(float points)

F90:	**subroutine gSetCharSize**(width, height) **subroutine gSetCharSizePoint**(points) real, intent(in) :: width,height real, intent(in) :: points

Arguments

width
Character width in current units

height
Character height in current units

points
Character size in points (1/72nd inch)

Description

The routines gSetCharSize() and gSetCharSizePoint() sets the current character size to the specified size. If **width**, **height** or **points** are less than or equal to zero, the current character size is reset to the default character size and a warning message is output. The routine gSetCharSizePoint() sets the width and height to the same dimension. The character width specifies the distance between the start of one character and the next and the character height specifies the height of upper-case characters.

The effect a call to either routine has when characters are output, depends on the character mode at that time:

(a) gSetHardCharSize()

Neither routine has any effect.

(b) gSetHardChars()

Characters are correctly spaced, but their size may be very different.

(c) gSetMixedChars()

Character size may differ by up to 10% from the requested size.

(d) gSetSoftChars()

Characters are output exactly as requested.

(e) gSetCharTransformMode(GON)

Characters are transformed so size is modified by current scale.

See Also

Page 147
gSetHardCharSize
gSetCharTransformMode
gSetHardChars
gSetMixedChars
gSetSoftChars

gSetCharTransformMode

Syntax

C/C++:	**void gSetCharTransformMode**(int sw);

F90:	**subroutine gSetCharTransformMode**(sw) integer, intent(in) :: sw

Arguments

sw
Character transforming switch

= GOFF,	Off (default)
= GON,	On

Description

The routine gSetCharTransformMode() switches the transforming of characters on and off. The call to gSetCharTransformMode() takes effect when characters are subsequently output provided there has been no intervening call to gSetHardCharSize().

When the transforming of characters is switched on, all characters are generated by GINO to exactly the size, orientation and italic angle requested. They are also drawn subject to the current transformation, clipping/masking limits and line style (ie. according to the current line width, broken line style and line end type).

If gSetCharTransformMode(GOFF) is called, the current character mode reverts to the character mode set by the last call to gSetHardChars(), gSetMixedChars() (default) or gSetSoftChars().

See Also

Page 160, 239
gSetHardCharSize
gSetHardChars
gSetMixedChars
gSetSoftChars

gSetClippingMode

Syntax

C/C++:	**void gSetClippingMode**(int sw);

F90:	**subroutine gSetClippingMode**(sw) integer, intent(in) :: sw

Arguments

sw
Clipping mode

= GNOCLIP,	Switch off hardware and software clipping
= GHARD,	Switch on hardware clipping
= GSOFT,	Switch on software clipping

Description

The routine gSetClippingMode() sets the clipping mode for all GINO drawing.

By default GINO will use hardware clipping if the current device has the facility in the driver. Otherwise GINO will perform its own clipping. In both cases graphics is clipped to the current window, this having been set using gSetWindow2D()/gSetWindow3D(), the current viewport limits (if gSetViewportClipSwitch(GOFF) has been called) or the device limits if none of these conditions apply.

Where a device has hardware clipping facilities, it should be noted that this will affect pixel output, whereas the pixel primitives are not affected by software clipping. Hardware clipping is, in most cases, more efficient than software clipping.

In the rare instance of not requiring hardware or software clipping, both operations can be switched off by using GNOCLIP as the argument to gSetClippingMode(). Where this mode is used, the effect of drawing outside the device limits vary from device to device and may corrupt output. However, where it is known that output is restricted to within the device limits, this mode of operation is likely to be significantly faster than with hardware or software clipping switched on.

Very few devices have the capability of hardware clipping - use the routine gEnqDeviceState() to check the settings for the current device.

See Also

Page 222
gEnqClippingMode
gEnqDeviceState
gSetViewportClipSwitch
gSetWindow2D
gSetWindow3D

gSetColourInfo

Syntax

C/C++:	**void gSetColourInfo**(int ndc, int ndt);

F90:	**subroutine gSetColourInfo**(ndc, ndt) integer, intent(in) :: ndc,ndt

Arguments

ndc

Number of colours that can be separately defined or selected

= 0,	Monochrome device
= 1,	Monochrome device with background erase
> 1,	Colour/greyscale device

ndt

Display type, identifying the colour/greyscale capabilities of the device

= 0,	No colour/greyscale variation (i.e. Monochrome)
= ±1,	Fixed colour/greyscale

= ±2,	Static colour/greyscale
= ±3,	Dynamic colour/greyscale
= ±4,	Direct colour/greyscale

N.B. Positive values indicate colour display, negative values indicate greyscale.

Description

The routine gSetColourInfo() can be used to restrict the colour settings of the currently nominated device. **ndc** and **ndt** define the number of colours and the colour type of the device and these may be set to less than or equal to those defined by the device driver at device initialisation. The maximum colour settings may be enquired through the routine gEnqDeviceState() and the current settings may be enquired through the routine gEnqColourInfo().

ndc defines the number of colours/greyscales that can be defined and selected (see gDefineRGB() & gSetLineColour()). If **ndc** is set to zero, the device cannot display any colours. **ndt** defines the colour/greyscale capabilities of the device as follows:

ndt= 0

No colours or greyscales are available.

ndt= ±1

The device has a fixed colour palette which cannot be redefined, gDefineRGB() has no effect.

ndt= ±2

The device has a static colour palette. gDefineRGB() can be used to redefine colours but colours already drawn on the device will not be affected.

ndt= ±3

The device has a dynamic colour palette, gDefineRGB() can be used and redefining a colour that has already been used will have immediate effect on the colour on the screen.

ndt= ±4

The device has a true colour capability and will associate an RGB value with each pixel on the device instead of using a look-up table. A pseudo colour table will be maintained by GINO or the device from which RGB values will be extracted where colour indices are used.

Only certain combinations of **ndt** can be set for any one device, for example if a device has **ndt** set to 2 it cannot be changed to 3 and vice versa, however most devices can be restricted by setting **ndt** to 0 or 1 and some can change from 2 to 4 and vice versa.

ndt can also be changed from positive to negative to force GINO to turn all colours into greyscales.

This routine may only be called after a device has been nominated and before the start of the first picture.

See Also

Page 47, 209
gEnqColourInfo
gEnqDeviceState
gSetLineColour
gDefineRGB

gSetCursorAction

Syntax

C/C++:	**void gSetCursorAction**(int action, int lverts, GPOINT *points);

F90:	**subroutine gSetCursorAction**(action, lverts, points) integer, intent(in) :: action, lverts type (GPOINT), intent(in) :: points(*)

Arguments

action
Cursor action type

= GDEFAULT,
= GRUBBERBAND,
= GRUBBERBOX,
= GRUBBERSQUARE,
= GRUBBERELLIPSE,
= GRUBBERCIRCLE,
= GPOLYLINE,

lverts
Number of vertices if polyline cursor

points
Coordinates of polyline cursor

Description

The routine gSetCursorAction() sets the action of the cursor or pointer when the gGetCursorEvent() routine is used.

When **action** = GDEFAULT, the normal action of the cursor or pointer is set such that it marks the current pointer position.

When **action** = GPOLYLINE an additional shape is displayed at the same time as the cursor and is continually updated as the cursor is moved. The rubber shape is drawn in XOR mode using the colour **bakcol** set by gSetCursorType() on most devices. Each shape has a static position which is the current pen position when gGetCursorEvent() is called and has a variable position which follows the cursor.

When **action** = GRUBBERBAND, or GRUBBERBOX a rubber line or box is displayed between the two points. When **action** = GRUBBERSQUARE, a rubber square is displayed, the dimension of each side being the minimum distance between the current pen position and the cursor. When **action** = GRUBBERELLIPSE, a rubber ellipse is displayed within the bounding rectangle whose centre is the current pen position and corner is the cursor position. When **action** = GRUBBERCIRCLE, a rubber circle is displayed in the same way as the ellipse except the bounding box is kept to a square centred at the current pen position.

When **action** = GPOLYLINE, the user can define a polyline which will follow the current pointer position. The number of vertices is set in **lverts** and the coordinates are placed in the array **points**. The coordinates are defined in absolute picture coordinates. When gGetCursorEvent() or gWaitForEvent() is called the coordinate (0.0,0.0) will follow the pointer position. A maximum of 200 vertices can be sent to the device. If more that 200 vertices are requested, a warning message is output and only the first 200 are used. The polyline is not subject to any GINO windowing or masking.

If **action** is outside the above range an error message is output and no change is made to the action type.

This routine only affects the shape of the cursor when **action**=GPOLYLINE, The routine gSetCursorType() should be used to set other shapes.

The availability of cursor action types is hardware dependent and users should refer to the relevant Appendix B document for the current device being used.

See Also

Page 243
gGetCursorEvent
gSetCursorType
gWaitForEvent
Appendix B

gSetCursorPos

Syntax

C/C++:	**void gSetCursorPos**(float x, float y);

F90:	**subroutine gSetCursorPos**(x, y) real, intent(in) :: x,y

Arguments

x,y
Cursor start position in picture coordinates

Description

A call to gSetCursorPos() defines the start position for the graphics pointer or cursor on non-windowing devices. If cursor positioning is supported by the device (see Appendix B), the pointer/cursor will appear at (**x,y**) when gGetCursorEvent() is next called.

The default for the pointer/cursor start position is the centre of the device limits. The cursor start position is reset to the default after any call to gNewDrawing(). If a call to gGetCursorEvent() is successful, the cursor start position may be set to the position returned by gGetCursorEvent() depending on the device.

The position (**x,y**) should be specified to be within the device limits. If it is outside these, a warning message is output and the cursor start position defaults to the centre of the device limits.

The routine gSetMousePos() can be used to set the position (in pixels) of the graphics pointer on windowing devices.

See Also

Page 242
gGetCursorEvent
gNewDrawing
gSetMousePos
Appendix B

gSetCursorType

Syntax

C/C++:	**void gSetCursorType**(int type, int forcol, int bakcol);

F90:	**subroutine gSetCursorType**(type, forcol, bakcol) integer, intent(in) :: type,forcol,bakcol

Arguments

type
Cursor Type

= GDEFAULT,	Default
= GSMALLCROSS,	Small cross
= GLARGECROSS,	Large cross (full screen/window if available)
= GX,	X
= GPOINTER,	Pointer
> 4,	Hardware dependent cursor types

forcol
Foreground colour index (default = 1)

bakcol
Background colour index (default = 0)

Description

The routine gSetCursorType() sets the shape of the gGetCursorEvent() or pointer for use within the gGetCursorEvent() or gWaitForEvent() routines.

When **type** = GDEFAULT, a default hardware cursor is defined. This may be a cross or pointer.

When **type** > 4, other cursor shapes can be defined most of which are dependent on the facilities of the device being used. Refer to the appropriate Appendix B document for a list of cursor types that are available.

If **type** is greater than the number of cursor types available on the current device (see gEnqDeviceState()) the default hardware cursor shape is used.

The arguments **forcol** and **bakcol** define the foreground and background/outline colours that are required for the cursor shape. The application of these colours is hardware dependent.

The shape of the cursor is changed either when the routine gGetCursorEvent() is called or when any event types are enabled with gAddEventType(). When GINO returns from gGetCursorEvent() or all event types are disabled (with gRemoveEventType()) the cursor is removed from the screen or its shape reverts to some default shape. Note that when **type** = GDEFAULT, the shape of the cursor is always different from that when GINO is not in cursor or event mode.

Some devices can define a polyline cursor of up to 200 vertices. The routine gSetCursorAction() is used for this option.

See Also

Page 242
gGetCursorEvent
gEnqDeviceState
gSetCursorAction
gWaitForEvent
Appendix B

gSetCurveAttribs

Syntax

C/C++:	**void gSetCurveAttribs2D**(float dxbeg, float dybeg, float dxfin, float dyfin, float xbeg, float ybeg, float xfin, float yfin); **void gSetCurveAttribs3D**(float dxbeg, float dybeg, float dzbeg, float dxfin, float dyfin, float dzfin, float xbeg, float ybeg, float zbeg, float xfin, float yfin, float zfin);

F90:	**subroutine gSetCurveAttribs2D**(dxbeg, dybeg, dxfin, dyfin, xbeg, ybeg, xfin, yfin) **subroutine gSetCurveAttribs3D**(dxbeg, dybeg, dzbeg, dxfin, dyfin, dzfin, xbeg, ybeg, zbeg, xfin, yfin, zfin) real, intent(in) :: dxbeg,dybeg,dzbeg,dxfin,dyfin,dzfin real, intent(in) :: xbeg,ybeg,zbeg,xfin,yfin,zfin

Arguments

dxbeg,dybeg,dzbeg
Slope angle/derivative for start of the curve

dxfin,dyfin,dzfin
Slope angle/derivative for end of the curve

xbeg,ybeg,zbeg
Extra point defining start angle of curve

xfin,yfin,zfin
Extra point defining end angle of curve

Description

The routines gSetCurveAttribs2D() and gSetCurveAttribs3D() specify the curve end conditions which may be used in calls to the GINO 2D and 3D curve drawing routines respectively. The curve end conditions define the direction of a curve at each end.

For the 2D piecewise cubic curves, the slopes are measured in terms of the cosine and sine of the angles at each end, whereas for the 2D and 3D spline curves, the slope is measured in terms of actual gradient and therefore will need to be scaled. For monotonic spline curves, it is sufficient to set **dxbeg**, **dxfin** = 1.0 and **dybeg**, **dyfin** = y'(x) and the routine will compute the correct values.

Alternatively, the direction may be defined by an extra point through which the curve would pass if that point was included with those that actually specify the curve.

In the absence of a call to gSetCurveAttribs2D() or gSetCurveAttribs3D(), the start and end slopes default to zero, i.e 1.0 and 0.0 for the cosine and sine of the angle, and the extra points (**xbeg**,**ybeg**,**zbeg**) and (**xfin**,**yfin**,**zfin**) both default to (0.0,0.0,0.0).

See Also Page 94, 100, 286, 287
gDrawAkimaBy2D
gDrawAkimaTo2D
gDrawCurveBy2D
gDrawCurveTo2D
gDrawSplineBy2D
gDrawSplineTo2D
gDrawSplineBy3D
gDrawSplineTo3D

gSetDebugSwitch

Syntax

C/C++:	**void gSetDebugSwitch**(int sw);

F90:	**subroutine gSetDebugSwitch**(sw) integer, intent(in) :: sw

Arguments

sw
Debug output switch

= GOFF,	Switch debug output off
= GON,	Switch debug output on (default if gDebug() is called)

Description

Following the nomination of the gDebug() intermediate device driver, its output may be controlled by switching its operation on and off using gSetDebugSwitch(). This allows the user to select gDebug() output for the desired section of the application program.

See Also

Page 32
gDebug

gSetDepthMode

Syntax

C/C++:	**void gSetDepthMode**(int mode, float dinit);

F90:	**subroutine gSetDepthMode**(mode, dinit) integer, intent(in) :: mode real, intent(in) :: dinit

Arguments

mode
Depth test mode

= GNEVER,	Never display output
= GLESSTHAN,	Display if depth < value in depth buffer (default)
= GLESSTHANOREQUALTO,	Display if depth <= value in depth buffer
= GEQUALTO,	Display if depth = value in depth buffer
= GNOTEQUALTO,	Display if depth >= value in depth buffer

= GGREATERTHANEQUALTO,	Display if depth >= value in depth buffer
= GGREATERTHAN,	Display if depth > value in depth buffer
= GALWAYS,	Always display output

dinit
Initial depth buffer setting (default = 1.0)

Description

The arguments to the routine gSetDepthMode() control the operation of the depth buffer which take effect when a flat or smooth shading mode is set in the next call to gSetShadingMode().

The setting of **mode** sets the test that is applied to each point that is to be displayed against the value in the depth buffer.

The depth buffer is initialized to the value in **dinit** when the display area is next cleared with gNewDrawing().

See Also

Page 328
gSetShadingMode
gSetViewport3D

gSetDeviceFilename

Syntax

C/C++: **void gSetDeviceFilename**(char filename[], int ntype);

F90: **subroutine gSetDeviceFilename**(filename,ntype)

character*(*), intent(in) :: filename
integer, intent(in) :: ntype

Arguments

filename
File name to be used for graphical output

ntype
Additional configuration/format information in connection with file

> 0,	Not used
= 0,	Unformatted output (default)
= -1,	Formatted 80 character records
= -2,	Expanded Ascii codes
= -3,	Expanded Octal codes
= -4,	Expanded Decimal codes
= -5,	Expanded Hex codes

Description

The routine gSetDeviceFilename() specifies an output file name and optional format for the graphics output to be sent to. This only applies to metafile and plotter devices and some graphics terminals as it is not possible to re-direct output from windowing devices.

Omitting to use gSetDeviceFilename() will result in device output being directed to a default output source which for most devices is a file called xxxxxx.OUT where xxxxx is the nomination routine name. Whether the default output source is used or that specified by this routine, an internal file unit is used for the output.

This routine may only be called after a device has been nominated and before the start of the first picture.

See Also Page 44
Appendix A
Appendix B

gSetDeviceTitle

Syntax

C/C++:	**void gSetDeviceTitle**(char title[]);

F90:	**subroutine gSetDeviceTitle**(title) character*(*), intent(in) :: title

Arguments

title
Device title string

Description

The routine gSetDeviceTitle() defines a device specific title string to be used as appropriate on the output device. The title string will be output in the title bar on window devices or in the header of metafiles.

This routine may be called during device qualification, ie. immediately after nominating the required device, in which case the title string will be displayed as part of opening the device or window. Some devices may allow the title or banner string to be changed while a device or window is open, in which case this routine may be called at any time during an application.

See Also Page 52
Appendix B

gSetDialogueVis

Syntax

C/C++:	**void gSetDialogueVis**(int diavis);

F90:	**subroutine gSetDialogueVis**(diavis) integer, intent(in) :: diavis

Arguments

diavis
Dialogue area visibility

= GINVISIBLE,	Dialogue area invisible
= GVISIBLE,	Dialogue area visible

Description

The routine gSetDialogueVis() sets the dialogue area to be visible or invisible. Setting it invisible only switches it off, it does not delete it.

This routine will only function on windowing devices and screens with separate graphics/dialogue planes (see Appendix B).

See Also Page 51
gSetGraphicsVis
Appendix B

gSetDrawingLimits

Syntax

C/C++:	**void gSetDrawingLimits**(GDIM *dim, int type);

F90:	**subroutine gSetDrawingLimits**(dim ,type) type (GDIM), intent(in) :: dim integer, intent(in) :: type

Arguments

dim
Physical dimensions of required drawing area in the current units:

= 0.0, Leave unaltered

type
Type of paper

Description

The routine gSetDrawingLimits() allows the selection of a variety of paper sizes on plotters, or window sizes on devices with window managers. This routine may only be called after a device has been nominated and before the start of the first picture, or after a call to gNewDrawing() and before the start of the next picture.

The default orientation can be changed on plotters/printers by calling gSetDrawingLimits() with the horizontal and vertical limits interchanged. Thus, if the default drawing area orientation is landscape and a call is made to gSetDrawingLimits() such that the vertical dimension is greater than the horizontal, the orientation is changed to portrait. The orientation can be changed from portrait to landscape in the same manner, but in reverse.

type can be used on some devices to select alternate paper trays - consult Appendix B for further information.

See Also

Page 45
Appendix B

gSetErrorFile

Syntax

C/C++:	**void gSetErrorFile**(GFILE *fp);

F90:	**subroutine gSetErrorFile**(unit) integer, intent(in) :: unit

Arguments

fp
Pointer to GINO-C Error/Tracer Message output file

unit
Fortran 90 file unit

Description The routine gSetErrorFile() sets the output unit for subsequent error and tracer messages. The default number is system dependent.

The required file should be opened prior to calling this routine using the function gFopen(), with the file pointer or unit number passed as appropriate.

Note that it is not possible to change the unit on which the GINO banner message is output as this routine does not take effect until after the banner has been produced.

See Also Page 31
gFopen
gSetTracerMode

gSetErrorMode

Syntax

C/C++:	**void gSetErrorMode**(int sw);

F90:	**subroutine gSetErrorMode**(sw) integer, intent(in) :: sw

Arguments

sw
Error and warning message switch

= GALLOFF,	Error and warning messages are suppressed
= GERRORON,	Warning messages are suppressed, error messages are output
= GALLON,	Error and warning messages are output (default)

Description GINO generates error messages and warning messages. The default is for GINO to output all messages. A call to gSetErrorMode() can arrange for either warning messages or all messages to be suppressed, according to the value of **sw**. Errors and warnings are always logged, even when output of the corresponding messages is suppressed.

See Also Page 30

gSetErrorTrap

Syntax

C/C++:	**void gSetErrorTrap**(int sw);

F90:	**subroutine gSetErrorTrap**(sw) integer, intent(in) :: sw

Arguments

sw
Error trapping flag

= GOFF,	Disable error trapping (default)
= GON,	Enable error trapping and reset error and warning count

Description The routine gSetErrorTrap() enables or disables error trapping, according to the value of **sw**. The default is for error trapping to be switched off. When error trapping is enabled, any errors or warnings that are generated are counted. The count is returned by gEnqNumberOfErrors() and it is reset whenever gSetErrorTrap(GON) is called. The count maintained for the purposes of error trapping is quite separate from the main error and warning count that is returned by gEnqLastErrors().

See Also Page 30
gEnqLastErrors
gEnqNumberOfErrors

gSetEscapeChar

Syntax

C/C++: **void gSetEscapeChar**(char cha);

F90: **subroutine gSetEscapeChar**(cha)
character*(*), intent(in) :: cha

Arguments ***cha***
Single character

Description The routine gSetEscapeChar() sets the escape character to **cha**. It is used in interpreting the character argument in gDisplayStr(). The default escape character is *.

The escape character may be set to any non-control character (ie. Ascii codes 32-255) however if that character has special significance to gDisplayStr(), that gDisplayStr() function will then not be available. i.e. if the escape character is changed to the character F, a change of font will not be possible within the gDisplayStr() routine.

See Also Page 158
gDisplayStr

gSetFacetFillStyle

Syntax

C/C++: **void gSetFacetFillStyle**(int fill);

F90: **subroutine gSetFacetFillStyle**(fill)
integer, intent(in) :: fill

Arguments ***fill***
Facet fill style

= GHOLLOW,	Draw boundary only
= GSOLID,	Solid fill (default)

Description The routine gSetFacetFillStyle() sets the current facet filling mode.

Note that facet boundaries, on their own are NOT subject to hidden surface removal. In other words, when boundaries are drawn it is only the lines that are hidden by closer objects, not the surface they represent.

See Also Page 301
gDrawFacet

gSetFacetMaterialProps

Syntax

C/C++:	**void gSetFacetMaterialProps**(int face, int amb, int diff, int spec, int emit, float shine, float trans);

F90:	**subroutine gSetFacetMaterialProps**(face, amb, diff, spec, emit, shine, trans) integer, intent(in) :: face,amb,diff,spec,emit real, intent(in) :: shine,trans

Arguments

face
Facet face

= GFRONT,	Sets material properties for front face
= GBACK,	Sets material properties for back face

amb
Ambient reflection colour

diff
Diffuse reflection colour

spec
Specular reflection colour

emit
Emission colour

shine
Specular concentration (shininess) as percentage

trans
Translucence (filtering) (0.0-1.0)

Description

The routine gSetFacetMaterialProps() sets the current facet material properties for either the front or back faces of all subsequently drawn facets (or objects). This overrides any material properties set using the material table using gSetMaterialIndex().

The parameters **amb**, **diff**, **spec** and **emit** are all integer values that may be indices into the GINO colour table or 24-bit RGB true colour values as returned by the function gTrueCol(). If any of the particular properties are not required the colour black (colour index 1 - GBLACK) should be used.

Translucence values less than 1.0 (opaque) are only utilized if surface blending is switched on by the gSetShadingMode() routine.

See Also

Page 342
gDrawFacet
gSetMaterialIndex
gSetShadingMode
gTrueCol

gSetFacetOffsetMode

Syntax

C/C++:	**void gSetFacetOffsetMode**(int mode);

F90:	**subroutine gSetFacetOffsetMode**(mode) integer, intent(in) :: mode

Arguments

mode

Facet offset mode

= GOFF,	Switch off all facet offsets
= GBOUNDARYAWAY,	Shift boundary away from view point
= GBOUNDARYNEAR,	Shift boundary towards the view point
= GINTERIORAWAY,	Shift interior away from view point
= GINTERIORNEAR,	Shift interior towards the view point

Description

The routine gSetFacetOffsetMode() sets an offset that facet interiors or boundaries are drawn subject to the current view. When set to a value other than GOFF, the facet interior or the boundary is shifted by a nominal amount nearer to or further from the view point. This affects the objects visibility when depth buffering is used, in that the facet will have different depth values to others drawn at the same distance from the viewer.

This routine has particular relevance when drawing facet interiors and boundaries (or other surface detail) because ordinarily, whichever is drawn second is partially hidden by whichever has been drawn first by the hidden surface removal software.

See Also

Page 302
gDrawFacet
gSetDepthMode

gSetFillMode

Syntax

C/C++:	**void gSetFillMode**(int sw);

F90:	**subroutine gSetFillMode**(sw) integer, intent(in) :: sw

Arguments ***sw***
Fill switch

= GHARD,	Fill areas using hardware if possible (default)
= GSOFT,	Area fill using software

Description Some devices have limitations on hardware area filling. However, GINO software filling is device independent. A call to gSetFillMode(GSOFT) will ensure that all filling is performed using software fill.

See Also Page 171

gSetFontFillStyle

Syntax

C/C++:	**void gSetFontFillStyle**(GFNTFILSTY *style);

F90:	**subroutine gSetFontFillStyle**(style) type (GFNTFILSTY), intent(in) :: style

Arguments ***style.type***
font style type

= GOUTLINE,	Outline only
= GFILLED,	Filled font using foreground and background fill and line styles
= GOUTFILL,	Filled font and outline
> 2,	Hardware dependent

style.ffill
Font foreground fill style

= GNOFILL,	No foreground filling
= GSOLID,	Solid fill
= 1 - 256,	Fill style index (hardware fill) or hatch style index (software fill)
> 256,	Fill style index (hardware fill) or solid fill (software fill)

style.fline
Font foreground line style

= GCURRENT,	Current line style
= 1 - 256,	Line style index
> 256,	Current line style

style.bfill
Font background fill style

= GNOFILL,	No background filling
= GSOLID,	Solid fill

= 1 - 256	Fill style index (hardware fill) or hatch style index (software fill)
> 256,	Fill style index (hardware fill) or solid fill (software fill)

style.bline
Font background line style

= GCURRENT,	Current line style
= 1 - 256,	Line style index
> 256,	Current line style

Description

The routine gSetFontFillStyle() sets the style for hardware and polygon fonts.

For hardware fonts, **style.type** sets the hardware font style. On most devices only solid fonts are available in which case the filled font will be displayed in the current colour and this routine will have no effect. However, on some devices a number of styles are available and **style.type** may be used to select from those available. It is unlikely that the fill and line styles will be used for hardware fonts.

For software fonts that are defined as polygon fonts **style.type** sets the style of the font. When **style.type** = GOUTLINE or GOUTFILL the outline is drawn in the current colour. The foreground and background fill and line styles are not used for **style.type** = GOUTLINE.

For fill styles GFILLED and GOUTFILL the foreground and background fill style can be independently selected by setting the fill style to GNOFILL if it is not required. The fill style and line styles are set using gDefineHatchStyle() and gDefineLineStyle() respectively. Filling may be done in hardware or software depending on the capabilities of the device and the setting of gSetFillMode().

See Also

Page 144
gSetCharFont
gSetFillMode
gDefineHatchStyle
gDefineLineStyle

gSetFontForm

Syntax

C/C++:	**void gSetFontForm**(int rep);

F90:	**subroutine gSetFontForm**(rep) integer, intent(in) :: rep

Arguments

rep
Software font representation

= 0,	Normal font representation (default)
= 1,	Display as font 0
= 2,	Display as boxes (type 1)
= 3,	Display as font 0 and boxes (type 1)
= 4,	Display as boxes (type 2)
= 5,	Display as font 0 and boxes (type 2)

= 6,	Display as boxes (type 3)
= 7,	Display as font 0 and boxes (type 3)

Description

The routine gSetFontForm() can be used to speed up the display of software fonts by representing the output string using font 0 and/or a series of boxes showing the size of the characters in the actual font.

Box type 1 is drawn with the width and height equal to the width and height of the character alone.

Box type 2 is drawn with the width and height equal to the width and height of the full character box including left and right bearings, and space above and below for underlining.

Box type 3 is drawn with the width equal to the width of the character and the height equal to the space occupied by the full character box height.

Setting **rep** > 0 has the additional effect of characters always being drawn with the default pen width, thus ignoring any setting of gSetFontWeight().

gSetFontForm() does not affect the default font 0 and hardware, Greek and symbol fonts are not replaced by font 0 for odd values of **rep**.

If **rep** is outside the above range, a warning message is output and the default value of zero is used.

See Also

Page 146
gSetFontWeight

gSetFontSpacing

Syntax

C/C++: **void gSetFontSpacing**(int space);

F90: **subroutine gSetFontSpacing**(space)
integer, intent(in) :: space

Arguments

space
Font spacing

= GNORMAL,	Normal font spacing (default)
= GFIXEDPITCH,	Force equal spacing

Description

The routine gSetFontSpacing() can be used to force proportional fonts to be output as non-proportional or fixed pitch fonts.

Where hardware proportional fonts are being used, and equal spacing is requested, the character box for each character is adjusted so that each one is of equal size. Each character is positioned separately on the base line but in the centre of the new box.

See Also

Page 146

gSetFontWeight

Syntax

C/C++:	**void gSetFontWeight**(int weight);

F90:	**subroutine gSetFontWeight**(weight) integer, intent(in) :: weight

Arguments

weight
Font weight

< 0,	Font thinning factor
= 0,	Normal font weight (default)
> 0,	Font weighting factor

Description

The routine gSetFontWeight() sets the font weight for subsequent character output. For software fonts, the width of vectors used to display the font or its boundary will be affected but not the interior area of filled fonts. Its effect on hardware fonts is device dependent.

Positive weighting factors increase the width of vectors above the default pen width, whereas negative weighting factors decrease the width (if vectors can be drawn thinner than the default width). The increase/decrease factor is proportional to the character width thus maintaining similar effects for each factor.

The following are suggested values for the standard font weights:

weight	Description
-6	Extra Thin
-3	Thin
0	Normal
+3	Bold
+6	Extra Bold

gSetFontWeight() does not affect software transformed characters as these are drawn using the current line style which included a thickness attribute. gSetFontWeight() is ignored on software fonts if non default font representations are set with gSetFontForm().

See Also

Page 145
gSetFontForm

gSetGraphicsVis

Syntax

C/C++:	**void gSetGraphicsVis**(int vis);

F90:	**subroutine gSetGraphicsVis**(vis) integer, intent(in) :: vis

Arguments ***vis***
Graphics area visibility

= GINVISIBLE,	Graphics area invisible
= GVISIBLE,	Graphics area visible (default)

Description The routine gSetGraphicsVis() sets the graphics area to be visible or invisible. Making it invisible only switches it off, it does not delete it.

This routine will only function on screens with separate graphics/dialogue planes (see Appendix B).

See Also Page 51
gSetDialogueVis
Appendix B

gSetHardChars

Syntax

C/C++: **void gSetHardChars**(void);

F90: **subroutine gSetHardChars**

Arguments None

Description The routine gSetHardChars() switches the current character mode to hardware character mode. This mode takes effect when characters are subsequently output provided there has been no intervening call to gSetSoftChars() or gSetCharTransformMode(GON).

When hardware character mode is active, GINO tries to output characters using whatever character-generation facilities the device might have. If the device cannot generate characters within the limits required by gSetMixedChars(), non-italicized character-generation is requested, and if this fails, GINO will search for the nearest size of characters that the device can generate. If the device can generate no characters at all, they will be generated by GINO just as if gSetSoftChars() was called. Appendix B specifies what character generation facilities are provided by the device.

See Also Page 162
gSetCharTransformMode
gSetMixedChars
gSetSoftChars
Appendix B

gSetHardCharSize

Syntax

C/C++: **void gSetHardCharSize**(int nsize, int nhv);

F90: **subroutine gSetHardCharSize**(nsize, nhv)
integer, intent(in) :: nsize,nhv

Arguments

nsize
Hardware character size index

= 0,	Default character size

nhv
Hardware character orientation

= GHORIZ,	Horizontal (0 degrees)
= GVERT,	Vertical (90 degrees)

Description

The routine gSetHardCharSize() is used to select hardware characters on older devices that only had a preset number of sizes available. It selects a hardware character size and orientation according to the values in **nsize** and **nhv**. If **nsize** is less than zero, a warning message is output and the absolute value of **nsize** is used to determine the character size.

The character size should be converted by the device driver into an actual character width and height. This can subsequently be enquired by calling gEnqCharAttribs(). Even if the device cannot generate characters as requested, the device driver may still carry out the conversion. GINO will then generate characters with a size suited to the device (pseudo-hardware character generation). If no response is obtained from the device driver, the current character size is set to the default character size. Available hardware character sizes for each device are given in Appendix B.

A call to gSetHardCharSize() overrides the combined effect of any calls to gSetHardChars(), gSetMixedChars(), gSetSoftChars(), gSetCharTransformMode(), gSetCharSize(), gSetStrAngle() or gSetItalicAngle(). gSetHardCharSize() will remain in effect until any one of these routines is called.

See Also

Page 138
gSetStrAngle
gEnqCharAttribs
gSetCharSize
gSetCharTransformMode
gSetHardChars
gSetItalicAngle
gSetMixedChars
gSetSoftChars
Appendix B

gSetInterlineSpace

Syntax

C/C++:	**void gSetInterlineSpace**(float drpfac);

F90:	**subroutine gSetInterlineSpace**(drpfac) real, intent(in) :: drpfac

Arguments

drpfac
Inter-line spacing factor

Description The routine gSetInterlineSpace() sets the current inter-line spacing factor for text blocks. Text blocks are positioned using gStartTextBlock(), and new lines are started by either calling gMoveToNextLine() or using the *N escape sequence within any of the string output routines.

The argument **drpfac** is a factor of the current character height (set by gSetCharSize()) by which each line of a text block is placed below the preceding line. The default **drpfac** of 2.0 sets each line to be positioned 2.0 * character height below the preceding line.

drpfac can be any real value, positive, negative or zero with the corresponding effect on the line spacing.

See Also Page 154
gStartTextBlock
gMoveToNextLine
gSetCharSize
gDisplayStr

gSetItalicAngle

Syntax

C/C++:	**void gSetItalicAngle**(float slant);

F90:	**subroutine gSetItalicAngle**(slant) real, intent(in) :: slant

Arguments

slant
Italic angle in degrees

Description

The routine gSetItalicAngle() sets the current italic angle to the value in **slant**. The italic angle is reduced to an equivalent angle between -90.0 and +90.0 degrees. The maximum possible magnitude of the slant is 85.0 degrees. The italic angle is measured in a clockwise direction from the character vertical.

What effect a call to gSetItalicAngle() has when characters are generated depends on the character mode at that time:

(a) gSetHardCharSize()

Call to gSetItalicAngle() has no effect

(b) gSetHardChars()

Characters may be non-italicized if not supported by the device

(c) gSetMixedChars()

Italic angle may differ by up to 5 degrees from the requested angle

(d) gSetSoftChars()

Characters are italicized exactly as requested

(e) gSetCharTransformMode(GON)

Characters are italicized as requested and transformed, so angle may be modified by current shear.

See Also

Page 150
gSetHardCharSize
gSetCharTransformMode
gSetHardChars
gSetMixedChars
gSetSoftChars

gSetLightSwitch

Syntax

C/C++:	**void gSetLightSwitch**(int light, int sw);

F90:	**subroutine gSetLightSwitch**(light, sw) integer, intent(in) :: light, sw

Arguments

light
Light source number (1-8)

sw
Light switch

= GOFF,	Off (default)
= GON,	On

Description

The routine gSetLightSwitch() switches the specified light source on or off. By default all light sources are switched off.

The type and specification of a light source is defined using gDefineLightSource().

See Also

Page 332
gDefineLightSource
gEnqLightAttribs

gSetLineColour

Syntax

C/C++:	**void gSetLineColour**(int col);

F90:	**subroutine gSetLineColour**(col) integer, intent(in) :: col

Arguments

col
Colour setting

= 0,	GBACKGROUND
= 1,	GBLACK
= 2,	GRED

= 3,	GORANGE
= 4,	GYELLOW
= 5,	GGREEN
= 6,	GCYAN
= 7,	GBLUE
= 8,	GMAGENTA
= 9,	GBROWN
= 10,	GWHITE
> 10,	Index up to device colour capability

Description

The routine gSetLineColour() selects a colour for subsequent graphical output. The argument **col** can be an index in the current GINO colour table or (on a true colour device) a 24bit RGB triplet returned by the function gTrueCol().

The colour index values/constants described above represent the initial setting of the GINO colour table, which may however be modified using one of the colour definition routines gDefineRGB(), gDefineHSV() or gDefineHLS(). Whether a colour actually appears on a device depends on the device's hardware capabilities.

The background colour is selected by setting **col**=GBACKGROUND. Graphical output may be erased from the picture by redrawing it with the background colour, provided that this is supported by the device.

If a negative value is passed in **col**, a warning message is output and the absolute value of **col** is used. When **col** is outside the range of colours of the device, a default colour is selected. A call to gEnqSelectedPen() returns the colour actually provided.

See Also

Page 117
gDefineHLS
gDefineHSV
gEnqSelectedPen
gDefineRGB
gTrueCol

gSetLineEnd

Syntax

C/C++:	**void gSetLineEnd**(int end);

F90:	**subroutine gSetLineEnd**(end) integer, intent(in) :: end

Arguments

end

Line end type

= GNONE,	No ends (default)
= GSQUARE,	Square ends
= GROUND,	Round ends
> 2,	No ends or device dependent ends

Description The routine gSetLineEnd() specifies a line end type for subsequent graphical output. The absolute value of **end** defines the end type. If **end** is less than zero, a warning message is output. End types greater than 2 will default to no ends unless there are other end types supported by the device (detailed in Appendix B). Round ends ensure that there is a smooth join between thick lines.

See Also Page 120

gSetLineStyle

Syntax

C/C++:	**void gSetLineStyle**(int line);

F90:	**subroutine gSetLineStyle**(line) integer, intent(in) :: line

Arguments

line
Line style index

= 0,	Current line style
= 1 - 256,	Stored line style

Description The routine gSetLineStyle() sets the current line attributes to the values defined by the specified line style. A call to gSetLineStyle() is equivalent to calling gSetLineVis(), gSetBrokenLine(), gSetLineColour(), gSetLineWidth(), gSetPenType() and gSetLineEnd().

If **line** is zero, gSetLineStyle() does nothing. If **line** is out of range, an error message is output and no further action is taken.

See Also Page 131
gSetBrokenLine
gSetLineColour
gDefineLineStyle
gSetLineEnd
gSaveLineStyle
gSetLineVis
gSetLineWidth
gSetPenType

gSetLineVis

Syntax

C/C++:	**void gSetLineVis**(int vis);

F90:	**subroutine gSetLineVis**(vis) integer, intent(in) :: vis

Arguments

vis
Line visibility

= GINVISIBLE,	Line invisible
= GVISIBLE,	Line visible (default)

Description Normally lines are generated so that they are visible. No lines will appear on the device when **vis** is set to GINVISIBLE with a call to gSetLineVis(). gSetLineVis() does not affect any of the invisible drawing routines (e.g. gMoveTo2D()).

See Also Page 116

gSetLineWidth

Syntax

C/C++: **void gSetLineWidth**(float width);

F90: **subroutine gSetLineWidth**(width)
real, intent(in) :: width

Arguments

width
Line width in current units

Description

The routine gSetLineWidth() specifies a line width for subsequent graphical output.

When **width** is set to zero (0.0) the line width is set to the default width for the currently nominated device. This would be equivalent to a single pixel's width on raster devices or the default pen width on plotters. In some cases a smaller line width is available in which case a small positive value should be used to select the finest line width available on any device.

Where only discrete line/pen widths are possible, the nearest available width is selected and a call to gEnqSelectedPen() will return the line width actually used.

If **width** is less than zero, a warning message is output. The absolute value of **width** is taken to be the line width.

Line width only affects characters if gSetSoftChars() has been called and gSetFontWeight() has not been called.

See Also

Page 119
gEnqSelectedPen

gSetLineWidthMode

Syntax

C/C++: **void gSetLineWidthMode**(int sw);

F90: **subroutine gSetLineWidthMode**(sw)
integer, intent(in) :: sw

Arguments

sw
Thick line generation mode

= GHARDWARE,	Set hardware thick line generation mode

= GMIXWARE,	Set mixed hardware and software generation mode
= GSOFTWARE,	Set software thick line generation mode

Description

The routine gSetLineWidthMode() sets the thick line generation mode for lines greater than one device unit wide as set by gSetLineWidth() or gDefineLineStyle().

Where hardware generation mode is selected, GINO only uses hardware line generation facilities available on the current output device. Where hardware thick generation facilities exist, thick broken lines with non-standard line ends may be generated incorrectly and line ends may extend into GINO masks or out of GINO window limits. If the device does not have hardware thick line generation facilities, no thick lines will be generated and all lines will be the default thickness.

The default mode uses a mixture of hardware and software generation depending on the capabilities of the output device and what sort of thick line is being output. That is, where a device can generate hardware thick lines, these are used for all lines except thick broken lines with non-standard line ends. (This is to ensure the line end is placed only on the end of each line rather than on the end of each broken line segment.) Where a device has no hardware thick line facilities all thick lines will be generated by software (see below). The default mode therefore uses the most efficient mode to generate the output requested but which may not be 100% accurate.

Selecting software generation will force thick lines to be generated using an appropriate emulation method for the current device driver. These include drawing lines parallel to the base line (appropriate to pen plotters), generating a series of horizontal/vertical lines (appropriate to raster displays) and using polygon fill where such facilities exist. The method selected by the device driver may be enquired through the routine gEnqDeviceState(). This mode is guaranteed to generate 100% accurate broken and thick lines.

See Also

Page 119
gEnqDeviceState
gDefineLineStyle
gSetLineWidth

gSetLineWidthScaling

Syntax

C/C++:	**void gSetLineWidthScaling**(float scale);

F90:	**subroutine gSetLineWidthScaling**(scale) real, intent(in) :: scale

Arguments

scale
Line width scale factor

Description

The routine gSetLineWidthScaling() sets a scale factor to be used for all future settings of line width. These include gSetLineWidth() and internal settings for character underlining and font weight.

If **scale** is less than or equal to 0.0 a warning message is output and the scale factor is set to 1.0.

See Also Page 119
gSetLineWidth

gSetMask2D

Syntax

C/C++:	**void gSetMask2D**(GLIMIT *limit);

F90:	**subroutine gSetMask2D**(limit) type (GLIMIT), intent(in) :: limit

Arguments

limit
Extent of mask limits in picture coordinates

Description

The routine gSetMask2D() defines the limits of a 2-D rectangular mask. The limits are in picture coordinates and are clipped to the current device limits. They are not transformable.

This routine also switches the state of masking on if the limits are within the device limits.

See Also

Page 224
gSetMaskMode

gSetMaskMode

Syntax

C/C++:	**void gSetMaskMode**(int sw);

F90:	**subroutine gSetMaskMode**(sw) integer, intent(in) :: sw

Arguments

sw
Masking switch

= GOFF,	Switches masking off
= GON,	Switches masking on

Description

Once a mask has been established using gSetMask2D() or gSetPolygonMask(), it may be switched on or off using gSetMaskMode().

While masking is switched on no graphics output will appear within the mask area.

See Also

Page 225
gSetMask2D

gSetMaterialColour

Syntax

C/C++:	**void gSetMaterialColour**(int fcol, int bcol);

F90:	**subroutine gSetMaterialColour**(fcol, bcol) integer, intent(in) :: fcol, bcol

Arguments

fcol
Front face material colour

bcol
Back face material colour

Description

The routine gSetMaterialColour() sets the current material colour for the front and back faces of facets. Both values may be indices into the GINO colour table or 24-bit RGB true colour values returned from the function gTrueCol().

The material colour is multiplied by the current material table lighting coefficients (as set by gSetMaterialIndex()) to calculate the actual material settings of either face.

Setting the facet material properties using gSetFacetMaterialProps() overrides any values set by this routine and gSetMaterialIndex().

See Also

Page 341
gSetFacetMaterialProps
gSetMaterialIndex
gTrueCol

gSetMaterialIndex

Syntax

C/C++:	**void gSetMaterialIndex**(int fmat, int bmat);

F90:	**subroutine gSetMaterialIndex**(fmat, bmat) integer, intent(in) :: fmat, bmat

Arguments

fmat
Front face material index

< 0,	No change
= GOFF,	No front face material (see below)
= 1-256,	Set front face material coefficients from material table

bmat
Back face material index

< 0,	No change

= GOFF,	No back face material (see below)
= 1-256,	Set back face material coefficients from material table

Description

The routine gSetMaterialIndex() sets the current facet material coefficients from the material table for both the front and back faces of facets. These are multiplied by the current material colour settings (as set by gSetMaterialColour()) to give the actual material settings of either face.

Entries in the material table are set through the routine gDefineMaterial().

When the back face material index is set to GOFF, no lighting calculations are performed on back facing faces. When both the front and back face material indices are set to GOFF, facet colours are switched to be set by the current line drawing colour (gSetLineColour()) and all material information is ignored.

Setting the facet material properties using gSetFacetMaterialProps() overrides any values set by this routine and gSetMaterialColour().

See Also

Page 341
gDefineMaterial
gSetLineColour
gSetFacetMaterialProps
gSetMaterialColour

gSetMaxErrorLimit

Syntax

C/C++: **void gSetMaxErrorLimit**(int nerrs);

F90: **subroutine gSetMaxErrorLimit**(nerrs)
integer, intent(in) :: nerrs

Arguments

nerrs

The maximum number of GINO errors allowed before the program is stopped (default 10)

= -1,	No limit set
= 0,	Any subsequent error stops the program

Description

GINO keeps a count of the number of errors generated. If this count exceeds the limit set by gSetMaxErrorLimit(), the program is stopped. Whenever GINO is initialized, the limit is set to 10, i.e. the 11th error stops the program. If **nerrs** is set to -1, no limit is set on the number of errors. The error count is reset to 0 whenever gSetMaxErrorLimit() is called.

Note: GINO does not count warnings against the error limit.

See Also

Page 30

gSetMixedChars

Syntax

C/C++:	**void gSetMixedChars**(void);

F90:	**subroutine gSetMixedChars**

Arguments

None

Description

The routine gSetMixedChars() switches the current character mode to hardware/software character mode. This mode takes effect when characters are subsequently output provided there has been no intervening call to gSetHardChars(), gSetSoftChars() or gSetCharTransformMode(GON). It is the default character mode.

When hardware/software character mode is active, GINO will try to use the device's character generation facilities. Characters will be generated by the device if they differ by less than 10% of the requested size, by less than 1 degree from the requested orientation and by less than 5 degrees from the requested italic angle. Otherwise they will be generated by GINO just as if gSetSoftChars() was called. Appendix B specifies what character generation facilities are provided by the device.

See Also

Page 162
gSetHardChars
gSetCharTransformMode
gSetSoftChars
Appendix B

gSetMousePos

Syntax

C/C++:	**void gSetMousePos**(int env, int xpos, int ypos);

F90:	**subroutine gSetMousePos**(env, xpos, ypos) integer, intent(in) :: env,xpos,ypos

Arguments

env

Mouse position environment

GSCREEN,	Relative to screen or display area
GDRAWINGAREA,	Relative to current window or drawing area

xpos,ypos

Mouse position in pixels

Description

A call to gSetMousePos() moves the graphics pointer or mouse to the position specified in **xpos**, **ypos**. The actual position is measured in pixels and is relative to the top left corner of the specified environment set in **env**.

Note that it is considered bad practice to continually move the mouse position in an interactive graphics application, so this routine should only be used when assisting the user to take some action by positioning the pointer over an item to be selected.

The routine gEnqPixelPos() can be used to translate a picture position on the current window or drawing area into pixels and the routine gEnqMousePos() can be used to enquire the current mouse position.

The routine gSetMousePos() differs from gSetCursorPos() in that the movement takes place immediately and is not dependent on any action or event that has or is about to take place.

See Also

Page 456
gSetCursorPos
gEnqMousePos
gEnqPixelPos
Appendix B

gSetPenType

Syntax

C/C++:	**void gSetPenType**(int type);

F90:	**subroutine gSetPenType**(type) integer, intent(in) :: type

Arguments

type
Pen type

= GDEFAULT,	Undefined
= GERASER,	Eraser
= GNOT,	NOT mode
= GAND,	AND mode
= GOR,	OR mode
= GXOR,	XOR mode
> 10,	Device dependent

Description

The routine gSetPenType() specifies a pen type or drawing mode for subsequent graphic output. The absolute value of **type** defines the pen type.

If **type** is less than zero, a warning message is output. When the requested pen type is not available, or if **type**=GDEFAULT, selection is left to the device.

A call to gEnqSelectedPen() will identify the pen type actually selected, which will be zero if the device does not implement pen types.

See Also

Page 120
gEnqSelectedPen

gSetPixelDisplayMode

Syntax

C/C++:	**void gSetPixelDisplayMode**(int mode);

F90:	**subroutine gSetPixelDisplayMode**(mode) integer, intent(in) :: mode

Arguments

mode

Pixel display mode

= GOFF,	Visibility off
= GON,	Pixel information displayed (default)
= GBOUNDARY,	Boundary box drawn

Description

The routine gSetPixelDisplayMode() enables a quick display facility for pixel information. For **mode**=GOFF and **mode**=GBOUNDARY the display of the pixel information is suppressed but all pixel transformations and windowing is calculated, producing warning and error messages if applicable.

See Also

Page 197
gDrawPixelArea

gSetPixelReplication

Syntax

C/C++:	**void gSetPixelReplication**(int xrep, int yrep);

F90:	**subroutine gSetPixelReplication**(xrep, yrep) integer, intent(in) :: xrep,yrep

Arguments

xrep,yrep

Direction and number of pixels to be replicated

Description

The routine gSetPixelReplication() specifies the replication area for a subsequent pixel array. The values **xrep**, **yrep** represent the dimensions of a rectangular area into which any pixel output from the routine gDrawPixelArea() is replicated (or clipped) so that the area is completely filled.

Either or both of the dimensions may be negative, in which case the output and replication is done in the negative direction from the pixel area origin for that dimension.

If either **xrep** or **yrep** is zero then replication will be switched off.

See Also

Page 201
gDrawPixelArea

gSetPixelTransform

Syntax

C/C++:	**void gSetPixelTransform**(int ori, float xsca, float ysca);

F90:	**subroutine gSetPixelTransform**(ori, xsca, ysca) integer, intent(in) :: ori real, intent(in) :: xsca,ysca

Arguments

ori
An integer specifying the orientation of the pixel array

= 0,	None
= 1,	90 degrees anti-clockwise
= 2,	180 degrees anti-clockwise
= 3,	270 degrees anti-clockwise

xsca,ysca
X and Y scaling factors

Description

The routine gSetPixelTransform() specifies the orientation and scaling transformation of subsequent pixel rectangles.

All subsequently drawn pixel rectangles will be subject to these transformations. The orientation and scaling transformations occur independently from the pixel coordinate space with the transformed pixel rectangle being drawn from the anchor point specified through gDrawPixelArea().

The orientation variable, **ori**, will affect a rectangular pixel array for 90 and 270 degree rotation by swapping the X and Y dimensions.

The X and Y scaling factors, **xsca**,**ysca,** scale along the X and Y pixel coordinate axes, independently from the orientation characteristics.

See Also

Page 199
gDrawPixelArea

gSetPointMode

Syntax

C/C++:	**void gSetPointMode**(int switch);

F90:	**subroutine gSetPointMode**(switch) integer, intent(in) ::switch

Arguments

switch
Point storage mode

= GOFF,	Switch point storing off
= GSPACE,	Store points in space coordinates
= GPICTURE,	Store points in picture coordinates
= GCLEAR,	Clear current point storage workspace

Description The routine gSetPointMode() sets the current point storage mode. When set to either GSPACE or GPICTURE, the vertices of all subsequent 2D and 3D drawing routines are stored in the point storage workspace. These include all internal vertices generated from the drawing of arcs, curves and software characters according to the respective tolerance and/or tension. Points are stored in either space (untransformed) or picture (transformed) mode according to the setting of **switch**. Note that all arcs are drawn in software when point storage is switched on.

The point storage workspace is emptied if GCLEAR is used or a different mode is set.

Before using this routine, both the global workspace and point storage workspace must be defined using the routines gSetWorkspaceLimit() and gDefinePointWorkspace().

Points can be returned to the application in either 2D or 3D using the functions gReturnInternalPoints2D() or gReturnInternalPoints3D(). In the former case all Z-coordinates are ignored.

See Also Page 104, 291
gDefinePointWorkspace
gReturnInternalPoints2D
gReturnInternalPoints3D
gSetWorkspaceLimit

gSetPolygonIdent

Syntax

C/C++: **void gSetPolygonIdent**(int ident);

F90: **subroutine gSetPolygonIdent**(ident)
integer, intent(in) :: ident

Arguments ***ident***
Current polygon identifier

Description The routine gSetPolygonIdent() defines the current polygon identifier which is the number assigned to a polygon when it is closed. The default current polygon identifier is 0.

See Also Page 249

gSetPolygonMask

Syntax

C/C++: **void gSetPolygonMask**(int list[], int n);

F90: **subroutine gSetPolygonMask**(list, n)
integer, intent(in) :: list(*),n

Arguments

list
Array of polygon identifiers

n
Number of entries in **list**

Description

The routine gSetPolygonMask() selects a list of polygon identifiers to define a polygon mask. The polygons are defined as moves or lines between calls to gStartPolygon() and gEndPolygon() and the polygons are stored as a series of vertices in picture coordinates. The list is also copied into a workspace and can be enquired with the routine gEnqPolygonMaskList().

Additional temporary workspace is required by gSetPolygonMask() to store the list and a copy of the polygons that are used to generate the polygonal mask. If gSetWorkspaceLimit() has not been called or if there is not enough workspace to set up the polygonal mask the appropriate error message is output and masking is switched off.

If **n**=0, no polygons are selected and masking is also switched off. A call to gSetMask2D() will revert to a rectangular mask.

See Also

Page 265
gSetWorkspaceLimit
gSetMask2D
gStartPolygon
gEndPolygon
gEnqPolygonMaskList

gSetPolygonMode

Syntax

C/C++:	**void gSetPolygonMode**(int sw);

F90:	**subroutine gSetPolygonMode**(sw) integer, intent(in) :: sw

Arguments

sw
Vertices storage switch

= GOFF,	Ignore all vertices for the purposes of defining polygon vertices
= GON,	Store all vertices

Description

The routine gSetPolygonMode() controls two switches that affect the storing of polygon vertices:

(a) When called between gStartPolygon() and gEndPolygon() it controls the local switch which when set to GOFF suspends the storing of vertices. This switch is set to GON when gStartPolygon() is called.

(b) When called outside gStartPolygon/gEndPolygon() it controls the global switch which when set to GOFF disables all definition of polygons. It suppresses the storing of vertices and inhibits the action of gClearPolygonWorkspace(). This switch is initially set to GON.

See Also Page 247
gStartPolygon
gClearPolygonWorkspace
gEndPolygon

gSetPolygonWindow

Syntax

C/C++:	**void gSetPolygonWindow**(int list[], int n);

F90:	**subroutine gSetPolygonWindow**(list, n) integer, intent(in) :: list(*),n

Arguments

list
Array of polygon identifiers

n
Number of entries in **list**

Description

The routine gSetPolygonWindow() selects a list of polygon identifiers to define a polygon window. The polygons are defined as moves or lines between calls to gStartPolygon() and gEndPolygon() and the polygons are stored as a series of vertices in picture coordinates. The list is also copied into a workspace and can be enquired with the routine gEnqPolygonWindowList().

Additional temporary workspace is required by gSetPolygonWindow() to store the list and a copy of the polygons that are used to generate the polygonal window. If gSetWorkspaceLimit() has not been called or if there is not enough workspace to set up the polygonal window the appropriate error message is output and windowing is reverted to device/viewport limits.

If **n**=0, no polygons are selected and windowing is switched off. A call to gSetWindow2D() or gSetWindowMode(GON2D) will revert to a rectangular clipping window.

See Also

Page 264
gSetWorkspaceLimit
gSetWindow2D
gSetWindowMode
gStartPolygon
gEndPolygon
gEnqPolygonWindowList

gSetRandSeed

Syntax

C/C++:	**void gSetRandSeed**(int seed);

F90:	**subroutine gSetRandSeed**(seed) integer, intent(in) :: seed

Arguments ***seed***
Random number seed value

Description The system utility gSetRandSeed() can be used in conjunction with gGetRand() to set a fixed seed value to the pseudo random number generator.

See Also Page 466
gGetRand

gSetSegHit

Syntax

C/C++:	**void gSetSegHit**(int nseg, int sens);

F90:	**subroutine gSetSegHit**(nseg, sens) integer, intent(in) :: nseg,sens

Arguments ***nseg***
Picture segment or segment group number

> 0,	Change sensitivity of segment(s) specified by **nseg**
= -1,	Change sensitivity of all segments
< -1,	Change sensitivity of all segments except those specified by **nseg**

sens
Sensitivity status

= GNONSENSITIVE,	Non hit-sensitive (default)
= GSENSITIVE,	Hit-sensitive

Description The routine gSetSegHit() is used to change the hit-sensitivity of picture segments. Only segments that are hit-sensitive may be selected using events 2 and 3 or gEnqSegHit().

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(GHARDWARE), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also Page 431
gWaitForEvent
gEnqSegHit
gSetSegMode

gSetSegMarkColour

Syntax

C/C++:	**void gSetSegMarkColour**(int col);

F90:	**subroutine gSetSegMarkColour**(col) integer, intent(in) :: col

Arguments

col
Colour index for marked segments

Description

The routine gSetSegMarkColour() sets the colour index which is used by the software emulation of gMarkSeg().

By default gMarkSeg() uses the highest colour index available on the current device.

See Also

Page 431
gMarkSeg

gSetSegMode

Syntax

C/C++:	**void gSetSegMode**(int sw);

F90:	**subroutine gSetSegMode**(sw) integer, intent(in) :: sw

Arguments

sw
Software Display File switch

= GHARDWARE,	Segmentation by hardware (if available)
= GMIXWARE,	Segmentation by hardware (if available) with software backup
= GSOFTWARE,	Segmentation by software emulation

Description

The routine gSetSegMode() sets or changes the segmentation mode. Where an application program is using segment facilities the correct operation of such a program depends on the existence of a Software Display File for storing the segment information. There are not many devices that have such facilities built into their hardware, so in order to make such a program device independent GINO provides software emulation of a segmented display file by storing the segment information either in a scratch file or program memory.

Because of the overheads of storing a Software Display File, the default operation is to rely on hardware segment facilities (**sw**=GHARDWARE). The storing of a Software Display File is activated by setting **sw** to GMIXWARE or GSOFTWARE in the call to gSetSegMode() and the selection of storage in memory or on a disk file is made by gDefineSegWorkspace(). If gDefineSegWorkspace() is not called the display file will be held on a scratch file.

When **sw**=GMIXWARE the display file is stored by GINO and the hardware (if possible) and all segment operations will be performed by the hardware if it is able to do so. If an operation is not successful, GINO will emulate the operation from the Software Display File.

When **sw**=GSOFTWARE the display file is stored by GINO and no segment operations are passed through to the hardware.

GINO's Software Display File is maintained across device nominations, and therefore segments can be used on devices other than those on which they were created.

Software emulation of segment facilities relies on the facility to remove a segment from the display using background erase (ie. use of gSetLineColour(0)).

Unpredictable results can occur when changing segmentation mode while a segment is open. In most cases the change is delayed until the close of that segment.

See Also

Page 424
gDefineSegWorkspace

gSetSegTransform

Syntax

C/C++:	**void gSetSegTransform**(int nseg, float xsca, float ysca, float ang, float xpos, float ypos); **void gSetSegTransform2D**(int nseg, GMAT2D a);

F90:	**subroutine gSetSegTransform**(nseg, xsca, ysca, ang, xpos, ypos) **subroutine gSetSegTransform2D**(nseg,a) integer, intent(in) :: nseg real, intent(in) :: xsca,ysca,ang,xpos,ypos real, intent(in) :: a(6)

Arguments

nseg
Picture segment or segment group number

> 0,	Transform segment(s) specified by segment **nseg**
= -1,	Transform all segments
< -1,	Transform all segments except those specified by **nseg**

xsca
Scaling factor for X coordinates

ysca
Scaling factor for Y coordinates

ang
Angle of rotation about anchor point

xpos,ypos
Coordinates of new anchor point

a
3 x 2 array containing a 2-D segment transformation matrix

Description The routines gSetSegTransform() and gSetSegTransform2D() sets the elements of a 2-D segment transformation matrix either by means of separate components or as a complete 2D matrix to be applied to segment **nseg**.

In the case of gSetSegTransform() the elements are applied in the order scale, rotation, and then translation. The scale and rotation are applied about the segment anchor position. **xpos** and **ypos** are assumed to be in picture coordinates and will not be affected by any modelling transformation that may be current.

In the case of gSetSegTransform2D(), if a unit transformation matrix is set the segment will appear in the original position it was created. The routines gBuildMatrix2D() and gCombineMatrix2D() can be used to build or compose a suitable transformation matrix.

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(GHARDWARE), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also Page 431
gEnqSegTransform

gSetSegVis

Syntax

C/C++:	**void gSetSegVis**(int nseg, int vis);

F90:	**subroutine gSetSegVis**(nseg, vis) integer, intent(in) :: nseg,vis

Arguments

nseg
Picture segment or segment group number

> 0,	Set visibility of segment(s) specified by segment **nseg**
= -1,	Set visibility of all segments
< -1,	Set visibility of all segments except those specified by **nseg**

vis
Visibility status

= GINVISIBLE,	Invisible (not displayed)
= GVISIBLE,	Visible (displayed)

Description The routine gSetSegVis() is called to change the display status of the specified picture segment.

When gSetSegVis(**nseg**,GINVISIBLE) is called, the picture segment ceases to be displayed on the screen, or is removed by redrawing the segment in the background colour, but continues to be held in the display file.

When the software emulation of picture segments is used and **nseg** does not exist an error message is generated. When using this routine in the default hardware segmentation mode, gSetSegMode(GHARDWARE), no error message is generated. However, the device may output a local error message.

Some displays do not permit this segment operation on the currently opened segment.

See Also

Page 430
gSetSegMode

gSetShadingMode

Syntax

C/C++:	**void gSetShadingMode**(int mode, ...);

F90:	**subroutine gSetShadingMode**(mode, gCulling, gBlending, gWinding) integer, intent(in) :: mode integer, optional, intent(in) :: gCulling, gBlending, gWinding

Arguments

mode
Shading mode

= GNONE,	Switch shading off (default)
= GFLAT,	Switch on flat shading (using facet normals)
= GGOURAUD,	Switch on smooth (Gouraud) shading (using vertex normals)
= GPHONG,	Switch on Phong shading (using interpolated normals)

Optional Args.

gCulling
Culling mode

= GOFF,	Two sided lighting (default)
= GBACK,	Ignore back (clockwise) facing polygons
= GFRONT,	Ignore front (anti-clockwise) facing polygons

gBlending
Blending mode

= GOFF,	Disable blending (default)
= GON,	Enable blending

gWinding
Facet winding mode

= GANTICLOCKWISE,	Anti-clockwise winding = front face (default)
= GCLOCKWISE,	Clockwise winding = front face

Description

The routine gSetShadingMode() sets up the lighting and shading mode according to the supplied arguments.

See Also

Page 325
gSetDepthMode

gSetSoftChars

Syntax

C/C++:	**void gSetSoftChars**(void);

F90:	**subroutine gSetSoftChars**

Arguments None

Description The routine gSetSoftChars() sets the current character mode to software character mode. This mode takes effect when characters are subsequently output provided there has been no intervening call to gSetHardChars(), gSetMixedChars or gSetCharTransformMode(GON).

When software character mode is active, all characters are generated by GINO to exactly the size, orientation and italic angle requested. Characters are drawn using vectors of solid lines in the current colour. The thickness of each vector is determined by the current setting of font weight (gSetFontWeight()).

See Also Page 162
gSetHardChars
gSetCharTransformMode
gSetFontWeight

gSetSplineTension

Syntax

C/C++:	**void gSetSplineTension**(float tension);

F90:	**subroutine gSetSplineTension**(tension) real, intent(in) :: tension

Arguments ***tension***
Spline curve tension (default=0.0)

Description The routine gSetSplineTension() controls the tightness of 2D and 3D spline curves.

Values in the range -2 to 10 give reasonable levels of tension control, with figures approaching 10.0 producing a polyline. Values less than zero give a more rounded shape to the curve.

See Also Page 100, 287
gDrawSplineBy2D
gDrawSplineTo2D
gDrawSplineBy3D
gDrawSplineTo3D

gSetStrAngle

Syntax

C/C++:	**void gSetStrAngle**(float angle);

F90:	**subroutine gSetStrAngle**(angle) real, intent(in) :: angle

Arguments

angle
Character orientation in degrees (default = 0.0)

Description

The routine gSetStrAngle() sets the current character orientation to the value in **angle**. The angle is measured in an anticlockwise direction from the positive x-axis.

What effect a call to gSetStrAngle() has when characters are output depends on the character mode at that time:

(a) gSetHardCharSize()

Call to gSetStrAngle() has no effect

(b) gSetHardChars()

Characters are drawn at the correct orientation if possible, otherwise they are stepped

(c) gSetMixedChars()

Character orientation may differ by up to 1 degree from the requested orientation

(d) gSetSoftChars()

Characters are angled exactly as requested

(e) gSetCharTransformMode(GON)

Characters are angled as requested and transformed, so orientation may be modified by the current rotation.

See Also

Page 149
gSetCharTransformMode
gSetHardChars
gSetMixedChars
gSetSoftChars

gSetStrExponent

Syntax

C/C++:	**void gSetStrExponent**(float relcw, float relch, float posexp, float posind);

F90:	**subroutine gSetStrExponent**(relcw, relch, posexp, posind) real, intent(in) :: relcw,relch,posexp,posind

Arguments

relcw
Relative character width of character string exponents and indices

relch
Relative character height of character string exponents and indices

posexp
Relative character height above the baseline at which exponents are drawn

posind
Relative character height below the baseline at which indices are drawn

Description

The routine gSetStrExponent() sets the relative size and position of both string exponents and indices. Exponents and indices are drawn by using the *E and *I escape sequences within any of the character string output routines gDisplayStr(), gDisplayStrPolylineBy2D(), gDisplayStrPolylineTo2D() and gFitCharStr().

A zero value for any of the arguments will return its value to default.

A negative argument will cause a warning message and the absolute value will be used.

See Also

Page 155
gDisplayStr
gDisplayStrPolylineBy2D
gDisplayStrPolylineTo2D
gFitCharStr

gSetStrJustify

Syntax

C/C++:	**void gSetStrJustify**(int jus);

F90:	**subroutine gSetStrJustify**(jus) integer, intent(in) :: jus

Arguments

jus
String justification

= GLEFT,	Left-justified (default)
= GCENTRE,	Centre-justified
= GRIGHT,	Right-justified

Description

The routine gSetStrJustify() sets the current string justification.

Left justification (the default) is where the string is drawn starting from the current position from left to right and the current position is left at the lower right end of the character string.

Centre justified strings are positioned such that the string is centred at the current position. The current position after a centre justified string is left at the bottom centre of the string.

For right justified strings, the string is positioned such that the end is at the current position. The current position after a right justified string is at the bottom left of the character string.

Justification affects all string and numeric output except gDrawMarker() and the effect of gMoveToNextLine().

See Also

Page 153
gMoveToNextLine
gDisplayStr
gDisplayStrPolylineBy2D
gDisplayStrPolylineTo2D
gFitCharStr

gSetStrUnderscore

Syntax

C/C++:	**void gSetStrUnderscore**(int und);

F90:	**subroutine gSetStrUnderscore**(und) integer, intent(in) :: und

Arguments

und

String underscore switch

= GOFF,	Underscore off (default)
= GON,	Underscore on

Description

The routine gSetStrUnderscore() switches underscoring of subsequent strings on or off. Underscoring affects all string output using gDisplayStr(), gDisplayStrPolylineBy2D(), gDisplayStrPolylineTo2D() and gFitCharStr().

When underscoring is switched on a solid line is drawn by GINO, using an appropriate line width related to the character height. The line is drawn 0.4*character height below the base line of the character string.

If **und** is neither GOFF or GON, a warning message is output and underscoring is switched off.

Underscoring can be temporary switched on and off using the *S and *A escape sequences respectively. These are described under gDisplayStr().

See Also

Page 152
gDisplayStr
gDisplayStrPolylineBy2D
gDisplayStrPolylineTo2D
gFitCharStr

gSetSysPriority

Syntax

C/C++:	**void gSetSysPriority**(int pri);

F90:	**subroutine gSetSysPriority**(pri) integer, intent(in) :: pri

Arguments ***pri***
Task priority

= GREALTIME,	Sets task to highest possible priority
= GHIGH,	Sets task to higher than normal
= GNORMAL,	(default)
= GLOW,	Sets task to lower than normal (not available under Windows)
= GIDLE,	Sets task into idle state

Description The routine gSetSysPriority() sets the priority of the GINO application as described by the above settings. The routine is system dependent and some settings may have no effect in some environments.

Users should be warned that setting a task to GREALTIME will prevent all other applications operating including some system tasks such as responding to mouse movement. This should be used VERY sparingly.

The current system priority can be obtained using the routine gEnqSysPriority().

See Also Page 466
gEnqSysPriority

gSetTextureCoordGeneration

Syntax

C/C++: **void gSetTextureCoordGeneration**(int mode, ...);

F90: **subroutine gSetTextureCoordGeneration**(mode, gSVec, gTVec)

```
integer, intent(in) :: mode
type (GTEXVEC), optional, intent(in) :: gSVec,gTVec
```

Arguments ***mode***
Texture coordinate generation mode

= GOFF,	Switch off texture coordinate generation (default)
= GOBJECT,	Use object coordinates
= GSPHERICAL,	Generate spherical texture coordinates

Optional Args. ***gSVec, gTVec***
Object coordinate transformation vectors for S and T texture coordinates

gSVec.trans, gTVec.trans
Object coordinate type

= GSPACE,	Use untransformed object coordinates
= GPICTURE,	Use transformed object coordinates

gSVec.xfactor, gTVec.xfactor
Scale factor for object's X coordinate

gSVec.yfactor, gTVec.yfactor
Scale factor for object's Y coordinate

gSVec.zfactor, gTVec.zfactor
Scale factor for object's Z coordinate

gSVec.wfactor, gTVec.wfactor
Scale factor for object's W coordinate

Description

The routine gSetTextureCoordGeneration() sets the current GINO texture coordinate generation mode which by default is off.

When the **mode** is set to GOBJECT the optional arguments **gSVec** and/or **gTVec** are used to specify the transformation of the object's coordinates required to generate the S and T texture coordinates respectively. The structure, of type GTEXVEC, contains the object coordinate type to be used and the scale factors to be applied to each of its coordinates.

Thus the texture coordinate at each vertex is calculated as:

S = svec.xfactor * x + svec.yfactor * y + svec.zfactor * z + svec.wfactor * w

and

T = tvec.xfactor * x + tvec.yfactor * y + tvec.zfactor * z + tvec.wfactor * w

where x,y,z,w are the objects' untransformed or transformed coordinates at this vertex.

Note that C/C++ users should pass the address of the structure along with the **gSVec** and/or **gTVec** arguments as shown in the example code in the section referenced below.

The GSPHERICAL mode generates texture coordinates based on a sphere centred around the view centre. It should only be used, however, when a correctly prepared environment texture map is available, representing the image of a scene taken through a very wide-angle lens.

See Also

Page 351
gDefineTexture
gDrawFacet

gSetTextureMappingMode

Syntax

C/C++:	**void gSetTextureMappingMode**(int mode, ...);

F90:	**subroutine gSetTextureMappingMode**(mode, gBlendCol, gWraps, gWrapt, gMaxfil, gMinfil, gBorderCol) integer, intent(in) :: mode integer, optional, intent(in) :: gBendCol,gWraps,gWrapt,gMaxfil,gMinfil,gBorderCol

Arguments

mode
Texture mapping mode

= GOFF, Switch off texture mapping (default)

= GOVERLAY,	Overlay texture on surface
= GMODULATE,	Modulate texture colours with facet colour
= GBLEND,	Blend texture colours with facet colour and a constant colour gBlendCol

Optional Args. ***gBlendCol***
Blend mode colour (default = current background colour)

gWraps
Texture wrapping switch in S direction

= GREPEAT,	Repeat texture map (default)
= GCLAMP,	Clamp texture map

gWrapt
Texture wrapping switch in T direction

= GREPEAT,	Repeat texture map (default)
= GCLAMP,	Clamp texture map

gMaxfil
Filter when enlarging texture map

= GNEAREST,	Use nearest texel (default)
= GLINEAR,	Use weighted average of 2x2 texels

gMinfil
Filter when reducing texture map

= GNEAREST,	Use nearest texel (default)
= GLINEAR,	Use weighted average of 2x2 texels
= GNEARESTNEAREST,	Nearest mipmap using nearest texel filter
= GNEARESTLINEAR,	Nearest mipmap using linear texel filter
= GLINEARNEAREST,	Linear interpolate mipmap using nearest texel filter
= GLINEARLINEAR,	Linear interpolate mipmap and linear texel filter

gBorderCol
Texture map border colour (default = current background colour)

Description The routine gSetTextureMappingMode() sets the current GINO texture mapping mode which by default is off.

When set to GOVERLAY, the current texture assigned by gDefineTexture() is mapped onto every facet or object subsequently displayed. When set to GMODULATE the current texture is merged with the facet or objects own colour. When set to GBLEND the current texture is merged with the facet colour and a constant colour defined in the optional argument **gBlendCol**.

The optional arguments **gWraps** and **gWrapt** define the action to be taken, where supplied or generated texture coordinates extend outside the default range of 0.0 to 1.0 in either the horizontal (S) or vertical (T) direction.

The optional arguments **gMaxfil** and **gMinfil** define the filters used when determining the texture map pixel (texel) to be displayed on the screen when enlarging or reducing the currently defined image. The additional settings for **gMinfil** apply where multiple texture maps (mipmaps) have been assigned using gDefineTexture() with levels greater than zero.

The optional argument **gBorderCol** is used to define a constant border colour, where no such border is attached to the currently defined texture map. This border colour is used in clamped images where the linear texel filtering option is also used.

Note that GOVERLAY mode only operates with textures with 3 or 4 bytes per pixel and GBLEND mode only works with textures with 1 or 2 bytes per pixel.

See Also

Page 345
gDefineTexture

gSetTracerMode

Syntax

C/C++:	**void gSetTracerMode**(int sw);

F90:	**subroutine gSetTracerMode**(sw) integer, intent(in) :: sw

Arguments

sw

Trace switch

= GOFF,	Switch off tracing (default)
= GGINOCALLS,	Switch on tracing for user calls to GINO library
= GLIBRARIES,	Switch on tracing for user calls to GINO library and GINO application libraries
= GALLCALLS,	Switch on tracing for all calls to GINO library and GINO application libraries

Description

When **sw** is not GOFF, gSetTracerMode() switches the tracing facility on. With tracing switched on GINO outputs a trace message which identifies routines from either the GINO library or GINO application libraries as they are called.

When **sw** = GGINOCALLS all calls to the GINO library are traced. When **sw** = GLIBRARIES user calls to GINO or GINO application libraries are traced, but internal calls to GINO by the application library are not traced. When **sw** = GALLCALLS all user calls and internal calls to GINO and any GINO application libraries used are traced.

See Also

Page 31, 37

gSetTransform

Syntax

C/C++:
void gSetTransform(int sw);
void gSetTransform2D(GMAT2D a2);
void gSetTransform3D(GMAT3D a3);

F90:
subroutine gSetTransform(sw)
subroutine gSetTransform2D(a2)
subroutine gSetTransform3D(a3)

integer, intent(in) :: sw
real, intent(in) :: a2(6),a3(16)

Arguments

sw
Transformation switch

= GRESET,	Initialize the current modelling transformation, initialize the current viewing transformation, initialize the viewing parameters (see gInitView()) and switch transforming off
= GOFF,	Switch transforming off
= GON,	Switch transforming on
= GINIT,	Initialize the current modelling transformation and switch transforming on

a2,a3
2D or 3D transformation matrix

Description

The gSetTransform set of routines set the current modelling transformation and/or switch transforming on and off.

The routine gSetTransform() is used to either initialize the modelling transformation matrix (and the viewing transformation matrix if GRESET is used) to unity and/or switch transforming on or off without affecting the current modelling transformation matrix.

The routines gSetTransform2D(), gSetTransform3D() set the current modelling transformation according to the supplied 2D or 3D matrix and switches transforming on. The supplied matrix can either have been saved from a previous call to gGetTransform(), or composed using the gBuildMatrix() or gCombineMatrix() routines. In the case of gSetTransform2D() the remaining ten 3-D components of the complete transformation matrix are reinitialized.

See Also

Page 240, 372
gBuildMatrix
gCombineMatrix
gGetTransform
gInitView

gSetTransformMode

Syntax

C/C++:	**void gSetTransformMode**(int sw);

F90:	**subroutine gSetTransformMode**(sw) integer, intent(in) ::sw

Arguments

sw
Picture mode switch

= GSPACE,	Switch modelling transforming into space mode
= GPICTURE,	Switch modelling transforming into picture mode

Description

When modelling transforming is in space mode (the default) each modelling transformation is specified with respect to the most recent set of space axes (sometimes called current or local axes). Subsequent changes to the transformation matrix are post-multiplied.

When in picture mode each modelling transformation is specified with respect to the picture (or screen) axes. Subsequent changes to the transformation matrix are pre-multiplied.

The picture axes never change and have their origin at the bottom left-hand corner of the picture, with X horizontal and to the right, Y vertical, positive upwards, and the Z axis perpendicular to the plane of the picture, in the direction of the user.

See Also

Page 239, 369, 382

gSetView

Syntax

C/C++:	**void gSetViewParams**(GMATV vdata): **void gSetViewState**(GVIEWSTATE vstate);

F90:	**subroutine gSetViewParam**s(vdata) **subroutine gSetViewState**(vstate) real, intent(in) :: vdata(15) type (GVIEWSTATE), intent(in) :: vstate

Arguments

vdata
Stored viewing array with at least 15 elements

vstate.mode
View mode

= 0,	No view defined
= 1,	Perspective view defined (gDefineSphericalView() or gDefinePerspView())
= 2,	Parallel view defined (gDefineParallelView())

vstate.cflag
View centre flag

= 0,	No view centre defined
= 1,	Default view centre defined
= 2,	User defined view centre (gPosViewCentre())

vstate.upflag
View up direction flag

= 0,	Default view up vector (0.0,1.0,0.0)
= 1,	User defined view up vector (gSetViewUpDirection())

vstate.dir
View direction vector

vstate.centre
View centre

vstate.dist
Perspective viewing distance

vstate.shift
View shift

vstate.upvec
View up direction vector

Description

The routines gSetViewParams() and gSetViewState() set the viewing parameters to the values stored in the appropriate structure. The current settings are obtained through the routines gGetViewParams() or gGetViewState(). The new settings may subsequently be modified by using one of the viewing routines but are only activated after updating the current view using gUpdateView().

See Also

Page 416
gGetViewParams
gGetViewState
gUpdateView

gSetViewAxis

Syntax

C/C++:	**void gSetViewAxis**(int nh, int nv);

F90:	**subroutine gSetViewAxis**(nh, nv) integer, intent(in) :: nh,nv

Arguments

nh
The axis to be horizontal

= GXAXIS,	X axis
= GYAXIS,	Y axis
= GZAXIS,	Z axis

nv
The axis to be vertical

= GXAXIS,	X axis
= GYAXIS,	Y axis
= GZAXIS,	Z axis

Description The routine gSetViewAxis() selects the view so that the two specified axes are horizontal and vertical respectively.

This permutes the rows of the modelling transformation matrix so that the **nh** axis is horizontal and the **nv** axis is vertical.

See Also Page 229, 362

gSetViewEyeDistance

Syntax

C/C++:	**void gSetViewEyeDistance**(float dist);

F90:	**subroutine gSetViewEyeDistance**(dist) real, intent(in) :: dist

Arguments

dist
Perspective distance

= 0.0,	Switch to parallel view

Description The routine gSetViewEyeDistance() calculates a new eye position, keeping the view plane fixed, so that the perspective distance is equal to **dist**.

If **dist**<0.0, the new eye position is on the opposite side of the view plane, i.e. the direction of viewing is reversed.

If **dist**=0.0, the view is set to a parallel view, using the current view direction and view plane.

See Also Page 404

gSetViewPlaneDistance

Syntax

C/C++:	**void gSetViewPlaneDistance**(float dist);

F90:	**subroutine gSetViewPlaneDistance**(dist) real, intent(in) :: dist

Arguments

dist
Perspective distance

Description The routine gSetViewPlaneDistance() calculates a new position for the view plane, keeping the eye position fixed, so that the perspective distance is equal to **dist**.

If **dist** < 0, the position of the view plane is set so that the direction of viewing is reversed.

If **dist** = 0 or if the current view has no perspective defined, an error message is output and no further action is taken.

See Also

Page 404
gDefineParallelView
gDefinePerspView

gSetViewport

Syntax

C/C++:	**void gSetViewport2D**(GLIMIT *piclim2, GLIMIT *viewlim); **void gSetViewport3D**(GLIMIT3 *piclim3, GLIMIT *viewlim);

F90:	**subroutine gSetViewport2D**(piclim2, viewlim) **subroutine gSetViewport3D**(piclim3, viewlim) type (GLIMIT), intent(in) :: piclim2,viewlim type (GLIMIT3), intent(in) :: piclim3

Arguments

piclim2
2D picture coordinate limits

piclim3
3D picture coordinate limits

viewlim
Viewport coordinate limits in current paper units

Description

The routines gSetViewport2D() and gSetViewport3D() set up a viewport mapping from user picture coordinates in 2D or 3D to device or paper coordinates. The picture limits may cover any range but the viewport limits must lie within the range of the current device or paper limits. Viewport limits are in millimetres unless gDefinePictureUnits() has been called.

Once a viewport mapping is set up, all subsequent drawing and enquiry routines are affected by it. Thus any range of picture coordinates may be mapped onto any area of the device or paper.

The routine gSetViewportMode() controls the position of the viewport and whether the requested mapping is allowed to deform the picture.

If the viewport limits have a zero range or are completely outside the device limits an error message is output and the viewport transformation is unchanged. If the viewport limits are larger than the device limits, the viewport is clipped to the device limits for windowing purposes but the transformation is calculated according to the requested limits.

See Also

Page 49, 219
gDefinePictureUnits
gSetViewportMode

gSetViewportClipSwitch

Syntax

C/C++:	**void gSetViewportClipSwitch**(int sw);

F90:	**subroutine gSetViewportClipSwitch**(sw) integer, intent(in) :: sw

Arguments

sw
Viewport clipping switch

= GOFF,	Do not clip to viewport limits
= GON,	Clip to viewport limits (default)

Description

The routine gSetViewportClipSwitch() determines the setting of window limits after a call to gSetViewport2D(). When **sw** = GON, after a call to gSetViewport2D() the window limits are restricted to the viewport limits, and therefore it will not be possible to draw outside these limits. In these circumstances the setting of a viewport effectively becomes a mapping and clipping operation. If a call is made to gSetViewportClipSwitch(GOFF), any viewport setting will act simply as a mapping operation and window limits may extend outside the viewport limits.

Note that gSetViewportClipSwitch() only affects subsequent calls to gSetViewport2D().

See Also

Page 51, 221
gSetViewport2D
gEnqViewportState

gSetViewportMode

Syntax

C/C++:	**void gSetViewportMode**(int sw);

F90:	**subroutine gSetViewportMode**(sw) integer, intent(in) :: sw

Arguments

sw
Viewport scaling switch.

= GCENTRAL,	Keep aspect ratio and centre in viewport (default)
= GBOTTOMLEFT,	Keep aspect ratio and place at bottom left of viewport
= GDEFORMED,	Allow deformation of picture

Description

The routine gSetViewportMode() determines whether a viewport transformation set by gSetViewport2D() should keep the aspect ratio of the picture coordinate area or allow deformation.

If **sw** = GCENTRAL the picture area is centred in the viewport in either the vertical or horizontal direction as appropriate, whereas if **sw** = GBOTTOMLEFT the picture area is placed at the bottom left of the viewport. In both these cases the aspect ratio of the picture is kept by calculating the minimum scale value required to fit the picture coordinate area into the viewport area.

If **sw** = GDEFORMED the exact values of both the picture and viewport limits are used to calculate the viewport transformation which may allow different scales in the horizontal and vertical directions and so deform the picture.

The routine gSetViewportMode() **must** be called before gSetViewport2D() to have an effect on the viewport transformation.

If **sw** is out of range, a warning message is output and the default viewport scaling is used.

See Also

Page 50, 220, 274
gSetViewport2D

gSetViewTransformMode

Syntax

C/C++:	**void gSetViewTransformMode**(int mode);

F90:	**subroutine gSetViewTransformMode**(mode) integer, intent(in) :: mode

Arguments

mode

Viewing/Transformation mode

= GHARD,	Switch to hardware viewing/transformation mode
= GSOFT,	Switch to software viewing/transformation mode

Description

When operating on 3D devices, this routine can be used to switch between hardware and software viewing/transformation modes. Where **mode** = GHARD (the default), all changes to the current transformation and/or view are passed to the device in order for it to interpret 3D coordinates directly. GINO does however maintain a copy of both the transformation and viewing state so the appropriate enquiries can be made.

When **mode** is set to GSOFT, the 3D device is set back to a default transformation state, and GINO will interpret all 3D coordinates through its own 3D pipeline.

This routine has no effect on 2D devices.

See Also

Page 371

gSetViewUpDirection

Syntax

C/C++:	**void gSetViewUpDirection**(float dx, float dy, float dz);

F90:	**subroutine gSetViewUpDirection**(dx, dy, dz) real, intent(in) :: dx,dy,dz

Arguments

dx,dy,dz
Vector to be projected parallel to the picture Y-axis

Description

The routine gSetViewUpDirection() defines the vector (**dx,dy,dz**) which would be projected parallel to the y-axis of the picture plane when gGenerateView() or gUpdateView() is called to set up the view transformation. If a zero vector is specified, an error message is output and no further action is taken.

The view up direction is internally modified by the routines which rotate the line of sight - gViewRotate() and gViewTurn.

See Also

Page 402
gGenerateView
gViewRotate
gViewTurn
gUpdateView

gSetWindow

Syntax

C/C++:	**void gSetWindow2D**(GLIMIT *window2); **void gSetWindow3D**(GLIMIT3 *window3);

F90:	**subroutine gSetWindow2D**(window2) **subroutine gSetWindow3D**(window3) type (GLIMIT), intent(in) :: window2 type (GLIMIT3), intent(in) :: window3

Arguments

window2,window3
Position of the window boundaries in picture coordinates

Description

The routines gSetWindow2D() and gSetWindow3D() allow the user to define new 2D or 3D window limits. The window limits are clipped to the current device limits and windowing is switched on.

With windowing on, all movements outside the limits are noted, so the re-entry point can be calculated.

See Also

Page 223

gSetWindowMode

Syntax

C/C++:	**void gSetWindowMode**(int sw);

F90:	**subroutine gSetWindowMode**(sw) integer, intent(in) :: sw

Arguments

sw
Windowing switch

= GOFF,	Switch windowing off (default)
= GON,	Switch on the windowing limits set prior to switching off
= GON2D,	Switch 2-D windowing on and set the limits to those of the current viewport limits
= GON3D,	Switch 3-D windowing on and set the limits to those of the current viewport limits

Description

The routine gSetWindowMode() switches windowing on or off, or sets the window limits to the current device limits.

Windowing can be switched on or off at any time during the picture sequence. Initially the windowing limits are set to the current device limits, so that the first time gSetWindowMode() is called these will be its limits for an argument value of **sw** not equal to GOFF.

With windowing on, all movements outside the limits are noted, so the re-entry point can be calculated.

The window limits are defined in picture coordinates and cannot be transformed.

See Also

Page 222
gSetPolygonWindow
gEnqPolygonWindowList
gSetWindow2D
gSetWindow3D
gSetWindowMode

gSetWorkingDir

Syntax

C/C++:	**int gSetWorkingDir**(char directory[]);

F90:	**integer function gSetWorkingDir**(directory) character*(*), intent(in) :: directory

Arguments

directory
Directory pathname

Description

The system utility gSetWorkingDir() allows an application to change the current directory in which it operates. The directory pathname must obviously be in the correct format for the implementation, and thus may render the application implementation dependent.

The function returns a non zero value if the operation was not successful for any reason and zero if successful.

Note that when the application finishes, the current directory will still be that set by gSetWorkingDir() unless reset before application termination.

See Also

Page 460
gEnqWorkingDir

gSetWorkspaceLimit

Syntax

C/C++:	**void gSetWorkspaceLimit**(int n)**;**

F90:	**subroutine gSetWorkspaceLimit**(n1, n2) integer, intent(in) :: n1,n2

Arguments

n

Number of real words reserved for use as a workspace area

= 0, Workspace area is freed

n1,n2

Range of real words in global workspace area

= 0, Workspace area is freed

Description

The routine gSetWorkspaceLimit() defines the size of GINO's workspace area. The size of the workspace area must be large enough to cope with the combined requirements of the workspaces that exist at any given time. Workspaces are required for storing polygons (see gDefinePolygonWorkspace()), for storing a list of polygon identifiers (see gSelectPolygons()) and for area-filling polygons (see gFillSelectedPolygons()). Additional space is also required if software emulation of picture segments is required and this is to be held in memory rather than in a scratch disk file (see gDefineSegWorkspace() and gSetSegMode()).

An existing workspace area may be enlarged by calling this routine a second or subsequent time each with a larger value of **n** or **n2**. A smaller value does not reduce the area.

The complete workspace area is removed (and all its contents lost) when gSetWorkspaceLimit() is called with all arguments = 0 and by gCloseGino(). However, any workspace defined with gSetWorkspaceLimit() and its contents is maintained across device nominations, and therefore does not need to be redefined for each device.

See Also

Page 33, 51, 246
gFillSelectedPolygons
gDefinePolygonWorkspace
gSelectPolygons
gDefineSegWorkspace
gSetSegMode

gShear

Syntax

C/C++:	**void gShear2D**(int dep, float a); **void gShear3D**(int dir, int dep, float a);

F90:	**subroutine gShear2D**(dep, a) **subroutine gShear3D**(dir, dep, a) integer, intent (in) :: dep,dir real, intent (in) :: a

Arguments

dep
The dependent direction

= GXAXIS,	X-axis
= GYAXIS,	Y-axis
= GZAXIS,	Z-axis

dir
The direction of the shear

= GXAXIS,	X-axis
= GYAXIS,	Y-axis
= GZAXIS,	Z-axis

a
Shear factor

Description

The routines gShear2D() and gShear3D() superimpose a shear factor **a** along the dependent axis **dep**, such that the point one unit along the sheared axis is translated a distance **a** in a parallel direction along another axis. The sign of the shear factor will indicate direction.

In the case of gShear2D(), the shear may be with respect to the X or Y axis only and the direction will be parallel to the other axis. In the case of gShear3D(), the shear may be with respect to any axis and in a direction parallel to the axis specified by **dir**, the third axis being unaffected.

In space mode (the default) this factor is specified with respect to the current space axes, whilst in picture mode (see gSetTransformMode()) the factor is specified with respect to the picture axis.

If transforming is not on before this routine is called then gShear2D() switches it on.

See Also

Page 231, 364
gSetTransformMode

gShift

Syntax

C/C++:	**void gShift2D**(float dx, float dy); **void gShift3D**(float dx, float dy, float dz);

F90:	**subroutine gShift2D**(dx,dy) **subroutine gShift3D**(dx,dy,dz) real, intent (in) :: dx,dy,dz

Arguments

dx,dy,dz
X,Y (and Z) displacement of origin in current units

Description

The routines gShift2D() and gShift3D() move the transformation origin by the specified amounts.

In space mode (the default) this increment is specified with respect to the current space axes, whilst in picture mode (see gSetTransformMode()) the increment is specified with respect to the picture axes.

If transforming is not on before this routine is called then gShift2D() switches it on.

See Also

Page 228, 360
gSetTransformMode

gSkipCGMElement

Syntax

C/C++:	**void gSkipCGMElement**(int element);

F90:	**subroutine gSkipCGMElement**(element) integer, intent(in) :: element

Arguments

element
CGM element identifier

Description

The routine gSkipCGMElement() skips a CGM element when interpreting a CGM metafile element by element, having obtained the next element identifier using gGetCGMElement().

The CGM file must be opened using the routine gOpenCGMFile().

See Also

Page 70
gOpenCGMFile
gGetCGMElement

gStartBatchUpdate

Syntax

C/C++:	**void gStartBatchUpdate**(void);

F90:	**subroutine gStartBatchUpdate**

Arguments None

Description The routine gStartBatchUpdate() is used to begin a batch of modifications to a display surface or display file. No changes will be displayed on the screen until the batch has been ended via gEndBatchUpdate().

If gStartBatchUpdate() is called while a batch of updates is in progress, GINO makes an internal call to gEndBatchUpdate().

See Also Page 50
gEndBatchUpdate

gStartPolygon

Syntax

C/C++:	**void gStartPolygon**(void);

F90:	**subroutine gStartPolygon**

Arguments None

Description The routine gStartPolygon() causes a new polygon to be started. gStartPolygon() makes an internal call to gSetPolygonMode(GON) i.e. it switches on the storing of polygon vertices. If there is a previously defined polygon that was not closed by calling gEndPolygon(), then gStartPolygon() closes it.

See Also Page 247
gEndPolygon
gSetPolygonMode

gStartTextBlock

Syntax

C/C++:	**void gStartTextBlock**(float xbeg, float ybeg);

F90:	**subroutine gStartTextBlock**(xbeg, ybeg) real, intent(in) :: xbeg,ybeg

Arguments ***xbeg,ybeg***
The absolute coordinates of the start of a text block

Description The routine gStartTextBlock() moves the current drawing position to the specified coordinate and indicates the start of a text block.

If character transformations are switched off the position is specified in picture coordinates. If character transformation is switched on (using gSetCharTransformMode()) the position is specified in the current units and with respect to the current axes.

Subsequent calls to gMoveToNextLine() or use of the *N escape sequence will cause a move to the next line within the text block.

See Also Page 154
gMoveToNextLine
gDisplayStr
gSetCharTransformMode

gSuspendDevice

Syntax

C/C++:	**void gSuspendDevice**(void);

F90:	**subroutine gSuspendDevice**

Arguments None

Description The routine gSuspendDevice() suspends output to the currently nominated device. This may result in the device remaining in graphics mode or the window remaining on the screen as appropriate to the device. gSuspendDevice() is useful if control is required to be temporarily suspended while output is directed to another device.

If any further pictures are to be drawn, a device must be nominated prior to calling any further GINO routines.

If a new device is nominated without a previous call to gSuspendDevice() or if gCloseGino() is called, output to the old device is terminated.

Note: Under Windows, gSuspendDevice() must be followed by a call to gCloseGino() for the window to remain on the screen.

See Also Page 52
gCloseGino

gSwitchBrokenLineStyles

Syntax

C/C++:	**void gSwitchBrokenLineStyles**(int switch);

F90:	**subroutine gSwitchBrokenLineStyles**(switch) integer, intent(in) :: switch

Arguments ***switch***
Broken line type

= GCONTDASH
= GDISCONTDASH

Description The routine gSwitchBrokenLineStyles() switches ALL the lines styles in the line style table to the type defined in **switch**. The settings of repeat length, dash and dot lengths are not affected. Individual entries in the line table can be changed using gDefineBrokenLineStyle().

See Also Page 128
gDefineBrokenLineStyle

gTimeDelay

Syntax

C/C++:	**void gTimeDelay**(int wait);

F90:	**subroutine gTimeDelay**(wait) integer, intent(in) :: wait

Arguments ***wait***
Delay time in milliseconds

Description The system utility gTimeDelay() provides a means to suspend the GINO application for a specified amount of time. The time delay is only as accurate as the system clock.

See Also Page 463

gTransformHomogPoint3D

Syntax

C/C++:	**void gTransformHomogPoint3D**(float xs, float ys, float zs, GPOINT3 *point, float *wh);

F90:	**subroutine gTransformHomogPoint3D**(xs, ys, zs, point, wh) real, intent(in) :: xs,ys,zs type (GPOINT3), intent(out) :: point real, intent(out) :: wh

Arguments ***xs,ys,zs***
3-D position in space coordinates

point
Transformed position in homogeneous coordinates

wh
Transformed position in homogeneous coordinates

Description The routine gTransformHomogPoint3D() transforms the point (**xs,ys,zs**) into homogeneous coordinates using the current global transformation (modelling and viewing). If transforming is switched off, **point** is set to **xs,ys,zs** and **wh** is set to 1.0 respectively. Homogeneous coordinates are related to picture coordinates in the following way:

xp = point.x/wh;

yp = point.y/wh;

zp = point.z/wh;

If the modelling transformation contains no perspective, then by definition **wh** = 1.0. With perspective in the modelling transformation, the view plane corresponds to **wh** = 1.0 and the eye plane corresponds to **wh** = 0.0. All points on or behind the eye plane do not project into picture coordinates. GINO clips all graphical output which lies behind the plane **wh** = 0.001.

See Also Page 239, 368

gTransformPoint

Syntax

C/C++:	**void gTransformPoint2D**(float xs, float ys, GPOINT *point2); **void gTransformPoint3D**(float xs, float ys, float zs, GPOINT3 *point3);

F90:	**subroutine gTransformPoint2D**(xs, ys, point2) **subroutine gTransformPoint3D**(xs, ys, zs, point3) real, intent(in) :: xs,ys,zs type (GPOINT), intent(out) :: point2 type (GPOINT3), intent(out) :: point3

Arguments ***xs,ys,zs***
2D or 3D position in space coordinates

point2,point3
Transformed position in picture coordinates

Description The routines gTransformPoint2D() and gTransformPoint3D() transform the specified point into picture coordinates according to the current global transformation matrix (modelling and viewing). If transforming is switched off, the returned point is the same as the input point.

If the global transformation contains 3-D (in the case of gTransformPoint2D()) or perspective terms (either routine), warning messages are output. In addition to this, (in the case of gTransformPoint3D()) if the point lies behind the eye plane, which means that it does not project into picture coordinates, **point3** is returned set to zero and a warning message is output.

See Also Page 238

gTrueCol

Syntax

C/C++:	**int gTrueCol**(float red, float green, float blue);

F90:	**integer function gTrueCol**(red, green, blue) real, intent (in) :: red,green,blue

Arguments

red
Red intensity, 0.0 to 1.0

green
Green intensity, 0.0 to 1.0

blue
Blue intensity, 0.0 to 1.0

Description

The function gTrueCol() returns a packed 24bit true colour value derived from the supplied red, green and blue intensities.

The returned value contains 8 bits of colour information for each of the three primaries and can be used for passing true colour information to the current device through routines such as gSetLineColour() etc, when operating in true colour mode (as set by gSetColourInfo()).

Devices which can operate in true colour mode and have been set in that mode using gSetColourInfo() can accept true colour information through gSetColourInfo(), gDrawPixel() and gDrawPixelArea(). Users should refer to Appendix B to determine whether a device can operate in true colour mode before using this routine.

See Also

Page 217
gDrawPixel
gDrawPixelArea
gSetColourInfo
gSetLineColour

gTrueLen

Syntax

C/C++:	**int gTrueLen**(char string[]);

F90:	**integer function gTrueLen**(string) character*(*), intent (in) :: string

Arguments

string
Character string

Description

The system utility gTrueLen() provides an implementation independent means of enquiring the length of a character variable up to and including the last non-blank character. A null string will return a length of zero.

See Also Page 467

gUntransformHomogPoint3D

Syntax

C/C++:	**void gUntransformHomogPoint3D**(float xh, float yh, float zh, float wh, GPOINT3 *point);

F90:	**subroutine gUntransformHomogPoint3D**(xh, yh, zh, wh, point) real, intent(in) :: xh,yh,zh,wh type (GPOINT3), intent(out) :: point

Arguments

xh,yh,zh,wh
Position in homogeneous coordinates

point
Untransformed 3-D position in space coordinates

Description

The routine gUntransformHomogPoint3D() transforms the point (**xh,yh,zh,wh**) into space coordinates using the inverse of the current modelling transformation. If transforming is switched off, **point** is set to **xh,yh,zh**. There is redundant information in (**xh,yh,zh,wh**), i.e. gUntransformHomogPoint3D() has to reduce four coordinate values down to three values. If there is no perspective in the transformation, **wh** is ignored. Otherwise, a check is made to see if the position in homogeneous coordinates is consistent with the current modelling transformation. If not, a warning message is output.

Homogeneous coordinates are related to picture coordinates as follows:

xp = xh/wh;

yp = yh/wh;

zp = zh/wh;

The view plane corresponds to **wh** = 1.0 and the eye plane corresponds to **wh** = 0.0.

Homogeneous coordinates are obtained from calls to gTransformHomogPoint3D().

See Also Page 239, 368

gUntransformPoint

Syntax

C/C++:	**void gUntransformPoint2D**(float xp, float yp, GPOINT *point2); **void gUntransformPoint3D**(float xp, float yp, float zp, GPOINT3 *point3);

F90:	**subroutine gUntransformPoint2D**(xp, yp, point2) **subroutine gUntransformPoint3D**(xp, yp, zp, point3) real, intent(in) :: xp,yp,zp type (GPOINT), intent(out) :: point2 type (GPOINT3), intent(out) :: point3

Arguments

xp,yp,zp
2D or 3D position in picture coordinates

point2,point3
Untransformed position in space coordinates

Description

The routines gUntransformPoint2D() and gUntransformPoint3D() transform the specified point into space coordinates according to the current global transformation matrix (modelling and viewing). If transforming is switched off, the returned point is the same as the input point.

The routine gUntransformPoint2D() uses the inverse of the 2-D part of the current global transformation. If the global transformation contains 3-D (in the case of gUntransformPoint2D()) or perspective terms (either routine), warning messages are output and the returned point is set to zero.

See Also

Page 238, 369

gUpdateView

Syntax

C/C++:	**void gUpdateView(void);**

F90:	**subroutine gUpdateView**

Arguments

None

Description

The routine gUpdateView() updates the viewing transformation matrix from data specified by the viewing routines. (It does **not** affect the modelling transformation matrix). The viewing transformation is always pre-multiplied with the modelling transformation matrix to form the complete transformation through which all subsequent drawing will pass.

The default view is a parallel view pointing along the negative z-axis with the view centre at the origin. It can be redefined by calls to gDefinePerspView(), gDefineSphericalView() or gDefineParallelView() and modified by calls to gSetViewEyeDistance(), gMoveViewCentre(), gViewShift() or gViewTurn(). The default view can be restored by calling gInitView(). If the current view has perspective defined, it may also be modified by calls to gSetViewPlaneDistance() or gViewRotate().

Unless gPosViewCentre() is called the view centre is projected onto the centre of the current window limits. The window limits are set to the viewport limits unless a user-defined window is currently specified (see gSetWindow2D(), gSetWindow3D() or gSetWindowMode()). If gSetViewUpDirection() is not called, gGenerateView() attempts to project the 3-D y-axis parallel to the screen y-axis, or if the 3-D y-axis is parallel to the view direction, then the 3-D x-axis is projected parallel to the screen x-axis.

The viewing transformation matrix can be initialized either by calling gSetTransform() with an argument of -1, or calling gInitView().

See Also

Page 388
gSetTransform
gMoveViewCentre
gInitView
gDefineParallelView
gSetViewEyeDistance
gDefinePerspView
gPosViewCentre
gViewRotate
gViewShift
gDefineSphericalView
gViewTurn
gSetViewUpDirection
gSetViewPlaneDistance
gSetWindow2D
gSetWindow3D
gSetWindowMode

gViewRotate

Syntax

C/C++: **void gViewRotate**(int plane, float angle, float dist);

F90: **subroutine gViewRotate**(plane, angle, dist)

integer, intent(in) :: plane
real, intent(in) :: angle,dist

Arguments

plane

Plane in which rotation is to take place

= GYZPLANE,	YZ plane
= GXZPLANE,	XZ plane
= GXYPLANE,	XY plane

angle
Angle of rotation in degrees

dist
Distance from eye position to centre of rotation

Description The routine gViewRotate() calculates a new eye position such that the line of sight is rotated about a point of distance **dist** along the view direction from the eye position. The line of sight is rotated by the specified **angle** in the specified plane. A positive rotation is anticlockwise when looking at the positive side of the plane.

Rotating the line of sight has the side-effect or modifying the view-up direction in moost circumstances.

If **plane** is not in range or if the current view has no perspective defined, an error message is output and no further action is taken.

See Also Page 411
gSeViewUpDirection

gViewShift

Syntax

C/C++:	**void gViewShift**(float dx, float dy, float dz);

F90:	**subroutine gViewShift**(dx, dy, dz) real, intent(in) :: dx,dy,dz

Arguments

dx,dy,dz
Amount by which line of sight is to be shifted

Description The routine gViewShift() moves the line of sight by the specified vector increment along the current view axes..

See Also Page 410

gViewTurn

Syntax

C/C++:	**void gViewTurn**(float xr, float yr, float zr, float dx, float dy, float dz, float angle);

F90:	**subroutine gViewTurn**(xr, yr, zr, dx, dy, dz, angle) real, intent(in) :: xr,yr,zr,dx,dy,dz,angle

Arguments

xr,yr,zr
Origin of rotation axis in space coordinates

dx,dy,dz
Rotation axis vector

angle
Angle of rotation in degrees

Description

The routine gViewTurn() rotates the current view direction and view centre about an arbitrary 3-D rotation axis. The rotation axis points in the direction (**dx,dy,dz**) and goes through the point (**xr,yr,zr**). The view is rotated in a right-handed sense with respect to the rotation axis by an angle in degrees specified by **angle**.

Rotating the line of sight has the side-effect or modifying the view-up direction in moost circumstances.

If a zero rotation axis vector is specified, an error message is output and no further action is taken.

Unlike gViewRotate(), gViewTurn() can rotate a parallel view.

See Also

Page 410
gSeViewUpDirection
gViewRotate

gWaitForEvent

Syntax

C/C++:	**void gWaitForEvent**(int *intype);

F90:	**subroutine gWaitForEvent**(intype) integer, intent(out) :: intype

Arguments

intype
The event type being returned

= GNULL,	Null event type
= GKEYPRESS,	Key or mouse button press
= GSEGMENT,	Picture segment number
= GSEGMENTANDKEY,	Picture segment number and key/mouse button
= GLOCATOR,	Screen position and key/mouse button press
= GSTRING,	Text string
= GREALS,	String of real values
= GINTEGERS,	String of integer values
= GMOVEMENT,	Pointer, mouse or tablet movement
= GKEYRELEASE,	Key or mouse button release
= GRESIZE,	Window resize event
= GPOINTERLEAVING,	Pointer leaving window
= GPOINTERENTERING,	Pointer entering window
= GMOUSEWHEEL,	Mouse wheel movement

Description

The routine gWaitForEvent() reads the next discrete event on the event queue and passes back its event type in **intype**. If the Null event has been enabled (gAddEventType(GNULL)), gWaitForEvent() will return immediately, even if no event has occurred, otherwise the routine will wait for a requested event to be placed on the event queue by the device and then returns to the application.

If events are not available on the current output device GINO will attempt to emulate the first 5 events through the following means. Only the last event enabled with gAddEventType() will be emulated in this way and **intype** will return one of the following values. Events are emulated as follows:

GKEYPRESS - Single key or cursor input

GSEGMENT - Cursor followed by gEnqSegHit()

GSEGMENTANDKEY - Cursor input followed by gEnqSegHit()

GLOCATOR - Cursor input

GSTRING - Input record

All other information returned by gWaitForEvent() is obtained by calling the routine gGetEventRecord().

If gWaitForEvent() is called before calling routine gAddEventType() or no event of the types requested have occurred, **intype** will be set to 0.

See Also

Page 451
gGetEventRecord
gAddEventType

Appendix A

MACHINE IMPLEMENTATIONS

GENERAL

This Appendix contains machine-specific and operating system-specific details. Compiler-specific details can be found in the on-line documentation.

The following table shows the default word size and numeric range used by GINO under the various implementations available.

	Silicon Graphics SV1	Silicon Graphics double-double precision	CVF, FTN95, LF90/95 & SGI double-precision	All others
Default Integer	*8	*8	*4	*4
Default Real	*8	*8	*8	*4
Integer range	-2**62+1 to 2**62-1	-2**62+1 to 2**62-1	-2**31+1 to 2**31-1	-2**31+1 to 2**31-1
Real range	-1.0e2439 to 1.0e2439	-1.0e**307 to 1.0e**307	-1.0e**307 to 1.0e**307	-1.0e**37 to 1.0e**37

The following table shows the default Fortran channel numbers used by GINO for the various file handling tasks. It also shows the GINO Configuration Variable that can be used to change the default and also the maximum that can be used.

	Config. Variable	VAX ALPHA OpenVMS	UNIX	Salford FTN95	Absoft	Lahey LF90/LF95	Compaq Visual Fortran
Screen Driver Output	NDEVIC	5	6	Direct	Direct	Direct	Direct
Plotter Driver Output	NDEVIC	1	1	1	1	1	1
Metafile Output	NSAVDF	7	7	7	7	7	7
Error Message Input	NFMESS	Any	Any	Any	Any	Any	Any
Error Message Output	NFERTR	6	0[1]	6	6	6	6

Software Display File	NFSDF	Any	Any	Any	Any	Any	Any
Software Font File	NFFONT	Any	Any	Any	Any	Any	Any
GINOMENU Icon File	NFICON	Any	Any	Any	Any	Any	Any
GINO State Stack File	NFSTAT	Any	Any	Any	Any	Any	Any
Highest Valid File Unit		99	99	99	100	32767	32767

[1] Unit 6 on HP/UX

This and additional compiler and machine specific information may be enquired through the GINO enquiry routine gEnqImpAttriubs() which returns information in a structure GIMPLEMENTATION.

Refer to Appendix B for details on how to use the configuration variables to change the defaults if required.

UNIX

File Handling

If the routine gSetDeviceFilename() has not been used, GINO will write to a file called xxxxxx.OUT where xxxxxx is the current device nomination routine. Alternatively, under Fortran 90, the file name may be assigned by setting an environment variable of the form FORTn, where n is the default file unit as detailed above. Thus, ‘setenv FORT7 ginoex1.sav’ will assign an external file name for a SAVDRA output file.

8-bit Data

Certain printing devices such as laser and dot-matrix printers use all ASCII integer codes in the range 0-255 for graphics output.

The printing device must therefore be configured for 8 bit data and users should ensure raw 8 bit data is passed to it. If data is passed directly to the device through a /dev/gino_n port, the line should be configured to receive raw data and a character size of 8 within the termcap (5) database. Alternatively, a line may be configured temporarily by using the stty command as part of a shell script, eg:

(stty 9600 -parity tabs raw ixon;cat file)> /dev/ttya

OpenVMS

File Handling

If the routine gSetDeviceFilename() or an OPEN statement has not been used, GINO will send the output to a logical name FOR0nn where nn is either the default channel number as specified in the above table or is taken from the configuration variable applicable to the current device.

If the logical name has not been assigned anywhere, a default file will be opened of the name FOR0nn.DAT.

The logical name can be assigned to an alternative file or terminal port as follows:

```
$ DEFINE FOR001 PLOT.DAT

$ DEFINE FOR001 TXA3:
```

File Format

The default file format used by GINO under OpenVMS is unformatted Fortran carriage_control. i.e. each record begins with a 'NULL' character. (This type of file is also created when calling gSetDeviceFilename() with **ntype** = 0). This type of file can then be copied directly to the printer or plotter using the COPY, TYPE or PRINT commands.

If the PRINT command is used with unformatted files either the /PASSALL option must be used, eg:

```
$ PRINT/QUEUE=queuename/PASSALL PLOT.HPJ
```

or the default form associated with the queue must be defined with /NOTRUNCATE/NOWRAP options to suppress any page control by the print symbiont. This is achieved using the following steps (you must be System Administrator):

```
$DEFINE/FORM/NOTRUNCATE/NOWRAP formname formnumber

$START/QUEUE/DEFAULT=FORM=formname/FORM_MOUNTED=formname
queuename
```

Exporting Files to other Systems

If the file is to be exported to another computer system, e.g. a PC to be read into a DTP or word-processor package, formatted output files are recommended and these are created either by calling gSetDeviceFilename() with **ntype** = -1 or there may be a driver specific configuration setting which will ensure this type of file is created by default. In each case each record is written with no carriage control character (A format). This filetype will have LIST carriage_control attributes so it can also be used on a local printer under OpenVMS.

CGM Files

GINO CGM files are opened as Direct-Access Unformatted with Fixed Records of length 80 bytes.

The second parameter in the routine gSetDeviceFilename() is ignored and if an OPEN statement is used, only the filename is looked at and the file is re-opened internally by GINO. The CGM interpreter can read in files with records of any length up to 512 bytes.

8-bit Data

Certain printing devices such as laser and dot-matrix printers use all ASCII integer codes in the range 0-255 for graphics output.

The terminal line therefore must be configured for 8-bit output as the line driver must pass everything through without interpreting any control characters. This is achieved with the /PASTHRU qualifier to SET TERM:

```
$ SET TERM/PASTHRU/PERM TXA3:
```

Microsoft Windows

File Handling

If the routine gSetDeviceFilename() or an OPEN statement has not been used, GINO will write to a file called xxxxxx.OUT where xxxxxx is the current device nomination routine.

Appendix B

DEVICE DRIVERS

Device Drivers Introduction

This Appendix contains details of all the most commonly used device drivers in GINO. - Check the file DRIVERS.LST in your installation directory or check with your System Administrator for the list of drivers that are available at your site.

The drivers are grouped in three sections; Screens, Printer/Plotters and Device-independent Metafiles. Each driver has one table listing the available nomination routines and a second table listing the characteristics and attributes of each nomination routine.

It is important to note that the Characteristics table details the hardware characteristics of that device. As the general GINO philosophy is to use a hardware function if possible and if not, emulate it in software, a **No** in a table will produce a software emulation of that function if possible.

There are advantages and disadvantages for using hardware or software features such as; using software is recommended for consistent appearance on all devices, but using hardware is recommended for speed, smaller file sizes and a more professional appearance. For this reason, most features can be switched between hardware and software and the following table lists the routines used for this purpose.

The table also lists the routines used for selecting/setting a particular feature as well as the enquiry routines used for checking the current settings. The general enquiry routine gEnqDeviceState() can also be used to enquire what facilities are available on any one device.

	Selection/Setting	Hard/Software switch	Attributes Enquiry
Max Width x Height	gSetDrawingLimits()	N/A	None
Default Width x Height	gSetDrawingLimits()	N/A	gEnqDrawingLimits()
Colours / Pens	gSetLineColour()	N/A	gEnqLineColour()
Colour Palette	gDefineRGB()/gSetColourInfo()	N/A	gEnqRGB()/gEnqColourInfo()
Broken Linestyles	gSetBrokenLine()	gSetBrokenLineMode()	gEnqBrokenLine()
Drawing Mode	gSetPenType()	None	gEnqSelectedPen()
Thick Lines	gSetLineWidth()	gSetLineWidthMode()	gEnqLineWidth()
Line Ends	gSetLineEnd()	gSetLineWidthMode()	gEnqLineEnd()
Arcs	gDrawArcxxx()	gSetArcMode()	None
Symbols	gDrawMarker()	gSetCharTransformMode()	None
Fonts	gSetCharFont()/gSetFontxxx()	gSetCharTransformMode()/gSetSoftChars()	gEnqFontStyle()
Character Sizes	gSetCharSize()	gSetCharTransformMode()/gSetSoftChars()	gEnqCharAttribs()
Character Angles	gSetStrAngle()	gSetCharTransformMode()/gSetSoftChars()	gEnqCharAttribs()
Italic Characters	gSetItalicAngle()	gSetCharTransformMode()/gSetSoftChars()	gEnqCharAttribs()
Polygonal Filling	gFillRect()/gFillSelectedPolygons()/ gFillPolygonxxx()	gSetFillMode()	None
Segments	gxxxSeg()	gSetSegMode()	gEnqSegAttribs()
Image Handling	gxxxPixel()	None	gEnqPixelxxx()
Cursor Types	gSetCursorType()	N/A	gEnqCursorType()
Cursor Actions	gSetCursorAction()	N/A	gEnqCursorAction()
Cursor Positioning	gSetCursorPos()	N/A	None
Event Types	gAddEventType()	N/A	None
Batch Updates	gxxxBatchUpdate()	N/A	None
Clipping/Masking	gSetWindowxxx()/gSetMaskxxx()	gSetClippingMode()	gEnqWindowState()/ gEnqMaskState()
Transform/Viewing	gScalexx()/gRotatexx()/gShiftxx()/ gUpdateView()	gSetTransformSwitch()	gEnqTransformState()/ gGetViewParams()
Shading	gSetShadingMode()	N/A	gEnqShadingMode()
Lights	gDefineLight()	N/A	gEnqLightAttribs()
Texture Mapping	gDefineTexture()	N/A	gEnqTextureMappingMode()
Auxiliary Drawing Areas	gxxxAuxDrawingArea()	N/A	None
Window/Device Titling	gSetDeviceTitle()	N/A	None

N\A Implies that there is no method of switching the feature into software emulation

None Implies that there is no routine available for this purpose

Configuration File

The facility to change device driver output channels under F90 and other driver settings is by way of the GINO configuration file GINO.CON (gino.config under UNIX). GINO.CON is initially just supplied with serial numbers appropriate to each licenced package. These encrypted serial numbers must not be altered as each GINO package will not run if it detects an illegal number.

The configuration settings should be added on new lines in the following form:

VARIABLE=setting

where VARIABLE is up to 12 characters long (always in upper case) and setting can be any string or value up to 50 characters.

All other records in the file are ignored and can be used for comments. Up to 50 configuration settings are permitted.

At the beginning of each subsequent driver section, there is a table listing all of the available configuration settings for that group of devices.

See your *Getting Started Guide* for more details about the configuration file.

Dummy Device

Device Nominations

	Nomination Routine	Description of Device
A	gDummy()	Dummy device called if no other is nominated

Device Characteristics

	A
Maximum Width	No Limit
Maximum Height	No Limit
Default Width (mm)	200.0
Default Height (mm)	200.0
Colours / Pens	255
Colour Palette	Dynamic
Broken Linestyles	No
Drawing Modes	No
Thick Lines	No
Line Ends	No

Arcs	No
Symbols	No
Fonts	No
Character Sizes	Pseudo-hardware in multiples of 1.5mm square
Character Angles	0 or 90°
Italic Characters	No
Polygonal Filling	No
Segments	No
Image Handling	No
Clipping	No
Transform/Viewing	No
Shading	No
Lights	No
Texture Mapping	No
Window/Device Titling	No

The gDummy() device-driver produces no graphical output or metafile but can be used for checking the functionality of a program as all error-checking is still performed.

GINO nominates the gDummy() driver if no device driver has been nominated by the user's program.

SCREENS AND WORKSTATIONS

This section includes graphics screen, terminals and workstation devices.

Output Filenames and Unit Numbers (Fortran only)

By default, all graphics output is sent directly to the screen using an implementation-dependent Fortran unit number. On OpenVMS and UNIX system, the screen output (except GLX and X Windows) can be sent to a file which can then be TYPEed or 'cat'ed at a later stage. If a file has not been opened, a file will be created with a default name depending on the system, such as FOR005.DAT on OpenVMS or eps.out under UNIX. A different unit number can be used by setting the relevant GINO configuration variable for that device, alternatively, the routine gSetDeviceFilename() can be used whereby the actual filename is specified and GINO will automatically select the next available unit number.

Screen Driver Configuration Settings

The following table lists all the available settings applicable to drivers in this screens section:

Device Driver	Config. Variable	Settings	Default	Description
GLX	GLXTITLE	string		Window title if not set by gSetDeviceTitle()
REGIS	REGIS	N (1-99)	6	F90 output channel for REGIS driver
WINDOWS	NFERTR	N (0-2)	0	Dialogue window iconized (0), visible (1), none (2)
WINDOWS	MWINTITLE	string		Window title if not set by gSetDeviceTitle()
WOGL	NFERTR	N (0-2)	0	Dialogue window iconized (0), visible (1), none (2)
WOGL	WOGLTITLE	string		Window title if not set by gSetDeviceTitle()
WOGL	WOGLDEPTH	N(16,24,32)	32	OpenGL depth buffer depth
XWIN	XCOLS	1-256	64	Number of entries in colour palette
XWIN	XWINTITLE	string		Window title if not set by gSetDeviceTitle()

GLX OpenGL Extension to X

Device Nominations

	Nomination Routine	Description of Device
A	gGlx()	Default GLX Window
B	gGlxw(pause,double,xp,yp,width,height)	User-defined GLX Window
C	gGlxao(window_id,pixmap_id)	Opens GINO using an existing bitmap

Device Characteristics

	A	B	C
Default Width	$^2/_3$ Maximum	User Defined	Application Dependent
Default Height	$^2/_3$ Maximum	User Defined	Application Dependent
	Devices A,B and C		
Maximum Width	Server Dependent		
Maximum Height	Server Dependent		
Colours / Pens	1023		
Colour Palette	True colour GLX visual		
Broken Linestyles	No		
Drawing Modes	GERASER, GNOT, GAND, GOR, GXOR		
Thick Lines	Yes		
Line Ends	16		
Arcs	Yes		

Symbols	No
Fonts	Server Dependent
Character Sizes	Server Dependent
Character Angles	Server Dependent
Italic Characters	Server Dependent
Polygonal Filling	Single Polygons & Solid only
Segments	Yes
Image Handling	All Functions
Cursor Types	GHOURGLASS,GDEFAULT, GSMALLCROSS, GLARGECROSS, GX, GPOINTER & 74 X specific types
Cursor Actions	GPOLYLINE, GDEFAULT, GRUBBERBAND, GRUBBERBOX, GRUBBERSQUARE, GRUBBERELLIPSE, GRUBBERCIRCLE
Cursor Positioning	No
Event Types	GKEYPRESS, GLOCATOR, GMOVEMENT, GKEYRELEASE, GRESIZE, GPOINTERLEAVING, GPOINTERENTERING
Batch Updates	Yes
Clipping	Yes
Transform/Viewing	Yes
Shading	Yes
Lights	Yes
Texture Mapping	Yes
Auxiliary Drawing Areas	Yes (Up to 50 pairs)
Window/Device Titling	Yes
Mouse Positioning	Yes

Linking with GLX and Xlib Library

To satisfy all outstanding references made by this driver, the GL and XLIB libraries needs to be referenced in your link statement:

UNIX:

f90 -o myprog myprog.o -lgino-f90 -lGL -lGLU -lX11

Some UNIX systems may require extra libraries to satisfy further references from the X11 library. e.g. on Solaris:

f90 -o myprog myprog.o -lgino-f90 -lGL -lGLU -L/usr/openwin/lib
-R/usr/openwin/lib -lX11 -lXext -lXmu

Device Nomination

Three nomination routines are available with GLX-windows:

gGlx()

This nomination routines opens a single buffered, standard window of 2/3 screen size in the top left-hand corner.

gGlxw(pause,double,xp,yp,width,height)

This routine allows the user to set the position, size and action at program termination where:

pause

= 0	specifies there is no pause at gCloseDevice() (default)
= 1	specifies that the process will pause with a prompt at gCloseDevice()
= 2	specifies that the process will pause (without prompt) at gCloseDevice()

double

= 0	GINO creates and controls backing store ensuring all exposed areas are automatically repaired (default)
= 1	An OpenGL double buffered visual is used which is often superior in performance but should only be used where the whole window is continually being redrawn by the application.

xp,yp
specify the pixel position of the top left of the window (default = 0,0)

width,height
specify the pixel width and height of the window (default = 2/3 of max.)

The graphics window, however will always be cleared at the end of the program.

The nomination routine gGlx() is equivalent to calling gGlxw() as follows:

gGlxw(0,0,0,0,width,height)

where width,height are equivalent to two-thirds the dimensions of the full drawing area.

gGlxao(window_id,pixmap_id)

This routine allows GINO to draw to an existing window where window_id is the identifier of the external window and pixmap_id is the identifier of a pixmap the same size as the external window. If window_id or pixmap_id have a NULL value then they will be created by GINO.

The normal operation of the GLX-Windows driver is to open a window, map it, clear it and set up a colour table. A pixmap is also created to act as a backing store to enable the repainting of exposed regions.

When using gGlxao(), the window and pixmap are not created, but their attributes are obtained and used by GINO's initialization procedures. The window is not blanked and the colour map is not overwritten.

The window set up externally will need to be mapped before entering GINO.

If external drawing is to be used within the window, a pixmap will need to be created to allow damage repair on the window. To enable this all drawing must be done to the pixmap as well as to the window so that they are a copy of each other. Within the GINO gWaitForEvent()/gGetCursorEvent() loop, expose events on the window are trapped and used to trigger XCopyArea commands to repaint the window. If no external drawing is to be done, set pixmap_id to NULL and GINO will create and maintain the pixmap until gCloseDevice().

Single and Double Buffer Modes of operation

As can be seen above, GINO provides for two operational modes with its GLX driver.

I) Utilizing a GINO specific backing store (or pixmap)
ii) In OpenGL double buffered mode

The former works exactly the same as the XWIN driver which has the advantage of the user not worrying about expose events or iconization as these are dealt with by the driver, reparing and redrawing the appropriate areas from backing store. In this mode OpenGL is instructed to work in single buffered mode, drwing solely to the invisible backing store whilst GINO controls the copying of the backing store to the screen as appropriate to the GINO application.

Operating in OpenGL dubble buffered mode often provides a performance benefit, but at the expense of extra programming effort to cater for this mode of operation (see below). In this mode GINO drawns to the OpenGL back buffer, and on a call to gFlushGraphics or gEndBatchUpdate, OpenGL swps the cointents of the front buffer (i.e. The screen) with that of the back buffer providing almost instantaneous updates of the image. However, according to the strict rules of OpenGL, after this update has been completed, the contents of the back buffer are undefined and so the whole picture has to be redrawn to the back buffer again ready for the next update.

Unfortunately, different implementations of OpenGL on X workstations and PC X emulators operate differently and do not always provide for the pixmap mode of operation. The table below shows these in summary:

Environment	Pixmap Mode	Double Buffered Mode	Back Buffer Lost
AIX	No	Yes	No
CRAY	Yes	Yes	Yes
HP/UX	Yes	Yes	Yes
IRIX	Yes	Yes	Yes)
Solaris	No	Yes	No
L			
Exceed	Yes	Yes	Yes
MESA	Yes	Yes	No

By default the gGlx nomination routine will create a pixmap and OpenGL will work in single buffered mode except where such mode of operation is not possible. Fortunately in these cases, the OpenGL implementation automatically repairs exposed area (AIX) or the back buffer is not lost so GINO can carry out the repairs on an expose event (Solaris), so the extra programming effort normally required for double buffered modes is not needed.

Double buffered mode is available in all implementations of GINO and this can be selected by calling the gGlxw with **double** set to 1. This mode should only be used for animation type applications where the program is either in a hard loop where the window is continually being cleared (using gNewDrawing()) and a new image is being created, ending the loop with gFlushGraphics(). Alternatively, GINO events may be used, catering for key and mouse button presses to control the animation.GINO example programs 10 and 11 are cases in point. In this mode of operation the GLX driver causes an event type GRESIZE whenever an expose event occurs (see below for exceptions) allowing the application to redraw the complete contents of the window to repair the exposed area. Such an event should be added to the event list using gAddEventType() and progrmmed accordingly in any application that operates in double buffered mode. GINO example program 9 shows this type of application.

Where the contents of the back buffer are not lost after swapping the contents (AIX, PC MESA and SOLARIS), GINO will automatically repair areas after expose events and iconization and not generate the GRESIZE event, but if your application is to be machine independent the above guidelines should be followed.

Window & Pixmap Identifiers

The identifiers of the primary window and pixmap can be obtained though a special device driver dependent routine:

gGlxid(window_id, pixmap_id)

These values may be passed to an appropriate X routine requiring such information.

Window Size

If the user of the application changes the window size during operation, the driver will ensure that the picture is updated by redrawing from the backing store, but GINO is not able to take any further action until after a gNewDrawing() is called. At this point the programmer can re-enquire the window size using gEnqDrawingLimits() and take the appropriate action.

If the window size is changed by the user, this takes priority over the programmers request for a change through gSetDrawingLimits().

Window Title

The default window title of a GINO application is a string composed of the GINO version and license owner. This can be superceded in the following order of precedence:

1) GINO config variable GLXTITLE

2) System environment variable GLXTITLE

3) Application call to gSetDeviceTitle()

Colour Map

GINO will attempt to open a Double Buffered, RGBA GLX Visual providing true colour operation on the display. Most GLX extension will allow this even on 8 plane, indexed displays but the number of colours is obviously limited. 1024 entries in the GINO colour table are available to store colour definitions to use as colour indices.

On an 8 plane device, the GLX visual will attempt to share colours with the system colour map, possibly reducing the range of colours available for lighting. The user may choose to allocate a provate colour map of 256 colours by setting the number of colours to 256 through one of the following methods:

1) Setting an environment variable GLXCOLS to 256

2) Calling the routine gSetColourInfo(ndc,ndt)

Graphics Visibility

The current graphics window can be popped to the front of the display or pushed to the back of the display using the GINO routine gSetGraphicsVis().

e.g.

gSetGraphicsVis(GINVISIBLE)

Pushes the window to the back

gSetGraphicsVis(GVISIBLE)

Pops the window to the front

Character Fonts

The following hardware character fonts are accessible through the routine gSetCharFont(font) where font can be one of the following:

0 or 100	adobe-courier (default)
101	adobe-helvetica
102	adobe-times
103	adobe-ITC Avant Garde Gothic
104	adobe-ITC Lubalin Graph
105	adobe-New Century Schoolbook
106	adobe-ITC Souvenir
170	adobe-Symbol

The availability of the fonts is server dependent. The font enquiry routine gEnqHardFontList() can be used to enquire which fonts are available on the server being used. Where one of the hardware fonts listed above (except font 170) is not available a software emulation is provided with similar character proportions.

Bold and/or italic versions of fonts 100-106 can be selected using the routine gSetFontWeight(weight) where **weight**>0 and the routine gSetItalicAngle(angle) where 10≤**angle**≤20 degrees.

Segment Facilities

The GLX driver supports some of GINO's segment facilities in line with the OpenGL Display List facilities. Thus the following segment routines operate through this driver:

gOpenSeg()	Open segment
gCloseSeg()	Close segment
gDrawSeg()	Draw segment
gDeleteSeg()	Delete segment
gEnqSegHit()	Enquire segment at hit point
gInsertSegRef()	Insert reference to segment
gEnqSegAttribs()	Enquire segment attributes (existence only)

Note that a segment must exist prior to making a reference to it using gInsertSegRef() and segments are always visible and hit sensitive.

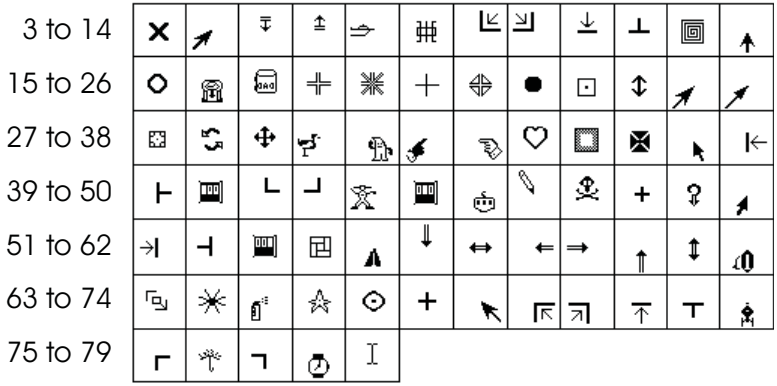
N.B. As hardware fonts are stored internally as segments by the GLX driver, any change to the current character/font settings will close the currently opened segment. Therefore it is essential that the required character/font attributes are set before a GINO segment is opened and are not altered within a segment.

Mouse Pointer Types

When the window is first initialised the pointer icon is set to be an 'hour glass' indicating that no interaction can be performed. This is changed when either gGetCursorEvent() is called, or one of the permitted event types is enabled. The icon used at this point can be set using the routine:

gSetCursorType(type,forcol,bakcol)

where **type** is the type number. Type -1 is an hour-glass, type 0 is a double cross, type 1 is a single small cross and type 2 is a large cross extending the whole height and width of the window. The following additional X cursor types are also available.



forcol and **bakcol** can also be used to set the cursor foreground and background colour numbers, remembering that cursors are drawn in XOR mode resulting in a different appearance to the GINO colour numbers on the actual display.

Graphics Cursor

In addition to the usual key values returned for a cursor hit, some special values of key are also returned as follows:

0	A call to gGetCursorEvent() has been made when the graphics window is iconized
1024	A resize event has taken place; the user should call gNewDrawing() and enquire the new size of the window (through gEnqDrawingLimits()) and redraw as required

While waiting for a key or mouse button press, any expose events that take place will cause an automatic update of the exposed area from the backing store. When gGetCursorEvent() is called, the input focus of the device is set to be the graphics window. When the key has been pressed, the focus is returned to the window which had it before gGetCursorEvent() was called.

GLX Hard Copy

A GLX driver specific routine is available to create a PostScript image file from the contents of the current window:

gGlxeps(mode,filename,ier)

Where **mode**=0 to create a monochrome or =1 to create a colour image, **filename** is the name of the PostScript file to be created and **ier** returns 0 if it has been successfullt created.

GLX Driver Error Messages

The following errors specific to this device driver may occur:-

At Initialization:

GINO ERROR 310 - Cannot Open Display
GLX Not Supported

No Conforming Visual

These messages are generated if the GLX driver cannot open the GLX extension library successfully, or that the display does not support any RGBA colour visual.

GINO ERROR 311 - Cannot Open Window

This is caused by an incorrect connection to the display or network problems. After outputting the error, the program will continue but no further output will take place.

At Initialization or gNewDrawing():

GINO ERROR 312 - Cannot obtain Window size

Character attribute setting:

GINO ERROR 313 - Cannot load required font

Iconized window at gCloseDevice():

GINO ERROR 314 - De-iconize Window

Regis Series Devices

Device Nominations

	Nomination Routine	Description of Device
A	gVt125()	Dec VT125 Terminal
B	gVt240()	Dec VT240 Terminal

C	gVt241()	Dec VT241 Terminal
D	gVt330()	Dec VT330 Terminal
E	gVt340()	Dec VT340 Terminal
F	gRainbo()	Dec Rainbow (running via POLY-REGIS)
G	gProf()	Dec Professional
H	gGigi()	Dec GIGI
I	gLn01()	Dec LN01 laser (running via PLOTLN)
J	gLcp01()	Dec LCP01 inkjet
K	gLcg01()	Dec LCG01 inkjet
L	gLj250()	Dec LJ250 inkjet (running via RETOS)

Device Characteristics

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**
Width (mm)	250	250	250	250	250	250	250	260	237	250	244	203
Height (mm)	156	149	149	149	149	149	149	140	150	149	183	297
X Resolution	768	800	800	800	800	800	800	768	2788	800	2872	2388
Y Resolution	480	476	476	476	476	476	476	414	1766	476	2154	3492
Colours / Pens	4	4	4	4Gr	16	4	4	8	1	4	8	256
Colour Palette	Dyn.	Dyn.	Dyn.	Dyn.	Dyn.	Dyn.	Dyn.	Fix.	Fix.	Fix.	Dyn.	Dyn.
Character Angles	x45°	x45°	x45°	x45°	x45°	x45°	x45°	x45°	x90°	x45°	x45°	x45°
Cursor Types	1	1	1	1	1	1	1	1	N	N	N	N
Cursor Positioning	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N
Event Types	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	N	N	N	N

[1] GKEYPRESS and GLOCATOR only

	Devices A to L
Broken Linestyles	4 (Dashed, dash-dotted, dotted, dash-dot-dotted)
Drawing Modes	No
Thick Lines	No
Line Ends	No
Arcs	YES
Symbols	No
Fonts	1
Character Sizes	16 multiples of 1.5mm square (LN01), 16 various (all others)
Italic Characters	0.0, +/-27.0, +/-45.0
Polygonal Filling	4 patterns (LN01), 8 patterns (LCG01), Solid and up to 256 vertices (all others)
Segments	No
Image Handling	No
Cursor Actions	No

	Devices A to L
Batch Updates	No
Clipping/Masking	No
Transform/Viewing	No
Shading	No
Lights	No
Texture Mapping	No
Auxiliary Drawing Areas	No
Window/Device Titling	No

Colours and Greyscales

On GIGI or LCG01, colours 3 & 9 revert to the default colour.
On LCG01, colour 10 will select the background colour of white.
On LJ250, colour 0 & 10 do not imply 'erase' mode and have no effect.
On VT125 & VT240, default colour numbers are 0,1,2,3.

Graphics Cursor

VT125, PROF, RAINBOW:
The cursor is positioned using the arrow-keys in conjunction with the auxiliary keypad. Pressing the arrow keys moves the cursor one pixel at a time. Pressing a number 'n' (where 'n'=1-9 or 0 for 10) on the auxiliary keypad followed by subsequent arrow keys, moves the cursor 'n' pixels at a time. Due to the nature of the software cursor, the following keys cannot be used to signify a cursor hit: A,B,C,D (ASCII values 65-68).

VT240, VT241, GIGI:
The crosshair cursor is positioned using the arrow-keys in conjunction with the <SHIFT> key. Pressing the arrow keys moves the cursor one pixel at a time. Pressing the <SHIFT> key at the same time, accelerates the cursor movement.

VT330, VT340:
The crosshair cursor can be positioned either with the arrow keys in conjunction with the <SHIFT> key as above, or with the mouse.

Outputting to the LN01 using PLOTLN

The drawing area given for the LN01 is when PLOTLN is used in landscape mode. This is not the default for PLOTLN and therefore the switch must be applied to the command:

PLOTLN/ORIENTATION=LANDSCAPE/REGIS

If PLOTLN is used in its default mode (portrait), then automatic scaling will occur and the resulting size of the plot will be unpredictable.

PLOTLN does not recognise the REGIS code for screen clear (gNewDrawing()) for giving a fresh sheet of paper, therefore it is recommended that single plots are created per program.

Outputting to the LJ250 using RETOS

The driver has been set up for the LJ250 with the background colour as white and the default colour as black and secondly the drawing area has been set up to an A4 media size rather than the assumed default of American A size. For this reason, the RETOS command should be as follows:

RETOS/NOREVERSE/SIZE=(8.,11.3) filename

The drawing area is by default in portrait mode.

VGA and SVGA PC Screens (LF90 only)

This document covers the VGA/SVGA driver supplied with the DOS version of GINO using the Lahey LF90 compiler.

Device Nominations

	Nomination Routine	Description of Device
A	gSvga(ixres,iyres,ncols)	SVGA using parameters (display mode > 255)
B	gSvgac()	SVGA using GINO.CON (display mode > 255)
C	gSvgam(imode)	SVGA using mode number (display mode > 255)
D	gMvga()	MVGA with colour monitor (display mode 19)
E	gVga()	VGA with colour monitor (display mode 18)
F	gVgam()	VGA with monochrome monitor (display mode 17)
G	gEga()	EGA with colour monitor (display mode 16)
H	gEgam()	EGA with monochrome monitor (display mode 15)
I	gCgac()	CGA with colour monitor (display mode 4)
J	gCga()	CGA with monochrome monitor (display mode 6)
K	gHcga()	Hercules monochrome (display mode 20)

Device Characteristics

	A,B,C	D	E	F	G	H	I	J	K
X Resolution	var	320	640	640	640	640	320	640	720
Y Resolution	var	200	480	480	350	350	200	200	348

Colours / Pens	16/256	256	16	0 & 1	16	0 & 1	4	0 & 1	0 & 1
Fonts	No	No	Yes	No	Yes	No	No	No	No
	Devices A to K								
Width (mm)	213.00								
Height (mm)	159.67								
Colour Palette	Dynamic								
Broken Linestyles	3								
Drawing Modes	GAND+GOR+GXOR - Not in MVGA								
Thick Lines	No								
Line Ends	No								
Arcs	Circles only								
Symbols	No								
Fonts	1								
Character Sizes	See Fonts								
Character Angles	No								
Italic Characters	No								
Filling	No								
Segments	No								
Image Handling	Output only								
Cursor Types	GPOINTER only								
Cursor Actions	None								
Cursor Positioning	No								
Event Types	GKEYPRESS, GLOCATOR, GMOVEMENT, GKEYRELEASE								
Batch Updates	No								
Clipping	No								
Transform/Viewing	No								
Shading	No								
Lights	No								
Texture Mapping	No								
Auxiliary Drawing Areas	No								
Window/Device Titling	No								

SVGA Nomination

To nominate a SVGA board as the current device, three routines are provided:

gSvga(ixres,iyres,ncols)

which selects the display by resolution and colour parameters

gSvgam(mode)

which selects the display by mode number

gSvgac()

which selects the display by resolution and colours set in the GINO configuration file GINO.CON. Three configuration variables are required to be set:

- SVGAXRES = horizontal resolution (default=640)
- SVGAYRES = vertical resolution (default=480)
- SVGACOLS = number of colours (default=16)

Among the possible combinations for VESA compatible boards are the following:

Resolution	Colours	Mode
640 x 400	256	256
640 x 480	256	257
800 x 600	16	258
800 x 600	256	259
1024 x 768	16	260
1024 x 768	256	261
1280 x 1024	16	262
1280 x 1024	256	263

It is possible, however, that either the display board or the monitor cannot be set in the requested mode. If so, an error message is printed and output is paused. Pressing a <CR> will continue processing and the board will revert to VGA mode.

Two additional modes are available as follows:

gSvgam(11)	SVGA, 16 colours
gSvgam(12)	SVGA, 256 colours

Colours

GINO colour 10 maps onto hardware colour 7 and vice versa. This is so that GINO colour 10 maps onto the default alpha text colour of white.

Fonts

1 font is available in multiples of 8 pixels.

Image Handling

Pixel values must be in the same range as the number of colours currently defined. Note that the same pixel array will occupy a different portion of the screen depending on the display mode because of the different number and size of pixels for each mode.

Cursor Types

The pointer is drawn by the hardware in colour numbers 15 for the arrow and 0 for the outline. To change the colour of the pointer, use gDefineRGB() to redefine either 15 or 0.

NB: As there is no XOR write operation in MVGA mode, the cursor cannot be erased properly when dragging it across the screen. Therefore an impression of the cursor is left wherever it is dragged to.

Cursor Control and Key Return

The cursor may be re-positioned using a mouse. The mouse will operate only if a mouse driver has been installed prior to running the program. The arrow keys can be used if no mouse driver is loaded. When a mouse is being used the arrows keys will return their usual key value, but a shifted arrow key will move the mouse position in an accelerating manner until the arrow key is released.

Any key can be used to signify a cursor hit with the corresponding ASCII value being returned. The joy-stick buttons return ASCII 84 (T).

Alpha Text

There is no control over the position of Fortran input/output, however gSetAlphaMode() will affect subsequent calls to the Lisk Output routine IOutString.

Device Termination or Suspension

The programmer has the option of termination or suspension when closing down the graphics output to the screen. The choice is provided through the routines gCloseDevice() and gSuspendDevice(). The routine gCloseDevice() clears all graphics output from the screen by returning the PC to TEXT MODE 3, while the routine gSuspendDevice() leaves the PC in the current graphics mode. In both cases no further graphics may be output to the screen unless the PC is nominated again. If a subsequent nomination of the same graphics mode occurs after suspending the device (with gSuspendDevice()), the new nomination will leave the screen in the same mode and will leave the hardware colour table unaltered since the previous nomination. The use of gSuspendDevice() is therefore recommended prior to a temporary nomination of an output device within a graphics application.
If the PC has been left in graphics mode, it can be reset using the command MODE 80 or MODE CO80

Error Messages

REQUESTED SVGA MODE UNAVAILABLE - VGA SELECTED

An SVGA mode has been requested that is unavailable on this board type. VGA will be selected instead. Try requesting a different resolution or number of colours in the SVGA call.

ERROR ATTEMPTING TO SET GRAPHICS MODE

An SVGA mode has been detected as available but when attempting to select it, the board has returned an error code and the program terminates. Try requesting a different resolution or number of colours in the SVGA call.

Windows (Microsoft) System

Device Nominations (C/C++)

	Nomination Routine	Description of device
A	gMwin(hInst,hPrevInst)	Opens a default window
B	gMwinw(hInst,hPrevInst,x,y,width,height)	Opens a window at the specified position and size
C	gMwinao(hInst,hPrevInst,hWnd,hDC)	Opens GINO using an existing bitmap
D	gMwinp (hInst,hPrevInst)	Opens a Windows printer for GINO output
E	gMwindp (hInst,hPrevInst)	Opens the default Windows printer for GINO output
F	status=gMwinpp(hInst, hPrevInst,mode,devname,dlen,filename,flen,n,prop)	Printing and Setup control for Windows printers
G	gGuiwin()	Window API graphics in GINOMENU

Device Nominations (F90)

	Nomination Routine	Description of device
A	gMwin	Opens a default window
B	gMwinw(x,y,width,height)	Opens a window at the specified position and size
C	gMwinao(hWnd,hDC)	Opens GINO using an existing bitmap
D	gMwinp	Opens a Windows printer for GINO output
E	gMwindp	Opens the default Windows printer for GINO output
F	status=gMwinpp(mode,devname,filename,n,prop)	Printing and Setup control for Windows printers
G	gGuiwin	Window API graphics in GINOMENU

Device Characteristics

	A	B	C	D	E	F	G
Default Width (of max)	3/4	variable	variable	Maximum	Maximum	Maximum	variable
Default Height (of max)	3/4	variable	variable	Maximum	Maximum	Maximum	variable
Image Handling (output, input, copy)	Yes	Yes	Yes	Printer Dependent	Printer Dependent	Printer Dependent	Yes[4]
Cursor Types*	Yes[1]	Yes[1]	Yes[1]	No	No	No	Yes[1]
Cursor Actions*	Yes[2]	Yes[2]	Yes[2]	No	No	No	Yes[2]
Event Types*	Yes[3]	Yes[3]	Yes[3]	No	No	No	No
Batch Updates*	Yes	Yes	Yes	No	No	No	Yes
Auxiliary Drawing Areas (up to 250 pairs)	Yes	Yes	Yes	No	No	No	Yes
Window/Device Titling	Yes	Yes	No	No	No	No	No
	Devices A to G						
Maximum Width	Variable						
Maximum Height	Variable						
Colours / Pens	Screen / printer dependent						
Colour Palette	<=256 colours = Dynamic, >256 = Direct						
Broken Linestyles	3						
Drawing Modes	GDEFAULT, GERASER, GNOT, GAND, GOR, GXOR and 11-19 hardware specific						
Thick Lines	Yes						
Line Ends	1						
Arcs	No						
Symbols	No						
Fonts	3 (100,101,102,150,151,170)						
Character Sizes	Any size (16 pseudo sizes)						
Character Angles	Any angle						

	A	B	C	D	E	F	G
Italic Characters	Yes						
Polygonal Filling	Multi-polygon, solid fill, 1024 vertices						
Segments	No						
Clipping	Yes (Windowing only)						
Transform/Viewing	No						
Shading	No						
Lights	No						
Texture Mapping	No						
Mouse Positioning	Yes						

* when using the gMwinao() nomination routine, the handle to the window must be provided for these features to be available

Yes[1] GHOURGLASS, GDEFAULT, GSMALLCROSS, GLARGECROSS, GX, GPOINTER, GLARGEX, GSTOP (Colour control only on GLARGECROSS)

Yes[2] GPOLYLINE, GRUBBERBAND GRUBBERBOX, GRUBBERSQUARE, GRUBBERELLIPSE,GRUBBERCIRCLE

Yes[3] GKEYPRESS, GLOCATOR, GMOVEMENT, GKEYRELEASE, GRESIZE, GMOUSEWHEEL

Yes[4] Input and copying is only available when the device is operating in direct colour mode (see gSetColourInfo())

Device Nomination & Usage

Seven nomination routines are available for using Microsoft Windows or Windows printers. When using the C routines, all except gGuiwin() require the handle to the current instance of the program (hInst) and, if available, the handle to the previous instance (hPrevInst) to be passed to the nomination routine as the first two parameters. In a C program these parameters are available in the WinMain() procedure. The nomination routines are described below:

gMwin

This nomination routine opens a standard window of 3/4 screen size at a default position. The device driver will update the window after every five hundred calls, upon a gFlushGraphics() / gCloseDevice() / gSuspendDevice(), before cursor or event routines are called, or upon a Windows paint event. The following examples show the nomination routine in the context of a program:

```
#include <windows.h>
#include <gino-c.h>
int PASCAL WinMain(HANDLE hInst,
HANDLE hPrevInst,
LPSTR lpszCmdParam, int nCmdShow)
{

/* Nominate the device */
gMwin(hInst, hPrevInst);

/* Use GINO routines here */

/* Close the device */
gCloseDevice();
}
```

```
Program Main
use gino_f90

! Nominate the device
call gMwin

! use GINO routines here

.

! Close the device
call gCloseDevice
Stop
End
```

gMwinw

gMwinw() is essentially the same as gMwin(), however the size & position of the window can be specified. x, y is the pixel position of the top left corner of the window (coordinate (0, 0) is the top left of the screen) and width, height are the width and the height of the required window in pixels. If width or height are less than or equal to zero the default size is used. If the requested size is larger than the screen then the size is automatically set to the same size as the screen.

gMwinao

This nomination routine is most useful when using this driver in a Windows programming environment, allowing for greater flexibility, but at the loss of some ease of use. Output can be directed to existing windows and/or device contexts using Windows handles passed to this routine by argument. Details are found in the section at the send of this driver description (see page 769).

gMwinp

This nomination routine will use a windows printer for output. A dialog box will be displayed upon nomination allowing the user to select a printer. The form of a call to this routine is the same as a call to gMwin().

gMwindp

This nomination routine will use the default windows printer for output. The form of a call to this routine is the same a call to gMwin() or gMwinp().

Print Status Detection

A function gMwinst() is provided with this driver to detect the status of a print job, used in conjunction with either gMwinp() or gMwindp(). gMwinst() can return the following values:

0	Print job OK
1	Print job cancelled
2	Could not find a default print device
3	Job cancelled while printing

status=gMwinpp

This nomination provides comprehensive control over the printer setup and printing of GINO output through the Windows printer interface. The additional arguments are:

mode

-2	Return current printer properties (returns status > 0)
-1	Restore default printer properties and return them (returns status > 0)
0	Open Printer Setup Window (returns status > 0)
1	Restore default printer properties and print (same as gMwindp())
2	Open Printer Dialogue and print (same as gMwinp())
3	Print using current/default printer

devname
Printer Device Name (input/output)

dlen
Length of devname (C/C++ only)

filename
Printer Output File name (in place of device) : used when PrinttoFile property = 1 in modes 2 and 3. If filename is blank when the PrinttoFile property is set, a file of the name <application>.prn is generated.

flen
Length of filename (C/C++ only)

n
Number of elements in prop array (rounded down to nearest even number)

prop
Integer array of pairs of <property> and <setting> (see table below). The setting values are used to change the current printer properties in modes 0, 2 and 3 but will be returned in all modes. Thus if prop contains the values: 7,2,1,2 the printer properties will be changed to landscape orientation and print 2 copies. If mode = -2 and prop contains the values: 2,0,8,0 the (device name,) paper size and print quality of the current printer will be returned. If prop[0]<=0 the first n/2 of the property flags and their values are returned.

Property	Property Description	Settings
1	No. of copies	Number of copies

2	Paper size (see Windows wingdi.h)	= 1, Letter = 5, Legal = 9, A4 = 256, User defined
3	Paper length	Length in tenths of mm
4	Paper width	Width in tenths of mm
5	Paper Source (see Windows wingdi.h)	= 1, Default (Upper) = 2, Lower = 4, Manual = 5, Envelope
6	Scale Factor	Scale/100 (ie 50 = half size)
7	Orientation	= 1, Portrait = 2, Landscape
8	DPI/Quality	> 0, DPI = -1, Draft = -2, Low = -3, Medium = -4, High Quality
9	Colour	= 1, Monochrome = 2, Colour
10	Duplex	= 1, Simple = 2, Duplex = 3, Vertical
11	TTopt	= 1, Bitmap = 2, Download (HP printers) = 3, Substitute (PostScript printers)
20	PrinttoFile	= -2, Disable and hide Print to File toggle = -1, Disable Print to File toggle = 0, Leave toggle as is (default) = 1, Select toggle and use **filename**

status

0	Printing in progress
1	Print Cancelled or not required
2	No default/current printer available

Return Status - If status > 0 the application should immediately call gCloseDevice() and re-nominate another GINO device (eg. gMwin() or gMwinw()).

gGuiwin

This nomination routine is only used in GINOMENU applications, where graphics frames are required to display GINO graphics in a Windows environment. Further details are found in the GINOMENU User Guide.

Window Size

If the gMwin() or gMwinw() routines have been used to nominate the device any changes the user of the application makes to the window size during operation will be handled by the driver. The driver will ensure that the picture is updated by redrawing from the backing store and scroll bars added to the window, if needed. However GINO is not able to take any further action until after gNewDrawing() is called. At this point the driver will resize the drawing limits to fit the window and these can be enquired by the application using the routine gEnqDrawingLimits() if the user wishes to re-scale the picture.

If the window size is changed by the user, this takes priority over the programmers request for a change through gSetDrawingLimits().

Window Title

The default window title of a GINO application is a string composed of the GINO version and license owner. This can be superceded in the following order of precedence:

1) GINO config variable MWINTITLE

2) System environment variable MWINTITLE

3) Application call to gSetDeviceTitle()

Colour Palette

When running a GINO application in one of the Windows 8bit (256) colour modes, the driver will, by default, create and use a dynamic colour palette (ndt=3) and the number of colours is set to the number of palette entries that Windows reports as being available. In this mode, an exact colour match may not always be possible as Windows will use the nearest match when no more palette entries are available. There may also be some re-painting problems when other high-colour applications are running (see below).

When running in one of the 16bit, 24bit or 32bit colour modes (>256 colours), the driver will operate in direct mode (ndt=4), but an internal GINO palette with 1024 entries is still available. In this mode, gDefineRGB() can still be used but will have no effect on graphics already drawn.

An application may alter this default situation by using the routine gSetColourInfo() to either reduce the number of palette entries (in dynamic colour mode) or change to another colour mode which may give better results when running with other applications. The routine gSetColourInfo() may also be used to increase the number of colours with printing devices where Windows reports that there is only one colour available but the printer is capable of producing greyscales.

In all modes, changes to colour 0 (GBACKGROUND) may not be seen unless followed by a call to gNewDrawing().

Drawing Modes

In addition to the GINO drawing modes 6-10, this driver implements a further nine pen types:

11	The pen is always black [Output = Black]
12	The pen is always white [Output = White]
13	The inverse of the GINO AND pen [Output = ~(Pen & Screen)]
14	The inverse of the pen colour [Output = ~Pen]
15	The combination of colours common to the pen and the inverse of the screen[Output = (~Screen) & Pen]
16	A combination of the screen and the inverse of the pen [Output = (~Pen) | Screen]
17	The inverse of the GINO OR pen [Output = ~(Pen | Screen)]
18	A combination of the pen & the inverse of the screen [Output = (~Screen) | Pen]
19	The inverse of the GINO XOR pen [Output = ~(Pen ^ Screen)]

Note that the effect of the different drawing modes varies according to the Windows screen mode that the application is operating. When the device is in indexed mode (ie. 256 colours), the resulting colour is determined from the bit operation on the colour indices of the pen and the screen, whereas when Windows is operating in Direct (True) colour mode (16bit, 24bit etc), the resulting colour is determined by the bit operation on the RGB values of the pen and screen.

Fonts

The default font used by the MWIN driver is a fixed pitch, modern font which normally maps to Courier. Other fonts available are selected by number using gSetCharFont() according to the following table:

Font Number	**Description**

0,100	Modern (fixed-pitch)
101	Swiss
102	Roman
150	Arial
151	Times New Roman
170	Symbol

Graphics Cursor

When gGetCursorEvent() is called, the input focus of the device is set to be the graphics window. When the key has been pressed, the focus is returned to the previous window before gGetCursorEvent() was called. This operation is not supported when using the gMwinao() nomination routine.

Events

The following event types can be set through the routine gAddEventType(intype): where **intype** can be one of the following:

GKEYPRESS	Key or Mouse button press
GLOCATOR	Screen position and Key/Mouse button press
GMOVEMENT	Pointer/Mouse movement
GKEYRELEASE	Key or Mouse button release
GRESIZE	Window resize
GMOUSEWHEEL	Mouse wheel movement

Multiple event types may be set concurrently. When the device has been nominated by gMwin() or gMwinw(), events are requested with the routine gWaitForEvent() with the resulting data for the event being returned through the routine gGetEventRecord() (see page 447).

Information concerning events in conjunction with the gMwinao() nomination routine are described in the Windows programming environment section below (see page 769). Events are only supported if using the gMwinao() nomination routine with a window handle.

Device Termination

When using either gMwin() or gMwinw(), the action at device termination is to halt GINO graphics and close the graphics window if gCloseDevice() has been called. If gSuspendDevice() has been called, the graphics window will remain on the screen and the program will pause at the call to gCloseGino(). The dialogue window will be closed at the same time as GINO is closed.

When using gMwinao(), at device termination it is the application programmers responsibility to release the bitmap device context.

Error Handling

When a GINO or other GINO library generates an error or warning message an error window is created and displayed at the bottom left of the screen. All subsequent error/warning messages are printed in this window. The user can alter the initial state of this window through the configuration variable NFERTR set in the configuration file GINO.CON as follows:

NFERTR	
= 0	Visible(Default)
= 1	Iconized
= 2	Hidden
> 2	Errors are sent to a file “APPLICATION”.ERR

Window Visibility

The current graphics and dialogue windows can be made visible/invisible using the routines gSetGraphicsVis() and gSetDialogueVis():

gSetGraphicsVis(gravis)	Make graphics window visible/invisible
gSetDialogueVis(diavis)	Make dialogue window visible/invisible

where gravis & diavis can be either GINVISIBLE or GVISIBLE to hide or show the window.

The routine gSetGraphicsVis() will have no effect when the device has been nominated by using the gMwinao() routine. Calls to gSetDialogueVis() apply in all cases.

Application Icon

The GINO icon will be assigned to all windows created through these nomination routines. An application can replace this with one of their choosing by adding an icon resource of the name “GinoIcon” to their application. A typical resource file should contain the line:

GinoIcon ICON myapp.ico

where myapp.ico is the name of the user supplied application icon file.

Interacting with other Windows Applications

GINO graphics sent to the screen can be copied onto the Windows clipboard by pressing <ALT> <PRINT SCREEN>. Transfer of graphics to other applications via OLE or DDE is not supported.

Error Messages

The WINDOWS driver may at one time or another popup a message box. These messages are mainly informational, but may effect the drivers behaviour. The messages are described below:

UNABLE TO CREATE A PALETTE. SOME COLOURS MAY LOOK WRONG.

When Windows memory or resources are low the WINDOWS driver will not be able to create an internal palette of colours. The program will still function, however the colours may not match the ones defined. Resolve this problem by closing one or more applications & restarting the GINO application. If the problem still persists, try restarting windows.

UNABLE TO CREATE THE MESSAGE WINDOW

Sometimes when an application has been improperly closed the dialogue window will still exist in memory, although the dialogue window may not be visible. The WINDOWS driver will not be able to use this instance or create a new instance, hence there will not be a dialogue window. Reasons for this can be programs crashing or being terminated part way through (i.e. by a debugger). The only solution to this problem is to restart windows.

Using Windows Driver in Windows Programming Environment

The GINO Windows driver can be used to direct GINO output to windows or device contexts that have been created through direct or indirect use of Windows API routines in any programming language. This enables GINO to be used in a wide variety of Windows programming environments or high level GUI systems as long as the user has access to the handle of the relevent object where the graphics is to be drawn. This is achieved by using the gMwinao() nomination routine and passing the handle(s) via its arguments.

There are three ways that this nomination routine can be called depending on the usage or handle(s) available:

i) Window handle only

```
gMwinao(hInst, hPrevInst,
         hWnd,NULL);
```

```
call gMwinao(hWnd,0)
```

When only the window handle of an existing window has been provided, the device driver will create a backing store (bitmap) and periodically update the window from the backing store (after every one hundred device driver calls). Windows paint events will not be handled automatically unless the mwinDefWindowProc() routine is used as the default window procedure, the next section (Handling Windows Events) explains how to achieve this.

Normally the whole window is used for drawing by the device driver, however if the window is resized the size of the drawing area will not change to fit the window until the device is next cleared with a call to gNewDrawing().

ii) Window and device context handle

```
gMwinao(hInst, hPrevInst,
         hWnd,hDC);
```

```
call gMwinao(hWnd,hDC)
```

When both the handle to the window and the handle to the device context are provided, the device driver will draw to the device context and periodically update the window (every one hundred device driver calls). See the next section (Handling Windows Events) for handling of Windows paint events.

The size of the bitmap in the device context provides the size of the drawing area. The drawing area will be displayed in the top left hand corner of the window. However, unlike the previous case a call to gNewDrawing() will not resize the drawing area to fit the window.

iii) gMwinao(hInst, hPrevInst, NULL, hDC)

```
gMwinao(hInst, hPrevInst,
         NULL,hDC);
```

```
call gMwinao(0,hDC)
```

If only the handle to a device context is provided the device driver will draw to this. The device driver will not be able to update the window, this must be handled by the applications's window procedure. The following section (Handling Window Events) explains how to handle updates in this case.

The best and most obvious place to nominate the Windows device, in this environment, is when processing the Windows WM_CREATE message in the application window procedure associated with the window. Similarly the device should be closed at the WM_DESTROY message using the GINO routne gCloseDevice(). Where a bitmap is created by the application for the purposes of drawing to, this too should be deleted at the WM_DESTROY message. See below for language specific programming examples.

Drawing to the desired window or device context can take place in one of three locations within such an application.

i) When creating the window - where a static picture is required this can be drawn immediately after nominating the device as long as repairing of this is catered for through automatic or manual handling of the WM_PAINT message.

ii) Within the WM_PAINT message code - although more expensive in processing time, utilizing a redraw function at this point ensures that the graphics is always up-to-date.

iii) Elsewhere in the application - graphics can be built up and/or modified at any point within an application again as long as repairing of this is catered for through automatic or manual handling of the WM_PAINT message.

The drawing of any graphics should be terminated by a call to gFlushGraphics() which will force an update of the window through a Window pait message (WM_PAINT).

Handling Windows Events (Visual Fortran only)

Where a Visual Fortran user has written code to handle windows and menu events using the Windows API directly, the Windows driver can be used to add graphics using the first of the gMwinao() cases described above. If the application wishes GINO to handle any Windows events the functions mwinDefWindowProc() or mwinDefMDIChildProc() should be used as shown in the skeleton window handling procedure below.

```
integer function MainWndProc(hWnd, mesg, wParam, lParam)

   use gdi32
   use user32
   use gino_f90
   integer *4 hWnd, mesg, wParam, lParam

   interface
   integer*4 function mwinDefWindowProc(hWnd, mesg, wParam, lParam)
!DEC$ ATTTRIBUTES C,ALIAS : '_mwinDefWindowProc' ::
mwinDefWindowProc
   integer*4 hWnd, mesg, wParam, lParam
   end function
   end interface

   select case (mesg)
      case (WM_CREATE)
         call gMwinao(hWnd, 0)
         return

      case (WM_DESTROY)
         call gCloseDevice
         call gCloseGino
         call PostQuitMessage(0)
         MainWndProc = 0
         return

      case DEFAULT
         MainWndProc = mwinDefWindowProc(hWnd, mesg, wParam, lParam)
   end select
end
```

Users are however recommended to use the GINOMENU library to greatly ease the task of creating Fortran GUI applications.

Handling Windows Events (C/C++ only)

The nomination routines gMwin() and gMwinw() use an internal window procedure provided by the device driver and automatically handle all Windows events. Programs should nominate the device, draw the graphics and then suspend or close the device, there is no need for any event handling code.

The gMwinao() nomination routine allows for the program, rather than the device driver, to control the window procedure. For the device driver to handle Windows events they must be passed on to the device driver, this is accomplished by calling mwinDefWindowProc() or mwinDefMDIChildProc() in the window procedure. The mwinDefWindowProc() routine takes the place of DefWindowProc() when using a standard window and mwinDefMDIChildProc() replaces DefMDIChildProc() when using the multiple document interface. The following example shows mwinDefWindowProc() in use:

```
/* The minimum window procedure for a GINO program using gMwinao()
*/
long FAR PASCAL MainWndProc(HWND hWnd, UINT message,
UINT wParam, LONG lParam)
{
     switch(message) {
     case WM_CREATE:
          /* Nominate the current window as the output device */
          gMwinao(hInst, hPrevInst, hWnd, NULL);
          gNewDrawing();
          return 0;

     case WM_DESTROY :
          /* Close device */
          gCloseDevice();
          PostQuitMessage(0);
          return 0;
     }

     /* Pass on unhandled messages (such as WM_PAINT) */
     return mwinDefWindowProc(hWnd, message, wParam, lParam);
}
```

The above example will handle window repaints when the gMwinao() nomination routine provides the driver with a handle to a window (when the hWnd parameter is not NULL). However, when the gMwinao() nomination routine does not provide the driver with a handle to a window the device driver cannot keep the window updated. In this case it is the application programmers responsibility to update the window using the contents of the device context. The following example shows a complete C program on how to accomplish this:

```
#include <windows.h>
#include <gino-c.h>

#  define ExportProc(x) (x)

int mwinRegisterClasses(void);
long WINAPI   mwinProc(HWND, UINT, UINT, LONG);

HINSTANCE hInst, hPrevInst;
HDC newdc;
HBITMAP bits,oldbits;
int quit=0;


int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow)
{
  HWND hwnd;
  MSG msg;

  hInst=hInstance;
  hPrevInst=hPrevInstance;

  mwinRegisterClasses();

  hwnd = CreateWindow("GINOProgram","Mwinao example",
              WS_OVERLAPPEDWINDOW,0,10,640,480,
              NULL,NULL,hInstance,NULL);
```

```
  ShowWindow(hwnd,SW_RESTORE);

  while (!quit) {
    GetMessage(&msg, NULL, 0, 0);
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return 0;

}


int mwinRegisterClasses(void)
{
   WNDCLASS wndclass;

  /* Load The GINO Icon */
  if (!hPrevInst) {
    wndclass.style         = CS_HREDRAW | CS_VREDRAW |
                             CS_SAVEBITS | CS_CLASSDC;
    wndclass.lpfnWndProc   = ExportProc(mwinProc);
    wndclass.cbClsExtra    = 0;
    wndclass.cbWndExtra    = 0;
    wndclass.hInstance     = hInst;
    wndclass.hIcon         = NULL;
    wndclass.hCursor       = NULL;
    wndclass.hbrBackground = GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName  = NULL;/* No class menu */
    wndclass.lpszClassName = "GINOProgram";
    return (RegisterClass(&wndclass));
  } else {
    return 1;
  }
}

/* The main window procedure */
long CALLBACK mwinProc(HWND hwnd, UINT message, UINT wParam,
                       LONG lParam) {
  HDC          hdc;
  PAINTSTRUCT  ps;
  GLIMIT limits = {10.0,50.0,10.0,50.0};

  switch (message) {

    /* Create window items */
    case WM_CREATE:
      hdc = GetDC(hwnd);
      newdc = CreateCompatibleDC(hdc);
      bits = CreateCompatibleBitmap(hdc,630,450);
      oldbits = SelectObject(newdc, bits);
      ReleaseDC(hwnd, hdc);

      qMwinao(hInst, hPrevInst, NULL, newdc);

      qNewDrawing();

      qFillRect(GSOLID,2,&limits);

      qSetLineColour(GBLUE);
      qMoveTo2D(0.0,0.0);
      qDrawLineTo2D(50.0,50.0);
      qFlushGraphics();
      return 0;
```

```
        /* Repaint client area */
        case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps);
          BitBlt(hdc,0,0,630,450,newdc,0,0,SRCCOPY);
          EndPaint (hwnd, &ps);
          return 0;

        /* Window has been killed */
        case WM_DESTROY :
          gCloseDevice();
          gCloseGino();

          SelectObject(newdc, oldbits);
          DeleteObject(bits);
          DeleteDC(newdc);
          PostQuitMessage(0);
          quit=1;
          return 0;
      }

      /* Pass on unhandled events */
      return mwinDefWindowProc(hWnd, message, wParam, lParam);
    }
```

NOTE: The Windows message number 0x7FFF is reserved for use by this driver. Your program should not use this message number for its own purposes.

When using the gMwinao() nomination routine with GINO events, the device must be nominated with a handle to a window. Events do not need to be requested with gWaitForEvent(), instead a WM_GINOEVENT message should be added to the event loop & processed in a similar way to the WM_COMMAND message. The word parameter (wParam) passed to the window procedure will contain the GINO event type. The following extract shows a C program with a button to start events and a button to stop the events. When the events are enabled a line will be drawn following the cursor:

```
/* A window procedure for a GINO program using GINO events */
long FAR PASCAL MainWndProc(HWND hWnd, UINT message,
UINT wParam, LONG lParam)
{
     GEVEREC event;
     switch(message) {
     case WM_CREATE:
           /* Create a start and a stop button */
          CreateWindow("button", "Start",
               WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, 0, 0, 60, 30,
               hWnd, (HMENU)IDM_STARTEVENTS, hInst, NULL);
          CreateWindow("button", "End",
               WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, 60, 0, 60, 30,
               hWnd, (HMENU)IDM_ENDEVENTS, hInst, NULL);
           /* Nominate the current window as the output device */
          gMwinao(hInst, hPrevInst, hWnd, NULL);
          gNewDrawing();
          return 0;
```

```
            case WM_COMMAND:
                 switch(wParam) {
                 /* Switch on events */
                 case IDM_STARTEVENTS:
                      gAddEventType(GMOVEMENT);
                      break;
                 /* Switch off events */
                 case IDM_ENDEVENTS:
                      gRemoveEventType(GMOVEMENT);
                      break;
                 default:
                      break;
                 }
                 return 0;
            case WM_GINOEVENT:
                 gGetEventRecord(wParam,&event);
                 switch(wParam) {
                 /* If a GMOVEMENT event then draw a line */
                 case GMOVEMENT:
                      gDrawLineTo2D(event.pos.x, event.pos.y);
                      gFlushGraphics();
                      break;
                 default:
                      break;
                 }
                 return 0;
            case WM_DESTROY:
                 /* Close device */
                 gCloseDevice();
                 PostQuitMessage(0);
                 return 0;
            }
            /* Pass on unhandled messages (such as WM_PAINT) */
            return mwinDefWindowProc(hWnd, message, wParam, lParam);
       }
```

Using Windows driver in a Visual Basic Environment

GINO can make good use of the Visual Basic picture box control for its output. Provided the AutoRedraw property of the picture control is set to True there is no need to handle the paint events as they will be handled automatically by Visual Basic. However, the picture controls Refresh method will need to be called after drawing if an immediate update is required.

The best place to nominate a device is in the forms load procedure. The handle to the programs instance is provided by the routine gGetHinstance().The following example nominates a picture box control, called Picture1, as the output device. The Picture1 object must already exist in the form:

```
Sub Form_Load ()
     Dim hInst As Long
     gMwinao 0, Picture1.hDC
End Sub
```

The most appropriate place to close the device is in the forms unload procedure:

```
Sub Form_Unload (Cancel As Integer)
     gCloseDevice
End Sub
```

After drawing to the picture box using GINO routines a call must be made to the objects Refresh method, the following example will draw a square in the nominated picture box when the button Command1 is pressed (clicked), but will not be displayed until Picture1.Refresh is called:

```
Sub Command1_Click ()
     gMoveTo2D 10, 10
     gDrawLineTo2D 110, 10
     gDrawLineTo2D 110, 110
     gDrawLineTo2D 10, 110
     gDrawLineTo2D 10, 10
     Picture1.Refresh
End Sub
```

Alternatively a complete program can be contained in the form as follows:

```
Private Sub Form_Load()
Dim hInst As Long
Dim I As Long
Dim XPAP As Single
Dim YPAP As Single
Dim ipap As Long
Dim paper As GDIM
Dim papertype As Long

gOpenGino
Rem Open The GINO Device
gMWINAO 0, Picture1.hdc
gEnqDrawingLimits paper, papertype
Rem Colours
gDefineRGB 0, 1#, 0#, 0.75
gNewDrawing
gSetLineColour 4
gMoveTo2D 10#, 10#
gDrawLineTo2D paper.XPAP / 2#, paper.YPAP / 2#
ggAddGrid GNONE, GTICKS, GANNOTATION, GANNOTATION
gCloseDevice
gCloseGino
End Sub
```

All calls to GINO routines have to be made via a Declarations Module which details the interface between VB and the GINO DLL. Declarations are provided for every GINO, GINOGRAF and GINOSURF routine in the files *vbd.bas, but a subset just for the above program would look as follows:

```
Attribute VB_Name = "Module1"
' Direct GINO declarations
Declare Sub gOpenGino Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GOPENGINO@0" ()
Declare Sub gMWINAO Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GMWINAO@8" (ByRef hwnd As Long, ByRef hdc As
Long)
Declare Sub gEnqDrawingLimits Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GENQDRAWINGLIMITS@8" (ByRef paper As GDIM, ByRef
papertype As Long)
Declare Sub gCloseDevice Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GCLOSEDEVICE@0" ()
Declare Sub gCloseGino Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GCLOSEGINO@0" ()
Declare Sub gMoveTo2D Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GMOVETO2D@8" (ByRef X As Single, ByRef Y As
Single)
Declare Sub gDrawLineTo2D Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GDRAWLINETO2D@8" (ByRef X As Single, ByRef Y As
Single)
Declare Sub gDefineRGB Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GDEFINERGB@16" (ByRef I As Long, ByRef X As
Single, ByRef Y As Single, ByRef Z As Single)
Declare Sub gSetLineColour Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GSETLINECOLOUR@4" (ByRef I As Long)
Declare Sub gNewDrawing Lib "ginlibdg.dll" Alias
"_GINO_F90_mp_GNEWDRAWING@0" ()
Declare Sub ggAddGrid Lib "ggrafdg.dll" Alias
"_GRAF_F90_mp_GGADDGRID@16" (ByRef style1 As Long, ByRef style2
As Long, ByRef anx As Long, ByRef anyy As Long)
```

To send the graphics to a printer, either the GINO routine gMwinp can be called and the graphics calls would need to be called again, or you can print directly from the Picture1 control as follows:

```
Private Sub Command4_Click()
  Printer.PaintPicture Picture1.Image, 0 0
  Printer.EndDoc
  End Sub
```

Windows OpenGL (Microsoft) System

Device Nominations (C/C++)

	Nomination Routine	Description of device
A	gWogl(hInst,hPrevInst)	Opens a default window
B	gWoglw(hInst,hPrevInst,x,y,width,height)	Opens a window at the specified position and size
C	gWoglao(hInst,hPrevInst,hWnd,hDC)	Opens GINO using an existing bitmap

D	gWoglp (hInst,hPrevInst)	Opens a Windows printer for GINO output
E	gWogldp (hInst,hPrevInst)	Opens the default Windows printer for GINO output
F	status=gWoglpp(hInst, hPrevInst,mode,devname,dlen,filename,flen,n,prop)	Printing and Setup control for Windows printers
G	gOglwin()	OpenGL graphics in GINOMENU

Device Nominations (F90)

	Nomination Routine	**Description of device**
A	gWogl	Opens a default window
B	gWoglw(x,y,width,height)	Opens a window at the specified position and size
C	gWoglao(hWnd,hDC)	Opens GINO using an existing bitmap
D	gWoglp	Opens a Windows printer for GINO output
E	gWogldp	Opens the default Windows printer for GINO output
F	status=gWoglpp(mode,devname,filename,n,prop)	Printing and Setup control for Windows printers
G	gOglwin	OpenGL graphics in GINOMENU

Device Characteristics

	A	**B**	**C**	**D**	**E**	**F**	**G**
Default Width (of max)	3/4	variable	variable	Maximum	Maximum	Maximum	variable
Default Height (of max)	3/4	variable	variable	Maximum	Maximum	Maximum	variable
Image Handling (output, input, copy)	Yes	Yes	Yes	Printer Dependent	Printer Dependent	Printer Dependent	Yes
Cursor Types*	Yes[1]	Yes[1]	Yes[1]	No	No	No	Yes[1]
Cursor Actions*	Yes[2]	Yes[2]	Yes[2]	No	No	No	Yes[2]
Event Types*	Yes[3]	Yes[3]	Yes[3]	No	No	No	No
Batch Updates*	Yes	Yes	Yes	No	No	No	Yes
Auxiliary Drawing Areas (up to 250 pairs)	Yes	Yes	Yes	No	No	No	Yes
Window/Device Titling	Yes	Yes	No	No	No	No	No
	Devices A to G						
Maximum Width	Variable						
Maximum Height	Variable						
Colours / Pens	Screen / printer dependent						
Colour Palette	Direct						
Broken Linestyles	16						
Drawing Modes	None						
Thick Lines	Yes						
Line Ends	1						
Arcs	No						

	A	B	C	D	E	F	G
Symbols	No						
Fonts	3 (100,101,102,150,151,170)						
Character Sizes	Any size (16 pseudo sizes)						
Character Angles	Any angle						
Italic Characters	Yes						
Polygonal Filling	Multi-polygon, solid fill, 1024 vertices						
Segments	Yes						
Clipping	Yes (Windowing only)						
Transform/Viewing	Yes						
Shading	Yes						
Lights	Yes						
Texture Mapping	Yes						
Mouse Positioning	Yes						

* when using the gWoglao() nomination routine, the handle to the window must be provided for these features to be available

Yes[1] GHOURGLASS, GDEFAULT, GSMALLCROSS, GLARGECROSS, GX, GPOINTER, GLARGEX, GSTOP (Colour control only on GLARGECROSS)

Yes[2] GPOLYLINE, GRUBBERBAND GRUBBERBOX, GRUBBERSQUARE, GRUBBERELLIPSE,GRUBBERCIRCLE

Yes[3] GKEYPRESS, GLOCATOR, GMOVEMENT, GKEYRELEASE, GRESIZE, GMOUSEWHEEL

Device Nomination & Usage

The Windows OpenGL driver operates under any of the Microsoft Windows environments (Windows 95,98, NT and 2000) using a combination of the Windows API and the OpenGL API to provide a fully functional 3D GINO driver. The driver requires access to OPENGL32.DLL and GLU32.DLL.

Many of the features provided by this driver are the same as the standard Windows driver to which several sections below will refer.

Seven nomination routines are available for using the Windows OpenGL driver. When using the C routines, all except gOglwin() require the handle to the current instance of the program (hInst) and, if available, the handle to the previous instance (hPrevInst) to be passed to the nomination routine as the first two parameters. In a C program these parameters are available in the WinMain() procedure. The nomination routines are described below:

gWogl

This nomination routine opens a standard window of 3/4 screen size at a default position. The device driver will only update the screen upon a call to gFlushGraphics() / gCloseDevice() / gSuspendDevice(), before cursor or event routines are called, or upon a Windows paint event. The following examples show the nomination routine in the context of a program:

```
#include <windows.h>
#include <gino-c.h>
int PASCAL WinMain(HANDLE hInst,
HANDLE hPrevInst,
LPSTR lpszCmdParam, int nCmdShow)
{

/* Nominate the device */
gWogl(hInst, hPrevInst);

/* Use GINO routines here */

/* Close the device */
gCloseDevice();
}
```

```
Program Main
use gino_f90

! Nominate the device
call gWogl

! use GINO routines here

.

! Close the device
call gCloseDevice
Stop
End
```

gWoglw

gWoglw() is essentially the same as gWogl(), however the size & position of the window can be specified. x, y is the pixel position of the top left corner of the window (coordinate (0, 0) is the top left of the screen) and width, height are the width and the height of the required window in pixels. If width or height are less than or equal to zero the default size is used. If the requested size is larger than the screen then the size is automatically set to the same size as the screen.

gWoglao

This nomination routine allows for greater flexibility, but at the loss of some ease of use. Existing windows and device contexts can be used for output enabling GINO to send its output to other applications, however a device context should always contain a valid bitmap. Usage of this routine in F90, C and Visual Basic is the same as the gMwinao() routine in the standard Windows driver. Refer to this section for further information (see page 769).

gWoglp

This nomination routine will use a windows printer for output. A dialog box will be displayed upon nomination allowing the user to select a printer. The form of a call to this routine is the same as a call to gWogl().

gWogldp

This nomination routine will use the default windows printer for output. The form of a call to this routine is the same a call to gWogl() or gWoglp().

Print Status Detection

A function gWoglst() is provided with this driver to detect the status of a print job, used in conjunction with either gWoglp() or gWogldp(). The function gWoglst() can return the following values:

0	Print job OK
1	Print job cancelled
2	Could not find a default print device
3	Job cancelled while printing

status=gWoglpp

This nomination provides comprehensive control over the printer setup and printing of GINO output through OpenGL and the Windows printer interface. The additional arguments are:

mode

-2	Return current printer properties (returns status > 0)
-1	Restore default printer properties and return them (returns status > 0)
0	Open Printer Setup Window (returns status > 0)
1	Restore default printer properties and print (same as gWogldp())
2	Open Printer Dialogue and print (same as gWoglp())
3	Print using current/default printer
4	Print to BMP file (default = 800x600 pixels)

devname

Printer Device Name (input/output)

dlen

Length of devname (C/C++ only)

filename

Printer Output File name (in place of device) : used when PrinttoFile property = 1 in modes 2 and 3. If filename is blank when the PrinttoFile property is set, a file of the name <application>.prn is generated. Also used in mode 4 for BMP filename. In this mode, if the filename is blank <application>.bmp is generated.

flen

Length of filename (C/C++ only)

n

Number of elements in prop array (rounded down to nearest even number)

prop

Integer array of pairs of <property> and <setting> (see table below). The setting values are used to change the current printer properties in modes 0, 2 and 3 but will be returned in all modes. Thus if prop contains the values: 7,2,1,2 the printer properties will be changed to landscape orientation and print 2 copies. If mode = -2 and prop contains the values: 2,0,8,0 the (device name,) paper size and print quality of the current printer will be returned. If prop[0]<=0 the first n/2 of the property flags and their values are returned.

Property	Property Description	Settings
1	No. of copies	Number of copies
2	Paper size (see Windows print.h)	= 1, Letter = 5, Legal = 9, A4 = 256, User defined
3	Paper length	Output length (in tenths of mm for printers or pixels for BMP files)
4	Paper width	Output width (in tenths of mm for printers or pixels for BMP files)
5	Paper Source (see Windows print.h)	= 1, Default (Upper) = 2, Lower = 4, Manual = 5, Envelope
6	Scale Factor	Scale/100 (ie 50 = half size)
7	Orientation	= 1, Portrait = 2, Landscape
8	DPI/Quality	> 0, DPI = -1, Draft = -2, Low = -3, Medium = -4, High Quality
9	Colour	= 1, Monochrome = 2, Colour
10	Duplex	= 1, Simple = 2, Duplex = 3, Vertical
11	TTopt	= 1, Bitmap = 2, Download (HP printers) = 3, Substitute (PostScript printers)
20	PrinttoFile	= -2, Disable and hide Print to File toggle = -1, Disable Print to File toggle = 0, Leave toggle as is (default) = 1, Select toggle and use **filename**

status

0	Printing in progress
1	Print Cancelled or not required

2 No default/current printer available

Return Status - If status > 0 the application should immediately call gCloseDevice() and re-nominate another GINO device (eg. gWogl() or gWoglw())

gOglwin

This nomination routine is only used in GINOMENU applications, where graphics frames are required to display GINO graphics in a Windows OpenGL environment. Further details are found in the GINOMENU User Guide.

Window Size

If the gWogl() or gWoglw() routines have been used to nominate the device any changes the user of the application makes to the window size during operation will be handled by the driver. The driver will ensure that the picture is updated by redrawing from the backing store and scroll bars added to the window, if needed. However GINO is not able to take any further action until after gNewDrawing() is called. At this point the driver will resize the drawing limits to fit the window and these can be enquired by the application using the routine gEnqDrawingLimits() if the user wishes to re-scale the picture.

If the window size is changed by the user, this takes priority over the programmers request for a change through gSetDrawingLimits().

Window Title

The default window title of a GINO application is a string composed of the GINO version and license owner. This can be superceded in the following order of precedence:

1) GINO config variable WOGLTITLE

2) System environment variable WOGLTITLE

3) Application call to gSetDeviceTitle()

Depth Buffer

The accuracy of hidden surface removal depends on the size of depth buffer in the OpenGL driver. By default this is set to be 32 bits (giving the highest quality output), but some graphics hardware will operate more efficiently using a smaller depth buffer (at the cost of lower quality output). The OPENGL32 DLL that is used by the WOGL driver always provides a 32bit option, but some hardware enhanced modes only work with 24bit or even 16bit depth buffers.

The default size of the depth buffer can be modified using the environment variable or GINO config variable, WOGLDEPTH.

Colour Palette

This driver only operates when windows is running in one of the 16bit, 24bit or 32bit colour modes (>256 colours). The driver is therefore always operating in direct mode (ndt=4), but an internal GINO palette with 1024 entries is still available. In this mode, gDefineRGB() can still be used but will have no effect on graphics already drawn.

Changes to colour 0 (the background colour) may not be seen unless followed by a call to gNewDrawing().

Fonts

The default font used by the Wogl driver is a fixed pitch, modern font which normally maps to Courier. Other fonts available are selected by number using gSetCharFont() according to the following table:

Font Number	Description
0,100	Modern (fixed-pitch)
101	Swiss
102	Roman
150	Arial
151	Times New Roman
170	Symbol

Segment Facilities

The Windows OpenGL driver supports some of GINO's segment facilities in line with the OpenGL Display List facilities. Thus the following segment routines operate through this driver:

gOpenSeg()	Open segment
gCloseSeg()	Close segment
gDrawSeg()	Draw segment
gDeleteSeg()	Delete segment
gEnqSegHit()	Enquire segment at hit point
gInsertSegRef()	Insert reference to segment
gEnqSegAttribs()	Enquire segment attributes (existence only)

Note that a segment must exist prior to making a reference to it using gInsertSegRef() and segments are always visible and hit sensitive.

N.B. As hardware fonts are stored internally as segments by the WOGL driver, any change to the current character/font settings will close the currently opened segment. Therefore it is essential that the required character/font attributes are set before a GINO segment is opened and are not altered within a segment.

Graphics Cursor

When gGetCursorEvent() is called, the input focus of the device is set to be the graphics window. When the key has been pressed, the focus is returned to the previous window before gGetCursorEvent() was called. This operation is not supported when using the gWoglao() nomination routine.

Events

The following event types can be set through the routine gAddEventType(intype) where **intype** can be one of the following:

GKEYPRESS	Key or Mouse button press
GLOCATOR	Screen position and Key/Mouse button press
GMOVEMENT	Pointer/Mouse movement
GKEYRELEASE	Key or Mouse button release
GRESIZE	Window resize
GMOUSEWHEEL	Mouse wheel movement

Multiple event types may be set concurrently. When the device has been nominated by gWogl() or gWoglw(), events are requested with the routine gWaitForEvent() with the resulting data for the event being returned through the routine gGetEventRecord() (see page 447).

Information concerning events in conjunction with the gWoglao() nomination routine are described in the Windows programming environment section (see page 769). Events are only supported if using the gWoglao() nomination routine with a window handle.

Device Termination

When using either gWogl() or gWoglw(), the action at device termination is to halt GINO graphics and close the graphics window if gCloseDevice() has been called. If gSuspendDevice() has been called, the graphics window will remain on the screen and the program will pause at the call to gCloseGino(). The dialogue window will be closed at the same time as GINO is closed.

When using gWoglao(), at device termination it is the application programmers responsibility to release the bitmap device context.

Error Handling

When a GINO or other GINO library generates an error or warning message an error window is created and displayed at the bottom left of the screen. All subsequent error/warning messages are printed in this window. The user can alter the initial state of this window through the configuration variable NFERTR set in the configuration file GINO.CON as follows:

NFERTR

= 0	Visible(Default)
= 1	Iconized
= 2	Hidden
> 2	Errors are sent to a file “APPLICATION”.ERR

Window Visibility

The current graphics and dialogue windows can be made visible/invisible using the routines gSetGraphicsVis() and gSetDialogueVis():

gSetGraphicsVis(gravis)	Make graphics window visible/invisible
gSetDialogueVis(diavis)	Make dialogue window visible/invisible

where **gravis** & **diavis** can be either GINVISIBLE or GVISIBLE to hide or show the window.

The routine gSetGraphicsVis() will have no effect when the device has been nominated by using the gWoglao() routine. Calls to gSetDialogueVis() apply in all cases.

Application Icon

The GINO icon will be assigned to all windows created through these nomination routines. An application can replace this with one of their choosing by adding an icon resource of the name “GinoIcon” to their application. A typical resource file should contain the line:

```
GinoIcon        ICON      myapp.ico
```

where myapp.ico is the name of the user supplied application icon file.

Interacting with other Windows Applications

GINO graphics sent to the screen can be copied onto the Windows clipboard by pressing <ALT> <PRINT SCREEN>. Transfer of graphics to other applications via OLE or DDE is not supported.

Error Messages

The WINDOWS driver may at one time or another popup a message box. These messages are mainly informational, but may effect the drivers behaviour. The messages are described below:

UNABLE TO CREATE THE MESSAGE WINDOW

Sometimes when an application has been improperly closed the dialogue window will still exist in memory, although the dialogue window may not be visible. The WINDOWS driver will not be able to use this instance or create a new instance, hence there will not be a dialogue window. Reasons for this can be programs crashing or being terminated part way through (i.e. by a debugger). The only solution to this problem is to restart windows.

Using Windows OpenGL Driver in Windows Programming Environment

The GINO Windows OpenGL driver can be used to direct GINO output to windows or device contexts that have been created through direct or indirect use of Windows API routines in any programming language. This enables GINO to be used in a wide variety of Windows programming environments or high level GUI systems as long as the user has access to the handle of the relevent object where the graphics is to be drawn.

This is achieved by using the gWoglao() nomination routine and passing the handle(s) via its arguments and is exactly comparable with the standard Windows driver as described in that section (see page 769).

References to gMwinao() should be replaced by gWoglao() and where the device driver is to handle Windows events they must be passed to the functions WoglDefWindowProc() or WoglDefMDIChildProc() as appropriate.

The only exception is when handling a WM_PAINT message in an application window procedure, the routine SwapBuffers() must be called prior to updating the screen as shown below:

```
/* window procedure to update screen from device context */
long FAR PASCAL MainWndProc(HWND hWnd, UINT message,UINT wParam,
LONG lParam) {
     PAINTSTRUCT ps;
     switch(message) {
     case WM_CREATE:
          /* Create a device context to store the picture,
             the device context contains a bitmap of
             400 by 400 pixels */
          hdc = GetDC(hWnd);
          newdc = CreateCompatibleDC(hdc);
          bits = CreateCompatibleBitmap(hdc, 400, 400);
          oldbits = SelectObject(newdc, bits);
          ReleaseDC(hWnd, hdc);
          /* Nominate device */
          gWoglao(hInst, hPrevInst, NULL, newdc);
          /* Clear the device */
          gNewDrawing();
          return 0 ;

     case WM_PAINT:
          /* Paint the window */
          hdc = BeginPaint(hWnd, &ps) ;
          /* swap buffers to flush all graphics to screen */
          SwapBuffers(hdc);
          /* Copy GINO output to the window at x=10, y=10 */
          BitBlt(hdc, 10, 10, 400, 400, newdc, 0, 0, SRCCOPY);
          EndPaint(hWnd, &ps);
          return 0;

     case WM_DESTROY :
          /* Close device */
          gCloseDevice();
          gCloseGino();
          /* Free bitmap & device context */
          SelectObject(newdc, oldbits);
          DeleteObject(bits);
          DeleteDC(newdc);
          PostQuitMessage(0);
          return 0;
     }

     /* Pass on unhandled events */
     return WoglDefWindowProc(hWnd, message, wParam, lParam);
}
```

X Windows System

Device Nominations

	Nomination Routine	Description of Device
A	gXwin()	Default X-Window
B	gXwinw(pause,clear,xp,yp,width,height)	User-defined X-Window
C	gXwinao(window_id,pixmap_id)	Opens GINO using an existing bitmap

Device Characteristics

	A	B	C
Default Width	$^2/_3$ Maximum	User Defined	Application Dependent
Default Height	$^2/_3$ Maximum	User Defined	Application Dependent
	Devices A,B and C		
Maximum Width	Server Dependent		
Maximum Height	Server Dependent		
Colours / Pens	Server Dependent (Default = 64)		
Colour Palette	Server Dependent		
Broken Linestyles	No		
Drawing Modes	GERASER, GNOT, GAND, GOR, GXOR		
Thick Lines	Yes		
Line Ends	3		
Arcs	Yes		
Symbols	No		
Fonts	Server Dependent		
Character Sizes	Server Dependent		
Character Angles	Server Dependent		
Italic Characters	Server Dependent		
Polygonal Filling	Single Polygons & Solid only		
Segments	No		
Image Handling	All Functions		
Cursor Types	GHOURGLASS,GDEFAULT, GSMALLCROSS, GLARGECROSS, GX, GPOINTER & 74 X specific types		
Cursor Actions	GPOLYLINE, GDEFAULT, GRUBBERBAND, GRUBBERBOX, GRUBBERSQUARE, GRUBBERELLIPSE, GRUBBERCIRCLE		
Cursor Positioning	No		
Event Types	GKEYPRESS, GLOCATOR, GMOVEMENT, GKEYRELEASE, GRESIZE, GPOINTERLEAVING, GPOINTERENTERING		
Batch Updates	Yes		
Clipping	Yes		

Transform/Viewing	No
Shading	No
Lights	No
Texture Mapping	No
Auxiliary Drawing Areas	Yes (Up to 50 pairs)
Window/Device Titling	Yes
Mouse Positioning	Yes

Linking with Xlib Library

To satisfy all outstanding references made by this driver, the XLIB library needs to be referenced in your link statement:

UNIX:

f90 -o myprog myprog.o -lgino-f90 -lX11

Some UNIX systems may require extra libraries to satisfy further references from the X11 library. e.g.

f90 -o myprog myprog.o -lgino-f90 -lX11 -lbsd

Device Nomination

Three nomination routines are available with X-windows:

gXwin()

This nomination routines opens a standard window of 2/3 screen size in the top left-hand corner.

gXwinw(pause,clear,xp,yp,width,height)

This routine allows the user to set the position, size and action at program termination where:

pause

= 0	specifies there is no pause at gCloseDevice() (default)
= 1	specifies that the process will pause with a prompt at gCloseDevice()
= 2	specifies that the process will pause (without prompt) at gCloseDevice()

clear

(Not Used)

xp,yp

specify the pixel position of the top left of the window (default = 0,0)

width,height

specify the pixel width and height of the window (default = 2/3 of max.)

The graphics window, however will always be cleared at the end of the program.

The nomination routine gXwin() is equivalent to calling gXwinw() as follows:

gXwinw(0,0,0,0,width,height)

where width,height are equivalent to two-thirds the dimensions of the full drawing area.

gXwinao(window_id,pixmap_id)

This routine allows GINO to draw to an existing window where window_id is the identifier of the external window and pixmap_id is the identifier of a pixmap the same size as the external window. If window_id or pixmap_id have a NULL value then they will be created by GINO.

The normal operation of the X-Windows driver is to open a window, map it, clear it and set up a colour table. A pixmap is also created to act as a backing store to enable the repainting of exposed regions.

When using gXwinao(), the window and pixmap are not created, but their attributes are obtained and used by GINO's initialization procedures. The window is not blanked and the colour map is not overwritten.

The window set up externally will need to be mapped before entering GINO.

If external drawing is to be used within the window, a pixmap will need to be created to allow damage repair on the window. To enable this all drawing must be done to the pixmap as well as to the window so that they are a copy of each other. Within the GINO gWaitForEvent()/gGetCursorEvent() loop, expose events on the window are trapped and used to trigger XCopyArea commands to repaint the window. If no external drawing is to be done, set pixmap_id to NULL and GINO will create and maintain the pixmap until gCloseDevice().

Window & Pixmap Identifiers

The identifiers of the primary window and pixmap can be obtained though a special device driver dependent routine:

gXwinid(window_id, pixmap_id)

These values may be passed to an appropriate X routine requiring such information.

Window Size

If the user of the application changes the window size during operation, the driver will ensure that the picture is updated by redrawing from the backing store, but GINO is not able to take any further action until after a gNewDrawing() is called. At this point the programmer can re-enquire the window size using gEnqDrawingLimits() and take the appropriate action.

If the window size is changed by the user, this takes priority over the programmers request for a change through gSetDrawingLimits().

Window Title

The default window title of a GINO application is a string composed of the GINO version and license owner. This can be superceded in the following order of precedence:

1) GINO config variable XWINTITLE

2) System environment variable XWINTITLE

3) Application call to gSetDeviceTitle()

Colour Map

GINO will open each window with the same colour attributes as the default display visual whether it be monochrome, pseudo, dynamic or true colour.

Special conditions apply on dynamic/pseudo colour displays, where a palette of 256 colours is usually available. On this type of display GINO allocates a default number of 64 colours from the system colour map, allowing multiple applications to operate without affecting the colours of each window. This default value can however be changed in one of three ways:

1) Setting an environment variable XNCOLS

2) Setting a variable XCOLS in the GINO configuration file

3) Calling the routine gSetColourInfo(ndc,ndt)

where the number of colours can be set to any value from 2 to 255. If the driver cannot obtain the requested number of colour cells from the Window Manager (being dependent on the number of cells allocated to other applications), it will use what it can or revert to monochrome output. Users should always use the routine gEnqColourInfo() to determine how many colours have been actually allocated in these circumstances.

If the value equals 256 then, rather than using the shared system colour map, a private colour map with 256 colours is allocated to the application. Whilst providing more colours to the GINO application, moving the mouse pointer in and out of the GINO window will cause an immediate colour change as the different colour palettes are loaded into memory.

Graphics Visibility

The current graphics window can be popped to the front of the display or pushed to the back of the display using the GINO routine gSetGraphicsVis().

e.g.

gSetGraphicsVis(GINVISIBLE)

Pushes the window to the back

gSetGraphicsVis(GVISIBLE)

Pops the window to the front

Character Fonts

The following hardware character fonts are accessible through the routine gSetCharFont(font) where font can be one of the following:

0 or 100	adobe-courier (default)
101	adobe-helvetica
102	adobe-times
103	adobe-ITC Avant Garde Gothic
104	adobe-ITC Lubalin Graph
105	adobe-New Century Schoolbook
106	adobe-ITC Souvenir

170	adobe-Symbol

The availability of the fonts is server dependent. The font enquiry routine gEnqHardFontList() can be used to enquire which fonts are available on the server being used. Where one of the hardware fonts listed above (except font 170) is not available, a software emulation is provided with similar character proportions.

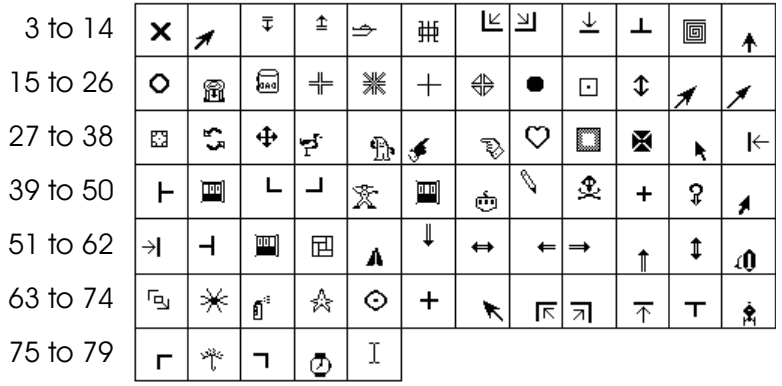
Bold and/or italic versions of fonts 100-106 can be selected using the routine gSetFontWeight(weight) where **weight**>0 and the routine gSetItalicAngle(angle) where 10≤**angle**≤20 degrees.

Mouse Pointer Types

When the window is first initialised the pointer icon is set to be an 'hour glass' indicating that no interaction can be performed. This is changed when either gGetCursorEvent() is called, or one of the permitted event types is enabled. The icon used at this point can be set using the routine gSetCursorType():

gSetCursorType(type,forcol,bakcol)

where **type** is the type number. Type -1 is an hour-glass, type 0 is a double cross, type 1 is a single small cross and type 2 is a large cross extending the whole height and width of the window. The following additional X cursor types are also available.



forcol and **bakcol** can also be used to set the cursor foreground and background colour numbers, remembering that cursors are drawn in XOR mode resulting in a different appearance to the GINO colour numbers on the actual display.

Graphics Cursor

In addition to the usual key values returned for a cursor hit, some special values of key are also returned as follows:

0	A call to gGetCursorEvent() has been made when the graphics window is iconized
1024	A resize event has taken place; the user should call gNewDrawing() and enquire the new size of the window (through gEnqDrawingLimits()) and redraw as required

While waiting for a key or button press, any expose events that take place will cause an automatic update of the exposed area from the backing store. When gGetCursorEvent() is called, the input focus of the device is set to be the graphics window. When the key has been pressed, the focus is returned to the window which had it before gGetCursorEvent() was called.

XWIN Hard Copy

A XWIN driver specific routine is available to create a PostScript image file from the contents of the current window:

gXwineps(mode,filename,ier)

Where **mode**=0 to create a monochrome or =1 to create a colour image, **filename** is the name of the PostScript file to be created and **ier** returns 0 if it has been successfullt created.

XWIN Driver Error Messages

The following errors specific to this device driver may occur:-

At Initialization:

GINO ERROR 310 - Cannot Open Display
GINO ERROR 311 - Cannot Open Window

These are caused by an incorrect connection to the display or network problems. In both cases, after outputting the error, the program will continue but no further output will take place.

At Initialization or gNewDrawing():

GINO ERROR 312 - Cannot obtain Window size

Character attribute setting:

GINO ERROR 313 - Cannot load required font

Iconized window at gCloseDevice():

GINO ERROR 314 - De-iconize Window

Using X-Windows from a remote host

When running an application using the X-Windows driver from a remote host, it is necessary to set permissions on the local workstation/terminal and set the correct display name on the remote host processor. This is achieved by carrying out the following steps:

Under UNIX:

```
On the local workstation / terminal
xhost +                                to allow access from all hosts OR
xhost <hostname>                       to allow access from a specific host
then login to the remote host using TELNET or rlogin
telnet <remote host name>              or rlogin
login:
password
and set the display name:
setenv DISPLAY <local display name>:0.0
```

PRINTERS AND PLOTTERS

This section includes hard copy devices such as pen-plotters, laser-printers and inkjet printers.

Output Filename and Unit Numbers (Fortran only)

By default, all printer/plotters use Fortran unit number 1 when creating a plot file. If the routine gSetDeviceFilename() or an OPEN statement has not been used, a file will be created with a default name such as FOR001.DAT under OpenVMS or HPGL.OUT under UNIX. A different unit number can be used by setting the relevant GINO configuration variable for that device as detailed below.

Printer and Plotter Configuration Settings

The following table lists all the available settings applicable to drivers in the printer/plotter section:

Device Driver	Config. Variable	Settings	Default	Description
CC907	CC907	N (1-99)	1	F90 output channel for CC907 driver
	CCHANDSH	String (TRUE/FALSE)	FALSE	ACK/NAK (TRUE), XON/XOFF (FALSE)
	CCCHECKSM	String (TRUE/FALSE)	FALSE	Enable checksum data
	CCSYNC	N (1-127)	2	Set plotter SYNC character (ASCII)
	CCEOB	N (1-127)	3	Set plotter EOB character (ASCII)
	CCCONTROL	N (906, 907)	Plotter-dependent	Set plotter controller mode
HPGL	HPGL	N (-8 to 99)	1	F90 output channel for HPGL driver
	HPHANDSH	String (P1, P2, P3)	P1	Set handshaking to XON/OFF (P1), ACK (P2) or hardwire (P3)
	HPGLFORMAT	String (YES/NO)	NO	Sets output file to be formatted
HPGL2	HPGL2	N (-8 to 99)	1	F90 output channel for HPGL2 driver
	HPGL2FORMAT	String (YES/NO)	NO	Sets output file to be formatted
HPLJ	HPLJ	N (-8 to 99)	1	F90 output channel for HPLJ driver
	HPLJNOFEED	String (YES/NO)	NO	Suppresses formfeed character at gCloseDevice()
HPPJ	HPPJ	N (-8 to 99)	1	F90 output channel for HPPJ driver
	HPPJNOFEED	String (YES/NO)	NO	Suppresses formfeed character at gCloseDevice()
LN03/LA100	LN03	N (1-99)	1	F90 output channel for LN03 driver
	LN03NOFEED	String (YES/NO)	NO	Suppresses formfeed character at gCloseDevice()
POSTSCRIPT	POSTSCRIPT	N (-8 to 99)	1	F90 output channel for POSTSCRIPT driver
	POSTFORMAT	String (YES/NO)	NO	Sets output file to be formatted
	POSTHEAD	N or String (ALL)	0	Output job creator/date on page N or ALL

Intermediate Vector File

Printers that are driven by raster output rather than vector output utilise an internal GINO vector to raster pre-processor to create the output for the device. Normally this conversion is done in a large memory area where such resources are available. However, where the drawing area is larger than the default an intermediate vector file is used and the output is banded. This file is written to and read from any output unit number that is available at device initialisation and then removed at gCloseDevice(). The output is not affected when banded in such a way except that pixel output is not available and a simple box is drawn to represent the pixel area. Drivers that use an Intermediate Vector File are as follows:

- HPLJ
- HPPJ
- LN03

8-bit data

Certain drivers produce code as 8-bit data. Whether output to the printer is direct or via a disk file, it is essential that the transmission is in raw 8 bit mode with no interpretation or conversion of characters. This may require special configuration of the printer, the communications line and/or the spool/plot command. (See Appendix A for the relevant implementation specific commands).

Drivers that use 8-bit data are as follows:

- HPLJ
- HPPJ

Calcomp 907 Series Plotters

Device Nominations

	Nomination Routine	Description of Device
A	gCc1023()	Calcomp 1023 Plotter
B	gCc5220()	Calcomp 5520 Plotter
C	gCc5725()	Calcomp 5725 Monochrome Electrostatic Plotter
D	gCc5835()	Calcomp 5835 Colour Electrostatic Plotter
E	gCc5912()	Calcomp 5912 Colour Electrostatic Plotter
F	gCc5735()	Calcomp 5735 Monochrome Electrostatic Plotter
G	gCcplot()	Calcomp PLOTMASTER

H	gCc5745()	Calcomp 5745 Plotter
I	gCc563(), gCc565()	Calcomp 563 and 565 Plotters
J	gCc1012()	Calcomp 1012 Plotter
K	gCc1036(), gCc1039()	Calcomp 1036 and 1039 Plotters
L	gCc1037()	Calcomp 1037 Plotter
M	gC1051n()	Calcomp 1051 Plotter with Narrow Paper
N	gCc1051()	Calcomp 1051 Plotter
O	gCc1055()	Calcomp 1055 Plotter
P	gCc945()	Calcomp 945 Plotter
Q	gCc960()	Calcomp 960 Plotter
R	gCc965()	Calcomp 965 Plotter
S	gCc1041()	Calcomp 1041 Plotter
T	gCc1042()	Calcomp 1042 Plotter
U	gCc1043()	Calcomp 1043 Plotter
V	gCalcmp(),gC1044n()	Calcomp 1044 Plotter with Narrow Paper
W	gCalwid(),gC1044w()	Calcomp 1044 Plotter with Wide Paper
X	gCc1077()	Calcomp 1077 Plotter

Device Characteristics

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**
Max. X Dimension (mm)	876	12m	12m	12m	406	12m	253	12m	12m	12m	12m	12m
Max. Y Dimension (mm)	610	858	597	895	256	895	195	1106	858	275	858	858
Default X Dimension	876	1500	1500	1500	406	1500	253	1500	1500	1500	1500	1500
Default Y Dimension	610	858	597	895	256	895	195	1106	858	275	858	858
Resolution (dots/cm)	800	200dpi	800	800	200dpi	800	200dpi	800	200	200	200	200
Colours / Pens	8	16	16	999	999	16	8	16	1	4	3	1
Colour Palette (fixed(F) or dynamic(D))	F	F	F	D	D	F	F	F	F	F	F	F
Broken Linestyles	6	No	6	6	6	No	6	6	No	No	6	No
Arcs	Yes	No	Yes	Yes	Yes	Yes	No	Yes	No	No	Yes	No
Polygonal Filling (solid only)	No	No	No	Yes	Yes	No	No	No	No	No	No	No
Image Handling (output only)	No	No	No	Yes	Yes	No	No	No	No	No	No	No
	Devices A to L											
Thick Lines	No											
Line Ends	No											
Symbols	No											
Fonts	1											
Character Sizes	Any Square Size and 4 Pseudo-hardware sizes in multiples of 1.5mm square											

	A	B	C	D	E	F	G	H	I	J	K	L
Character Angles	0 or 90°											
Italic Characters	No											

	M	N	O	P	Q	R	S	T	U	V	W	X
Max. X Dimension (mm)	12m	12m	12m	1520	1520	1520	1205	1270	1189	12m	12m	12m
Max. Y Dimension (mm)	275	858	858	841	841	864	594	858	858	275	858	858
Default X Dimension	1500	1500	1500	1500	1500	1500	1205	1270	1189	1500	1500	1500
Default Y Dimension	275	858	858	841	841	864	594	858	858	275	858	858
Resolution (dots/cm)	400	400	800	800	800	800	800	800	800	800	800	800
Colours / Pens	4	4	4	2	2	4	8	8	8	8	8	4
	Devices M to X											
Colour Palette	Fixed											
Broken Linestyles	6 (dashed, short-dotted, short-chained,long-dashed, long-dotted, long-chained)											
Thick Lines	No											
Line Ends	No											
Arcs	Yes											
Symbols	No											
Fonts	1											
Character Sizes	Any Square Size & 4 Pseudo-hardware sizes in multiples of 1.5mm square											
Character Angles	0 or 90°											
Italic Characters	No											
Polygonal Filling	No											
Image Handling	No											

Paper Advance

A call to gNewDrawing() or gCloseDevice() on pen-plotters will advance the paper and re-origin the pen, 12.7cm beyond the maximum X dimension of the previous plot. On the PLOTMASTER, 5700's, 5835 and 5912, a call to gNewDrawing() or gCloseDevice() will begin the rasterising and will eject the plot when finished. If gNewDrawing() has been called, a new sheet of paper will automatically be loaded.

Pen Selection

Models 5725, 5735, 5745

On monochrome electrostatic plotters, **col** can be in the range 1-16 referring to line thickness.

Models 5835, 5912

On the colour electrostatic plotter, model 5835 and the 5912 thermal plotter, **col** can be in the range 0-999 and in both line mode and fill mode, **col** = 0-10 give the standard GINO colours. **col** = 11-999 give the default hardware dither patterns. These can be redefined using gDefineRGB().

Communication Settings

The default configuration for communication to the Calcomp plotter required by this driver is:

- XON/XOFF handshaking
- No checksum data
- One SYNC character set to 2 (STX)
- End of message code set to 3 (ETX)

To change the above, one or more of the following configuration settings may be used:

CCHANDSH=TRUE

If software handshaking is required (plotter should be set to ACK/NAK handshaking mode).

CCCHECKSM=TRUE

If plotter requires checksum data as part of input record.

CCSYNC=N

To set plotter SYNC character to ASCII value N.

CCEOB=N

To set plotter EOB character to ASCII value N.

CCCONTROL=906 or 907

To set plotter controller if different to default.

Hewlett-Packard Series Plotters (HPGL)

Device Nominations

	Nomination Routine	Description of Device	Maximum Width	Maximum Height	Default Width	Default Height
A	gAb6610()	Advance Bryans 6610 A1 Plotter	884mm	580mm	Max	Max

B	gAb6680()	Advance Bryans 6680 A1 Plotter	884mm	580mm	Max	Max
C	gAb6910()	Advance Bryans 6910 A0 Plotter	1151mm	864mm	Max	Max
D	gAb6980()	Advance Bryans 6980 A0 Plotter	1151mm	864mm	Max	Max
E	gB1002()	Benson 1002 A4/A3 Plotter	410mm	287mm	270mm	170mm
F	gB1062()	Benson 1062 A3 Plotter	410mm	287mm	Max	Max
G	gDxy880()	Roland DXY880 A3 Plotter	380mm	270mm	Max	Max
H	gDx1100()	Roland DXY11/12/1300 A3 Plotters	431mm	297mm	Max	Max
I	gDs7()	Gould DS7 A4 Plotter	260mm	198mm	Max	Max
J	gDs10()	Gould DS10 A3 Plotter	360mm	280mm	Max	Max
K	gG6310()	Gould 6310 A4 Plotter	260mm	190mm	Max	Max
L	gG6320()	Gould 6320 A3 Plotter	360mm	280mm	Max	Max
M	gF4550()	Facit 4550 A4 Plotter	274mm	191mm	Max	Max
N	gF4551()	Facit 4551 A3 Plotter	401mm	274mm	Max	Max
O	gLvp16()	DEC LVP16 with A4 Paper	274mm	191mm	Max	Max
P	gLvp16w()	DEC LVP16 with A3 Paper	401mm	274mm	Max	Max
Q	gHp7220()	Hewlett Packard 7220 A3 Plotter	400mm	285mm	Max	Max
R	gHp7225()	Hewlett Packard 7225 A4 Plotter	285mm	203mm	Max	Max
S	gHp7470()	Hewlett Packard 7470 A4 Plotter	274mm	191mm	Max	Max
T	gHp7475()	Hewlett Packard 7475 with A4 Paper	274mm	191mm	Max	Max
U	gH7475w()	Hewlett Packard 7475 with A3 Paper	401mm	274mm	Max	Max
V	gHp7440()	Hewlett Packard 7440 A4 Plotter	271mm	190mm	Max	Max
X	gHp7550()	Hewlett Packard 7550 with A4 Paper	271mm	190mm	Max	Max
Y	gH7550w()	Hewlett Packard 7550 with A3 Paper	399mm	271mm	Max	Max
Z	gA47580()	Hewlett Packard 7580 with A4 Paper	278mm	171mm	262mm	151mm
AA	gA37580()	Hewlett Packard 7580 with A3 Paper	401mm	286mm	360mm	262mm
BB	gA27580()	Hewlett Packard 7580 with A2 Paper	599mm	385mm	559mm	361mm
CC	gA17580()	Hewlett Packard 7580 with A1 Paper	822mm	583mm	782mm	559mm
DD	gHpa4r()	HP7575/76/85/86/Benson 1834 - A4	254mm	195mm	238mm	175mm
EE	gHpa3r()	HP7575/76/85/86/Benson 1834 - A3	425mm	262mm	385mm	238mm
FF	gHpa2r()	HP7575/76/85/86/Benson 1834 - A2	575mm	409mm	535mm	385mm
GG	gHpa1r()	HP7575/76/85/86/Benson 1834 - A1	845mm	559mm	805mm	535mm
HH	gHpa4()	HP7575/76/85/86/Benson 1834 - A4	278mm	171mm	262mm	151mm
II	gHpa3()	HP7575/76/85/86/Benson 1834 - A3	401mm	286mm	360mm	262mm
JJ	gHpa2()	HP7575/76/85/86/Benson 1834 - A2	599mm	385mm	559mm	361mm
KK	gHpa1()	HP7575/76/85/86/Benson 1834 - A1	822mm	583mm	782mm	559mm
LL	gHpa0()	HP7575/76/85/86/Benson 1834 - A0	1170mm	830mm	1130mm	806mm
MM	gHp96a4()	Hewlett-Packard 7596 with A4 Paper	260mm	198mm	260mm	151mm
NN	gHp96a2()	Hewlett-Packard 7596 with A2 Paper	599mm	385mm	Max	Max
OO	gRn7586()	HP7585/86/96/Benson 1834 - 11" Roll	420mm	286mm	286mm	210mm
PP	gRm7586()	HP7585/86/96/Benson 1834 - 24" Roll	841mm	574mm	574mm	420mm

QQ	gRw7586()	HP7585/86/96/Benson 1834 - 36" Roll	1189mm	841mm	841mm	578mm
RR	gV8536()	Versatec 8536 electrostatic A0 Plotter	1219mm	893mm	Max	Max
SS	gLa7586()	HP 7586 Long Axis Plotter	30m	841mm	1189mm	817mm
TT	gX4030()	Xerox 4030 Laser Printer	297mm	210mm	Max	Max
UU	gSe7220()	Generic lower-left origin plotter	25m	25m	400mm	285mm
VV	gSehpa0()	Generic centre-origin plotter	25m	25m	1130mm	806mm

Device Characteristics

	A	**C**	**E**	**F**	**I**	**J**	**K**	**L**	**M,N,O, P**
Colours / Pens	1	1	4	9	7	10	7	10	6
	Devices A to P								
Resolution (dot/cm)	400								
Broken Linestyles	6 (dotted, short-dashed, long-dashed, dash-dotted, dash-long-dotted, dash-dot-dotted)								
Thick Lines	No								
Line Ends	No								
Arcs	Yes								
Symbols	No								
Fonts	1								
Character Sizes	Any and 4 Pseudo-hardware in sizes of 1.5mm square								
Character Angles	Any								
Italic Characters	Any								
Polygonal Filling	Solid & Rectangular only								
Image Handling	No								

	Q	**R**	**S**	**T & U**	**V**	**RR**	**SS**	**TT**	**UU**	**VV**	**All Others**
Colours / Pens	8	1	2	6	8	1	8	1	256	256	8
Arcs	Yes	Yes				Yes	Yes	Yes	Yes	Yes	Yes
Polygonal Filling (Rectangular)	No	No	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes
	Devices Q , R, S, T, U, V, RR, SS, TT, UU, VV, and all others										
Broken Linestyles	6 (dotted, short-dashed, long-dashed, dash-dotted, dash-long-dotted, dash-dot-dotted)										
Thick Lines	No										
Line Ends	No										
Symbols	No										
Fonts	1										
Character Sizes	Any and 4 Pseudo-hardware in sizes of 1.5mm square										
Character Angles	Any										
Italic Characters	Any										

Image Handling	No

gHpa4r()/gHpa3()/gHpa2r()/gHpa1()/gHpa0() draw on paper loaded with the long axis vertical.

gHpa4()/gHpa3r()/gHpa2()/gHpa1r() draw on paper loaded with the long axis horizontal.

The nomination routines gSe7220() and gSehpa0() are provided for generic HPGL devices with lower-left origin and centre-origin coordinate systems respectively. Apart from the maximum drawing area and number of pens available, they behave as per the devices gHp7220() and gHpa0().

Drawing Area

On plotter model 7586 and B1834, if using roll-media, the default drawing area is A4 landscape (gRn7586()), A2 landscape (gRm7586()) or A1 landscape (gRw7586()) and the drawing will be automatically rotated to fit across the paper.

If using single-sheet media on models 7570/75/76/80/85/86/96 the default drawing area is as stated above.

The maximum drawing area on models 7570/75/76/80/85/95/96 includes drawing beyond the pinch wheels and allows for the use of oversize sheets. If using standard size sheets, gSetDrawingLimits() should be called with values no more than 18mm less than the maximum X dimensions listed overleaf so as not to shift the origin off the bottom of the paper. In all cases the area defined by gSetDrawingLimits() is placed centrally within the current hard clip limits of the plotter. **type** can be any value and is ignored.

Paper Advance

A call to gNewDrawing() or gCloseDevice() will automatically wind the paper on or select a new sheet of paper on all plotters with this facility. On the Gould plotters, the paper is wound on only by however much has been used. On all other paper advance plotters, the wind-on is by a half-page or full-page (full page only on gB1062()).

On the HP7586 when using Roll-Media, a half-page or full-page advance is used depending on the size of the next required drawing area. For this reason, the advance is only done at the beginning of a plot and at a gNewDrawing(), but not at gCloseDevice().

Handshaking & File Format

The following configuration settings can be made to alter the default handshaking and file format:

HPHANDSH=P1 P2 or P3

Force handshaking mode to be Xon-Xoff, ENQ/ACK or Hardwire respectively

HPGLFORMAT=YES

Sets the default output file format to be formatted and therefore suitable for some spooler systems. This setting is overridden by application programs that use the routine gSetDeviceFilename().

Hewlett-Packard Series Plotters (HPGL-2)

Device Nominations

	Nomination Routine	Description of Device
A	gHplj3()	Hewlett Packard LaserJet III
B	gHpmx()	Hewlett Packard DraftMaster MX, RX and SX
C	gH76250()	Hewlett Packard 7600 Electrostatic Plotter Model 250
D	gH76255()	Hewlett Packard 7600 Electrostatic Plotter Model 255
E	gH76355()	Hewlett Packard 7600 Electrostatic Plotter Model 355
F	gHpgl2(xmax, ymax, xdef, ydef, npens)	Generic Routine (Portrait Printers/Plotters)
G	gHpgl2r(xmax, ymax, xdef, ydef, npens)	Generic Routine (Landscape Printers/Plotters)

Device Characteristics

	A	B	C	D	E	F	G
Maximum Width (mm)	388.6	1163.2	836.6	1117.6	1117.6	††	††
Maximum Height (mm)	266.0	882.4	558.8	863.6	863.6	††	††
Default Width (mm)	271.0	238.0	297.0	297.0	297.0	††	††
Default Height (mm)	197.0	177.0m	210.0	210.0	210.0	††	††
Resolution	300dpi	400dots/cm	406dpi	406dpi	406dpi	†	†
Colours / Pens	1	8	255 Grey	255 Grey	255	††	††
Colour Palette	Static	Static	Dynamic	Dynamic	Dynamic	†	†
Thick Lines	Yes	No	Yes	Yes	Yes	†	†
Line Ends	3	0	3	3	3	†	†
Fonts (0-courier, 101-Universal, 102-CG Times)	Yes	Yes	Yes	Yes	Yes	†	†

	Devices A to G
Broken Linestyles	16
Arcs	Yes
Symbols	No
Character Sizes	Any & 16 Pseudo-hardware in multiples of 1.5mm square
Character Angles	Any
Italic Characters	No
Polygonal Filling	Multi-polygon, solid only
Image Handling	No

† Device-dependent
†† User defined

Generic Routines

In order to use this driver for the large number of devices with HPGL/2 emulations, the following two generic routines are provided:

gHpgl2(xmax,ymax,xdef,ydef,npens)	for Portrait printer/plotters
gHpgl2r(xmax,ymax,xdef,ydef,npens)	for Landscape printer/plotters

Where

xmax
Maximum width of paper in mm.

ymax
Maximum height / length of paper in mm.

xdef
Default width of plot in mm.

ydef
Default height / length of plot in mm.

npens
Number of pens / colours available.

Drawing Area and Orientation

On the DraftMaster range of pen plotters the origin of the plot according to the hardware is set at half the drawable width of the media defined by the hardware taking physical measurements and half the length as defined by GINO plus the margin. This origin will be the centre point of a GINO drawing. GINO plots are drawn along the direction of paper travel. As the orientation of GINO plots is landscape, the paper must be loaded into a DraftMaster pen plotter with its longest side along the direction of paper travel. If the command gSetDrawingLimits() is called with parameters corresponding to a portrait plot, the plot will be automatically rotated in order to give greater paper usage efficiency.

Paper Advance

A call to gNewDrawing() or gCloseDevice() will eject the paper on or select a new sheet of paper. On plotters with automatic layout facilities a new suitable area for the next plot will be found if it follows within the specified time limits for the layout facility.

Colours

Colour 0 will select background erase except on the DraftMaster pen plotters.

Hardware Fonts

The default font is available in a variety of font weights which can be selected using the routine gSetFontWeight(weight) where weight can have values between -7 and 7 for weights varying from Ultra Thin to Ultra Black (with 0 being the default). Both the proportional fonts can be displayed either filled or in outline form and at 2 different font weights by using the routines gSetFontFillStyle() and gSetFontWeight():

gSetFontFillStyle(style)

gSetFontWeight(weight)

where **style.type** can be set to GOUTLINE or GFILLED for outline and filled fonts respectively, and **weight** can be set to 0 or 1 for normal and bold weights respectively. The remaining arguments to gSetFontFillStyle() have no effect on the font style.

File Format

The following configuration setting can be made to alter the default file format:

HPGL2FORMAT=YES

Sets the default output file format to be formatted and therefore suitable for some spooler systems. This setting is overridden by application programs that use the routine gSetDeviceFilename().

Hewlett-Packard Laserjet Series Printers (HPLJ)

Device Nominations

	Nomination Routine	Description of Device
A	gHpljr()	Hewlett Packard LaserJet Printers using run-length encoding (Series III onwards)
B	gHpljp()	Hewlett Packard LaserJet Printers using compressed encoding (Series IIP)
C	gHplj()	Hewlett Packard LaserJet Printers using uncompressed encoding (Series I & II)
D	gHplj6()	600cpi Hewlett Packard LaserJet Printers using run-length encoding (Series IV onwards)

Device Characteristics

	A, B and C	D
Max. X Dimension (mm)	388.6	253.9
Max. Y Dimension (mm)	266.0	196.7
Default X Dimension (mm)	280.4	280.4
Default Y Dimension (mm)	196.7	196.7
Resolution (dpi)	300	600
	Devices A to D	
Colours / Pens	0 (erase) & 16 dithered greys	
Broken Linestyles	No	
Thick Lines	No	
Line Ends	No	
Arcs	No	
Symbols	No	
Fonts	No	
Character Sizes	4 Pseudo-hardware in multiples of 1.5mm square	
Character Angles	0 or 90°	
Italic Characters	No	
Polygonal Filling	No	
Image Handling	Yes but not when using Intermediate Vector File	

Paper Size and Tray Selection

Where multiple paper trays or paper sizes are available on the nominated printer to which the HPLJ file is sent, the **type** argument to the routine gSetDrawingLimits() can be used to make the desired selection.

Values 1-99 select a paper/envelope size according to the PCL list which includes:

- 1 = executive
- 2 = letter
- 3 = legal
- 26 = A4

Values 100,200,300,400,500 and 600 select the paper source according to the PCL list which includes:

- 100 = First tray
- 200 = Manual feed - paper
- 300 = Manual feed - envelope

Note that this list may vary from printer to printer and version of PCL.

Formfeed at End of Output

The following configuration setting can be made to suppress the formfeed at gCloseDevice():

HPLJNOFEED=YES

Suppresses the formfeed character sent at gCloseDevice() for use when form feed is already sent by the print spooler.

Hewlett-Packard Paintjet and Deskjet Printers (HPPJ)

Device Nominations

	Nomination Routine	Description of Device
A	gDj500m()	Hewlett Packard DeskJet using the black ink cartridge
B	gDj500c()	Hewlett Packard DeskJet using the colour ink cartridge
C	gDj510()	Hewlett Packard DeskJet 510

D	gDj550c()	Hewlett Packard DeskJet 550 using both cartridges
E	gDj560c()	Hewlett Packard DeskJet 560
F	gDj1200()	Hewlett Packard DeskJet 1200
G	gDj690()	Hewlett Packard DeskJet 690 monochrome
H	gDj690c()	Hewlett Packard DeskJet 690 colour
I	gDj890c()	Hewlett Packard Deskjet 890 colour

Device Characteristics

	A	B	C	D	E	F	G	H	I
Max. X dimension (mm)	288.8	288.8	215.9	228.6	228.6	228.6	228.6	228.6	228.6
Max Y dimension (mm)	279.4	279.4	252.0	279.4	279.4	279.4	9144.0	9144.0	9144.0
Default X dimension (mm)	203.2	203.2	203.2	203.2	203.2	203.2	203.2	203.2	203.2
Default Y dimension (mm)	266.7	266.7	252.0	266.7	266.7	266.7	247.65	247.65	247.65
Resolution (dpi)	300	300	300	300	300	300	600/300	600/300	600
Colours / Pens	16gr	255	16gr	255	255	255	16gr	255	255
	Devices A to I								
Broken Linestyles	16								
Thick Lines	No								
Line Ends	No								
Arcs	No								
Symbols	No								
Fonts	No								
Character Sizes	4 Pseudo-hardware in multiples of 1.5mm square								
Character Angles	0 or 90°								
Italic Characters	No								
Polygonal Filling	No								
Image Handling	Yes, but not when using Intermediate Vector File								

Colour

On monochrome printers, 16 dithered greys can be obtained. On the colour printers, colours can be defined using gDefineRGB().

When printing using the gDj550c() nomination, the black ink may bleed into the colour ink due to differing chemical compositions of the inks. Two solutions are to use special glossy HP paper or switch to using the gDj500c() nomination which only uses 1 cartridge (the black will then appear as a dark green colour!)

Formfeed at End of Output

The following configuration setting can be made to suppress the formfeed at gCloseDevice():

HPPJNOFEED=YES

Suppresses the formfeed character sent at gCloseDevice() for use when form feed is already sent by the print spooler.

DEC LA100 andLN03 Series Printers

Device Nominations

	Nomination Routine	Description of Device
A	gLa50()	DEC LA50 Dot Matrix Printer
B	gLa75()	DEC LA75 Dot Matrix Printer
C	gLa100()	DEC LA100 Dot Matrix Printer
D	gLa210()	DEC LA210 Dot Matrix Printer
E	gLn03()	DEC LN03/LN03+ Laser Printer
F	gLg31()	DEC LG31 Dot Matrix Printer

Device Characteristics

	A	B	C	D	E	F
Max. X Dimension (mm)	215.0	215.0	332.7	332.7	286.0	332.7
Max. Y Dimension (mm)	1016.0	1016.0	1016.0	1016.0	203.0	1016.0
Default X Dimension (mm)	203.2	203.2	332.7	332.7	285.9	332.7
Default Y Dimension (mm)	152.4	152.4	254.0	254.0	197.9	254.0
Resolution (dpi)	144x72	144x72	132x72	132x72	300x300	132x72
	Devices A to F					
Colours / Pens	0 (Background erase) & 16 dithered greys					
Broken Linestyles	No					
Thick Lines	No					
Line Ends	No					
Arcs	No					
Symbols	No					
Fonts	No					
Character Sizes	4 Pseudo-hardware of multiples of 1.5mm square					
Character Angles	0 or 90°					
Italic Characters	No					

Polygonal Filling	No
Image Handling	Yes but not when using Intermediate Vector File

Formfeed at End of Output

The following configuration setting can be made to suppress the formfeed at gCloseDevice():

LN03NOFEED=YES

Suppresses the formfeed character sent at gCloseDevice() for use when form feed is already sent by the print spooler.

Postscript Series Printers

Device Nominations

	Nomination Routine	Description of Device
A	gLn03r()	DEC LN03R Postscript Laser Printer
B	gLaserw()	Apple/Sun Laser Writer
C	gLps40()	DEC Print Server 40
D	gD12150()	DEC Laser 2150
E	gEps(cflag, xoff, yoff, xsize, ysize, xdef, ydef)	Generic Routine (for printers)
F	gEpsexp(xsize, ysize, xmarg, ymarg, n, prop)	Generic Routine (for import filters)

Device Characteristics

	A	B	C	D	E	F
Max X Dimension (mm)	284.5	284.5	431.8	290.5	User Defined	User Defined
Max Y Dimension (mm)	197.3	201.0	297.0	198.0	User Defined	User Defined
Default X Dimension (mm)	284.5	284.5	293.0	290.5	User Defined	User Defined
Default Y Dimension (mm)	197.3	197.0	206.0	198.0	User Defined	User Defined
Resolution (dpi)	300	300	300	300	Device dependent	Device dependent
Colours / Greyscales	255	255	255	255	User Defined	User Defined
Colour Palette	Static (default) or Direct	Static (default) or Direct	Static (default) or Direct	Static (default) or Direct	User Defined	User Defined
	Devices A to F					
Broken Linestyles	16					
Thick Lines	Yes (Default=0.0)					
Line Ends	3					
Arcs	Yes					
Symbols	Yes					

Fonts	11
Character Sizes	Any + 16 Pseudo-hardware in multiples of 1.5mm square
Character Angles	Any
Italic Characters	Any
Polygonal Filling	Solid only and maximum of 1500 Points
Image Handling	Yes (Output only)
Clipping	No
Window/Device Titling	Yes

The PostScript code produced by this driver conforms to the EPSF (Encapsulated PostScript File) format specified by Adobe Systems Inc., enabling GINO output to be exported to other applications such as desktop publishing systems (DTP). This driver provides one generic nomination routine gEps() for printers, one generic nomination routine gEpsexp() for import filters plus a number of specific nomination routines for older printers.

The default orientation when using any of the printer nomination routines is for the picture to be drawn in landscape mode, but the default for the gEpsexp() routine is portrait. In both cases the orientation can be changed by switching the X and Y paper size (either in the nomination routine, by calling gSetDrawingLimits() or by setting the necessary flag in gEpsexp()).

In the case of the generic PostScript export nomination routine, the application can control various properties of the PostScript file through the arguments, so that they are suitable for the import filter being used. Users are referred to the chapter on 'Importing and Exporting' to enquire the requirements and options of the most popular import filters that accept PostScript files.

Device Nomination

The following generic printer routine is available with this driver:

gEps(cflag,xoff,yoff,xsize,ysize,xdef,ydef)

The parameters have the following values:

cflag

Int — Colour flag (+'ve = Colour, -'ve = Greyscales, 0 = Monochrome). See Colour Setting section below.

xoff,yoff

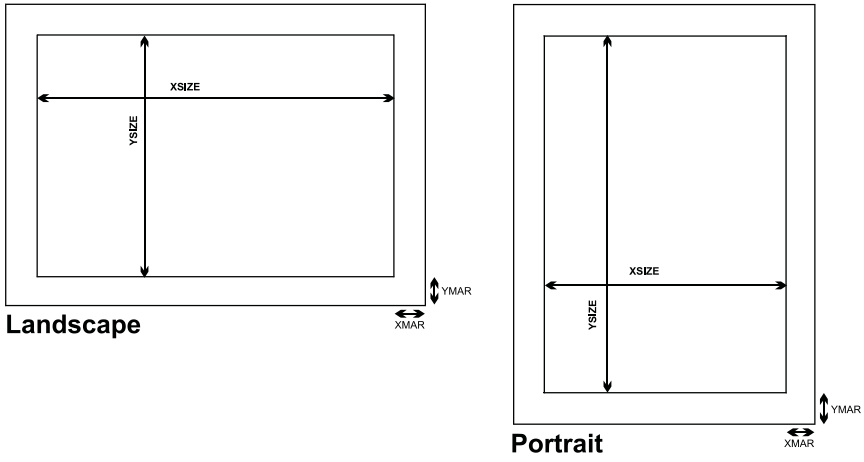
Float — Shortest distance in millimetres of drawable area from the PostScript origin.

xsize,ysize

Float — Maximum size of drawing area in millimetres.

xdef,ydef

Float — Default size of drawing area in millimetres.



The above diagram shows the definition of **xsize** and **ysize** which is the same for **xdef** and **ydef**. Note that the **xoff** and **yoff** distances are reversed depending on the orientation of the GINO picture. It must also be realised that the PostScript origin may not be the same as the paper origin on some devices.

A second generic routine is available aimed at exporting PostScript files from GINO to be read by WP, DTP import filters and PostScript viewers:

gEpsexp(xsize, ysize, xmarg, ymarg, n, prop)

The parameters have the following meanings:

xsize,ysize

Float — Print area in millimetres.

xmarg,ymarg

Float — Horizontal and vertical margins (added to Bounding Box limits).

n

Int	Number of elements in prop array (rounded down to the nearest even number)

prop

Int []	EPS properties array containing PAIRS of property flags and their settings according to the table below.

Property	Property Description	Settings
1	Initial Colour Mode (NDT)	= -2, Indexed Greyscale = 0, Monochrome = 2, Index Colour (Default) = 4, True Colour
2	Rotation Flag	= -1, Auto rotation (0° or 270°) = 0, No rotation (Default) = 1, 90° rotation = 2, 180° rotation = 3, 270° rotation
3	Bounding Box Flag	= 0, Print area at start of file = 1, Drawing area at end of file (Default) = 2, Drawing area at start of file
4	TIFF header	= 0, No TIFF header (Default)

The PostScript file generated through the gEpsexp() nomination routine will represent an area defined by **xsize,ysize** and contain optional margins as specified by the argument **xmarg, ymarg**. Additional properties, if different from the defaults, may be specified by passing a property and its setting, in pairs, through the **prop** array. Thus if an import filter required a monochrome PostScript file with the actual drawing limits at the start of the file, the values 1,0,3,2 would be passed in **prop** with **n** set to 4.

The auto-rotation setting is the same as the default operation of the gEps() nomination routine whereby a landscape print area (**xsize** > **ysize**) will cause a rotation of 270° of the plot.

Bounding Box

The bounding box flag determines the position and source of the values placed in the %%BoundingBox command within the PostScript file. The default option (1) places the %%BoundingBox (atend) command at the start of the file and places the actual drawing limits in a command at the end of the file. Many PostScript interpreters, however, require the %%Bounding Box command at the start of the file and for this, two options are available. Option 0 places the print/paper area as defined by the arguments in gEpsexp (or that subsequently re-defined in a call to gSetDrawingLimits) at the start of the file. Option 2 places the actual limits of the drawing at the start of the file by postprocessing it once these limits are known. The difference between the print/paper area and the actual drawing limits is that the latter only includes the rectangular area where drawing actually occurs ignoring any blank space round the edges and therefore can be more accurate.

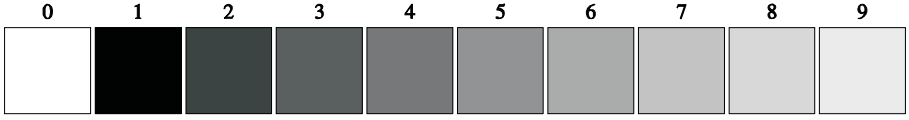
Paper Size and Paper Tray

The paper size can only be enlarged on model LPS40 or when using gEps() where the maximum drawing area is larger than the default. Any size greater than the default drawing size assumes that A3 paper is loaded in the printer. All other models have the default paper size the same as the maximum paper size. Where alternative paper trays are available, these may be selected using the **type** argument to gSetDrawingLimits(). Values of 1,2 or 3 will select paper trays 0,1 and 2 using the PostScript 'setpapertray' command. If **type** = 0 no paper tray selection is made.

Colours/Greyscales

PostScript devices can be defined as colour, monochrome or greyscale through the nomination routines gEps() and GEpsexp(), all other nominations are defined as greyscale. On colour and greyscale devices the number of colours/greyscales available is 255 and on monochrome devices it is 0 (erase) and 1. If the device has been setup as a monochrome device, gDefineRGB() will have no effect.

On greyscale devices colours 0-9 default to the following shades of grey with 0=white and 1=black.



Colour and greyscales are used for all output primitives except for single dot width lines on greyscale devices as this results in a dashed line effect.

Character Fonts

Hardware fonts can be selected using gSetCharFont(font) where **font** can be:

100	Courier (default)	**106**	Souvenir
101	Helvetica	**107**	Palatino
102	Times	**108**	ZapfChancery
103	AvantGarde-Book		
104	LubalinGraph	**170**	Symbol
105	NewCenturySchlbk	**171**	ZapfDingbats

Any value of **font** outside of this range or if the font selected is not available, will select the Courier font. The Courier font is a fixed-pitch font, the remainder are variable-pitch.

Bold versions of fonts 100-107 can be selected by using the routine gSetFontWeight(weight) where the standard font is selected if **weight** is less than or equal to zero and the bold version is selected if **weight** is greater than zero.

Outline versions of the fonts can be selected using the routine gSetFontFillStyle(style) where **style.type** is set to GOUTLINE. All other values of **style.type** select the solid filled font. The remaining arguments are only used for GINO software fonts.

File Format & Job Title

The following configuration settings can be made to alter the default file format and to add job titles to pages:

POSTFORMAT=YES	Sets the default output file format to be formatted and therefore suitable for inclusion into DTP systems. This setting is overridden by application programs that use the routine gSetDeviceFilename().
POSTHEAD=ALL	Automatically outputs job creator and date to top left of every page
POSTHEAD=n	Automatically outputs job creator and date to page n

METAFILES

This section includes device-independent metafiles used for storing picture data in an independent format or for transferring picture data from GINO to another package (see page). See also the ‘Postscript Series Printers’ section for details of Encapsulated Postscript (EPS) and the HPGL driver in the ‘Printers/Plotters’ section for two other formats used for exporting graphics.

Output Filename and Unit Numbers for Metafiles(Fortran only)

By default, all metafile drivers use Fortran unit number 7 when creating a file. If this unit number has not been opened, a file will be created with a default name depending on the system, such as FOR007.DAT under OpenVMS or CGM.OUT on the PC or under UNIX. A different unit number can be used, either by setting the relevant GINO configuration variable for that device or by setting the default metafile unit number via the configuration variable NSAVDF.

File Format

The second parameter in the routine gSetDeviceFilename() is ignored in all metafile drivers except for IMAGE.

Metafile Configuration Settings

The following table lists all the available settings applicable to drivers in this section:

Device Driver	Config. Variable	Settings	Default	Description
All	NSAVDF	N (1-99)	7	F90 output channel for all metafile drivers
CGM	CGM	N (1-99)	7	F90 output channel for CGM driver
DXF	DXF	N (1-99)	7	F90 output channel for DXF driver
IMAGE	IMAGE	N (1-99)	7	F90 output channel for IMAGE driver
JPEG	JPEG	N(1-99)	7	F90 output channel for JPEG driver
PNG	PNG	N(1-99)	7	F90 output channel for PNG driver
SAVDRA	SAVDRA	N (1-99)	7	F90 output channel for SAVDRA driver
WMF	WMF	N (1-99)	7	F90 output channel for WMF driver

Computer Graphics Metafile (CGM)

Device Nominations

	Nomination Routine	Description of Device
A	gCgmchi()	For character encoded formats with integer coordinates
B	gCgmchr()	For character encoded formats with real coordinates
C	gCgmbi()	For binary encoded formats with integer coordinates
D	gCgmbr()	For binary encoded formats with real coordinates

Device Characteristics

	A & C	B & D
Maximum Width (mm)	327.67	No Limit
Maximum Height (mm)	327.67	No Limit
	Devices A to D	
Default Width (mm)	200.0	
Default Height (mm)	200.0	
Colours / Pens	255	
Colour Palette	Dynamic	
Broken Linestyles	5	
Drawing Modes	No	

Thick Lines	Yes
Line Ends	No
Arcs	Yes
Symbols	No
Fonts	1
Character Sizes	Any & 16 Pseudo-hardware in multiples of 1.5mm square
Character Angles	Any
Italic Characters	Any angle
Polygonal Filling	Yes
Segments	No
Image Handling	Yes
Clipping	No

File Format

Both character and binary encodings of the CGM driver produce a file containing a single stream of eight-bit bytes with no record structure.

All nomination routines will always generate files with metric scaling mode as GINO operates in a real coordinate system rather than an abstract one.

Multi-Pictures

More than one picture can be stored in any CGM metafile and the routine gNewDrawing() is used to separate each picture.

CGM Elements

The full list of legal CGM element identifiers is given below together with notes on their use by the GINO generator and interpreter.

Element ID	Element Name	Generated	Interpreted
	Delimiter Elements		
131	Null	N	Y
132	BEGIN METAFILE	Y	Y
133	END METAFILE	Y	Y
134	BEGIN PICTURE	Y	Y
135	BEGIN PICTURE BODY	Y	Y
136	END PICTURE	Y	Y

Metafile Descriptor Elements

232	METAFILE VERSION	Y	I
233	METAFILE DESCRIPTION	Y	I
234	VDC TYPE	Y	Y
235	INTEGER PRECISION	Y	Y
236	REAL PRECISION	Y	Y
237	INDEX PRECISION	Y	Y
238	COLOUR PRECISION	Y	Y
239	COLOUR INDEX PRECISION	Y	Y
240	MAXIMUM COLOUR INDEX	Y	Y
241	COLOUR VALUE EXTENT	Y	Y
242	METAFILE ELEMENT LIST	Y	Y
243	BEGIN METAFILE DEFAULTS REPLACEMENT	Y	Y
244	END METAFILE DEFAULTS REPLACEMENT	Y	Y
245	FONT LIST	N	N
246	CHARACTER SET LIST	Y	Y
247	CHARACTER CODING ANNOUNCER	Y	Y

Picture Descriptor Elements

332	SCALING MODE	Y	Y
333	COLOUR SELECTION MODE	Y	Y
334	LINE WIDTH SPECIFICATION MODE	Y	Y
335	MARKER SPECIFICATION MODE	Y	Y
336	EDGE WIDTH SPECIFICATION MODE	Y	Y
337	VDC EXTENT	Y	Y
338	BACKGROUND COLOUR	Y	Y

Control Elements

432	VDC INTEGER PRECISION	Y	Y
433	VDC REAL PRECISION	Y	Y

434	AUXILIARY COLOUR	N	N
435	TRANSPARENCY	N	N
436	CLIP RECTANGLE	N	Y
437	CLIP INDICATOR	N	Y

Graphical Primitive Elements

032	POLYLINE	Y	Y
033	DISJOINT POLYLINE	N	Y
034	POLYMARKER	N	Y
035	TEXT	Y	Y
036	RESTRICTED TEXT	N	Y
037	APPEND TEXT	N	N
038	POLYGON	N	Y
039	POLYGON SET	Y	Y
040	CELL ARRAY	Y(4)	Y
041	GENERALIZED DRAWING PRIMITIVE	N	Y(1)
042	RECTANGLE	N	Y
532	CIRCLE	Y	Y
533	CIRCULAR ARC 3 POINT	N	Y
534	CIRCULAR ARC 3 POINT CLOSE	N	Y
535	CIRCULAR ARC CENTRE	Y	Y
536	CIRCULAR ARC CENTRE CLOSE	N	Y
537	ELLIPSE	N	N
538	ELLIPTICAL ARC	N	N
539	ELLIPTICAL ARC CLOSE	N	N

Attribute Elements

632	LINE BUNDLE INDEX	N	N
633	LINE TYPE	Y	Y
634	LINE WIDTH	Y	Y
635	LINE COLOUR	Y	Y
636	MARKER BUNDLE INDEX	N	N
637	MARKER TYPE	N	Y

638	MARKER WIDTH	N	Y
639	MARKER COLOUR	N	Y
648	TEXT BUNDLE INDEX	N	N
649	TEXT FONT INDEX	N	N
650	TEXT PRECISION	Y	N(2)
651	CHARACTER EXPANSION FACTOR	Y	Y
652	CHARACTER SPACING	Y	Y
653	TEXT COLOUR	Y	Y
654	CHARACTER HEIGHT	Y	Y
655	CHARACTER ORIENTATION	Y	Y
656	TEXT PATH	N	Y
657	TEXT ALIGNMENT	N	Y
658	CHARACTER SET INDEX	Y	Y
659	ALTERNATE CHARACTER SET INDEX	Y	Y
732	FILL BUNDLE INDEX	N	N
733	INTERIOR STYLE	Y(3)	Y(3)
734	FILL COLOUR	Y	Y
735	HATCH INDEX	Y	Y
736	PATTERN INDEX	N	N
737	EDGE BUNDLE INDEX	N	N
738	EDGE TYPE	Y	Y
739	EDGE WIDTH	Y	Y
740	EDGE COLOUR	Y	Y
741	EDGE VISIBILITY	N	N
742	FILL REFERENCE POINT	N	N
743	PATTERN TABLE	N	N
744	PATTERN SIZE	N	N
748	COLOUR TABLE	Y	Y
749	ASPECT SOURCE FLAGS	N	N

Escape Elements

832	ESCAPE	N	N
833	MESSAGE	N	N
834	APPLICATION	N	N

848	DOMAIN RING	N	N

Y= This element is generated/interpreted by the GINOCGM software

N= This element is not generated/interpreted by the GINOCGM software

I = This element is ignored by the GINOCGM interpreter

Notes:

1) Generalized Drawing primitives are interpreted as a polyline

2) All text is interpreted in software transformed mode

3) Pattern fill style is not implemented

4) Cell arrays are output as one element if the number of 'componenst' (nx*ny) is less than 2048. If the number is greater than this, the image is encoded as a series of cell arrays representing each row of the cell array.

Drawing Exchange Format (DXF) Metafile

Device Nominations

	Nomination Routine	Description of Device
A	gDxf()	Drawing exchange format file

Device Characteristics

	A
Maximum Width	100km
Maximum Height	100km
Default X Dimension	420mm
Default Y Dimension	297mm
Colours / Pens	255
Colour Palette	Dynamic
Broken Linestyles	No
Drawing Modes	No
Thick Lines	Any Thickness and only on vectors
Line Ends	0 and 1
Arcs	Yes
Symbols	No

Fonts	1
Character Sizes	Any size
Character Angles	Any angle
Italic Characters	No
Polygonal Filling	Up to 4 Vertices
Segments	No
Image Handling	No
Clipping	No
Window/Device Titling	Yes

DXF Content

The DXF metafile contains only selected system variables and the entities section. This will create consistency between the picture if it is completely GINO, or if it is only partly GINO. Each entity has no associated hierarchy so the original order cannot be guaranteed. Each entity in the DXF file is assigned to layer 0 and includes its own independent colour attribute.

Multi-pictures

Multi-pictures are not available. The routine gNewDrawing() has no effect.

Colour

There is no facility within DXF for using the background colour as a drawing colour. This results in colours 0, 1 and 10 being identical (being the inverse of the background colour). There is a limited colour redefinition facility which involves redefining pointers to a list of 255 pre-defined colours. These colours above colour 16 are not available on devices with less than 256 displayable colours.

Text

All text is stored as STANDARD style in blocks of up to 32 characters. This only shows as a problem if the font is changed to a proportional font when being imported.

Image File Formats (BMP, XWD, SUNRAS)

Nomination Routines

	Nomination Routine	Description of device
A	gXwd()	X Windows Dump
B	gBmp()	Windows Bitmap Format
C	gSunras()	SUN Microsystems Raster File

Device Characteristics

	A, B & C
Maximum Width (mm)	23119.29mm
Maximum Height (mm)	11559.47mm
Default Width (mm)	338mm
Default Height (mm)	254mm
Resolution	65536 x 32768 (960x720 default)
Colours / Pens	256
Colour Palette	Dynamic (default) or Direct
Broken Linestyles	No
Drawing Modes	No
Thick Lines	No
Line Ends	No
Arcs	No
Symbols	No
Fonts	No
Character Sizes	4 Pseudo-hardware in multiples of 1.5mm square
Character Angles	0 or 90°
Italic Characters	No
Polygonal Filling	No
Image Handling	Full Colour and Output only
Clipping	No
Window/Device Titling	No

X Windows Dump

These are image dumps used on Unix systems supporting X windows. Normally the files are created using the Unix command xwd and read in using the corresponding command xwud (X Windows UnDump). Files created by this driver can be created using the nomination routine gXwd(), which will create an image file at 72 dpi. The resulting file can be read into a third party application or displayed using the following Unix command;

xwud -in xwd.out

where xwd.out is the image file created by the driver.

Windows Bitmap

These are image files used on PCs under Windows and OS/2. Files created by this driver using the nomination routine gBmp() will create an image file at 72 dpi. The resulting file can be read into many third party applications using a simple image import filter.

Note: To create a BMP file from an OpenGL 3D picture, the nomination routine gWoglpp() must be used.

SUN Raster File

These are similar to screen dumps from SUN workstations. Files created by this driver using the nomination routine gSunras() will create an image at 72dpi. The resulting file can be read by the SUN application imagetool and a limited number of third party applications.

Image Size

The image size can be enquired using the GINO command gEnqDrawingLimits() or gEnqMaxDrawingLimits() but these values are returned as either millimetres or the current drawing units. The image will have a size defined in pixels, this value being calculated by using the resolution value of 72 dots per inch or by calling gEnqPixelResolution(). The image size can be changed using the command gSetDrawingLimits(), but the size must be in current drawing units.

Multiple Frames

The driver does not recognise multiple images within the application so only the first image will be stored. (i.e. up to the first call to gNewDrawing() after any drawing routines).

Intermediate Vector File

This driver uses an internal GINO vector to raster pre-processor to create the output for the device which contains a fixed sized memory area into which the image is rasterized.

JPEG File Interchange Format (JPG)

Nomination Routines

	Nomination Routine	Description of Device
A	gJpeg()	JPEG File Interchange Format

Device Characteristics

	A
Maximum Width (mm)	21229.29mm
Maximum Height (mm)	1159.47mm
Default Width (mm)	338mm
Default Height (mm)	254mm
Resolution	65536 x 32768 (960x720 default)
Colours / Pens	256
Colour Palette	Direct
Broken Linestyles	No
Drawing Modes	No
Thick Lines	No
Line Ends	No
Arcs	No
Symbols	No
Fonts	No
Character Sizes	4 Pseudo-hardware in multiples of 1.5mm square
Character Angles	0 or 90°
Italic Characters	No
Polygonal Filling	No
Image Handling	Full Colour and Output only
Clipping	No
Window/Device Titling	No

JPEG File Interchange Format

The GINO driver is based on the work of the Independent JPEG Group for the software to carry out the actual compression.

Image Size

The image size can be enquired using the GINO routine gEnqDrawingLimits() or gEnqMaxDrawingLimits() but these values are returned as either millimetres or the current drawing units. The image will have a size defined in pixels, this value being calculated by using the resolution value of 72 dots per inch or by calling gEnqPixelResolution(). The image size can be changed using the command gSetDrawingLimits(), but the size must be in current drawing units.

Image Quality

The IPAPTY argument to the routine gSetDrawingLimits() is used to set the compression quality. Quality can range from 1 (worst) to 100 (best) with a default setting of 75.

Multiple Frames

The driver does not recognise multiple images within the application so only the first image will be stored. (i.e. up to the first call to gNewDrawing() after any drawing routines).

Intermediate Vector File

This driver uses an internal GINO vector to raster pre-processor to create the output for the device which contains a fixed sized memory area into which the image is rasterized.

PNG Portable Network Graphics (PNG)

Nomination Routines

	Nomination Routine	Description of Device
A	gPng()	Portable Network Graphics Format

Device Characteristics

	A
Maximum Width (mm)	23119.29mm
Maximum Height (mm)	1159.47mm
Default Width (mm)	338mm
Default Height (mm)	254mm
Resolution	65536 x 32768 (960x720 default)
Colours / Pens	256
Colour Palette	Direct
Broken Linestyles	No
Drawing Modes	No
Thick Lines	No
Line Ends	No
Arcs	No
Symbols	No
Fonts	No
Character Sizes	4 Pseudo-hardware in multiples of 1.5mm square

	A
Character Angles	0 or 90°
Italic Characters	No
Polygonal Filling	No
Image Handling	Full Colour and Output only
Clipping	No
Window/Device Titling	No

PNG (Portable Network Graphics) Format

The GINO driver is based on the work of the PNG Development Group for the software to carry out the actual compression.

Image Size

The image size can be enquired using the GINO command gEnqDrawingLimits() or gEnqMaxDrawingLimits() but these values are returned as either millimetres or the current drawing units. The image will have a size defined in pixels, this value being calculated by using the resolution value of 72 dots per inch or by calling gEnqPixelResolution(). The image size can be changed using the command gSetDrawingLimits(), but the size must be in current drawing units.

Multiple Frames

The driver does not recognise multiple images within the application so only the first image will be stored. (i.e. up to the first call to gNewDrawing() after any drawing routines).

Intermediate Vector File

This driver uses an internal GINO vector to raster pre-processor to create the output for the device which contains a fixed sized memory area into which the image is rasterized.

SAVDRA and SAVPIC Metafile

Device Nominations

	Nomination Routine	Description of Device
A	gSavdra()	GINO's proprietary metafile (Drawings that have one or more pictures)
B	gSavpic()	GINO's proprietary metafile (To create segment libraries of objects)

Device Characteristics

	A & B
Maximum Width	No limit
Maximum Height	No limit
Default Width (mm)	400.0 (-200.0 to +200.0)
Default Height (mm)	400.0 (-200.0 to +200.0)
Colours / Pens	255
Colour Palette	Dynamic
Broken Linestyles	16
Drawing Modes	No
Thick Lines	Yes
Line Ends	Yes
Arcs	Yes
Symbols	Yes
Fonts	1
Character Sizes	Any & 4 Pseudo-hardware in multiples of 1.5mm square
Character Angles	Any
Italic Characters	Any angle
Polygonal Filling	Yes (Solid and any hatch style)
Segments	Yes
Image Handling	No
Clipping	No
Window/Device Titling	Yes

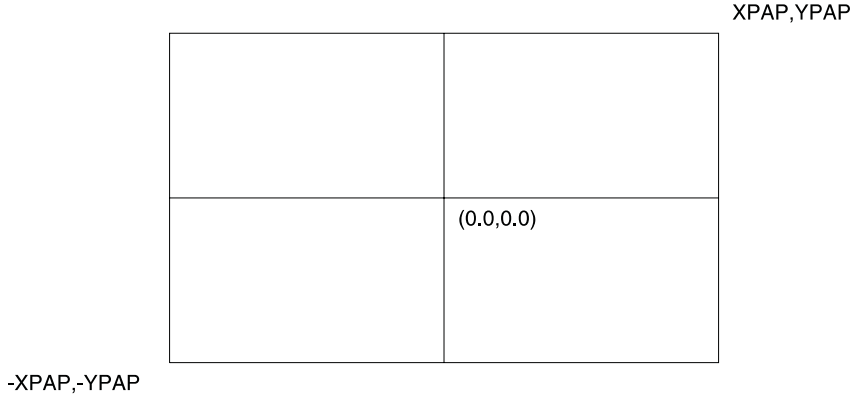
There are two nomination routines for the SAVDRA metafile depending on the intended use of the picture information.

In general, gSavdra() is used for storing complete drawings. All the current drawing modes are stored and so the same drawing can be reproduced exactly as specified on any other device.

Alternatively, gSavpic() is designed to store picture segments for subsequent use as library objects. A minimum of information is stored in the file, this being the vector part of the segment (i.e. coordinates). It is assumed that qualifiers such as object size and drawing attributes will be provided by the interpreting program.

Device Limits

Unlike most device drivers, the drawing area has its origin at the centre. Therefore setting the drawing limits using the routine gSetDrawingLimits(), defines limits that represent the positive quadrant of the complete drawing area. For example, setting the limits to **limit.xpap** by **limit.ypap** defines an effective drawing area of 2***xpap** by 2***ypap**. The device limits are thus defined to be -xpap, +xpap and -ypap, +ypap.



Savdra Device Limits

The routines gSetWindow2D() or gSetWindow3D() can be used to restrict output to the positive quadrant of the complete drawing area.

Metafile Composition

Multiple drawings in the gSavdra() format metafile can be separated using the gNewDrawing() routine.

Whilst most appropriate in the gSavpic() format, either form can contain multiple segments using the gOpenSeg() and gCloseSeg() routine. Any drawing not contained in a segment is added to segment zero.

File Format

The SAVDRA metafile records all graphical data generated by GINO. The file is written in records of up to 72 characters according to the ANSI printable character set (ie, ASCII codes 32-126).

Windows Metafile (WMF)

Device Nominations

	Nomination Routine	Description of Device
A	gWmf (idpi, iwid, iheight)	Standard metafile format
B	gWmfp(idpi, ixoff, iyoff, iwidth, iheight)	Placeable metafile format

Device Characteristics

	A and B
Maximum Width	unlimited
Maximum Height	unlimited
Default Width	**width** dots
Default Height	**height** dots
Colours / Pens	255
Colour Palette	Static (default) or Direct
Broken Line Styles	No
Drawing Modes	10 (XOR)
Thick Lines	Yes
Line Ends	No
Arcs	No
Symbols	No
Fonts	100, 101, 102, 150 (Arial), 151 (TimesNewRoman) and Weights -7 to 7
Character Sizes	Any + 4 Pseudo-hardware in multiples of 8 pixels square
Character Angles	Any
Italic Characters	Yes (between 10 and 20°)
Polygonal Filling	Rectangles and single polygons - solid only
Segments	No
Image Handling	Output only
Clipping	No

The WMF file format is used extensively within Microsoft Windows applications for storing vector based pictures. There are two formats, standard and placeable each of which can be generated on any platform with the two nomination routines provided with this driver.

The standard metafile (gWmf()) contains no scaling information and the **dpi** argument is required only to supply a scaling factor for GINO drawing purposes. Thus the drawing area as far as GINO is concerned is **wid x height** dots with **dpi** dots per inch, but the metafile simply contains a drawing area of **wid x height** dots. All arguments are integer.

e.g. If your drawing is approximately A4 size (11" x 8"), and you will be viewing the resulting file in VGA mode (640 x 480), an example call would be:

gWmf(60,640,480)

The placeable metafile (gWmfp()) contains a prefix header containing offset, size and scaling information and this format is required by many Windows applications. **xoff, yoff** specify the offset from the top-left corner and by default would be set to 0,0.

Multiple Frames

The WMF file format does not recognise multiple frames and therefore gNewDrawing() has no effect.

Character Fonts

The final form of the font face will depend on the typefaces available when interpreting the WMF file.

Appendix C

FONT TABLES

Font Tables Introduction

This section contains tables for the registered software and symbol fonts that are accessible through the gSetCharFont() and gDrawMarker() routines. Details of hardware fonts that may be available on the current output device are found in Appendix B.

The Font Tables

Roman Duplex **Software Font 2**

2000 0	2016 16	2032 32	0 2048 48	@ 2064 64	P 2080 80	` 2096 96	p 2112 112
2001 1	2017 17	! 2033 33	1 2049 49	A 2065 65	Q 2081 81	a 2097 97	q 2113 113
2002 2	2018 18	" 2034 34	2 2050 50	B 2066 66	R 2082 82	b 2098 98	r 2114 114
2003 3	2019 19	# 2035 35	3 2051 51	C 2067 67	S 2083 83	c 2099 99	s 2115 115
2004 4	2020 20	$ 2036 36	4 2052 52	D 2068 68	T 2084 84	d 2100 100	t 2116 116
2005 5	2021 21	% 2037 37	5 2053 53	E 2069 69	U 2085 85	e 2101 101	u 2117 117
2006 6	2022 22	& 2038 38	6 2054 54	F 2070 70	V 2086 86	f 2102 102	v 2118 118
2007 7	2023 23	' 2039 39	7 2055 55	G 2071 71	W 2087 87	g 2103 103	w 2119 119
2008 8	2024 24	(2040 40	8 2056 56	H 2072 72	X 2088 88	h 2104 104	x 2120 120
2009 9	ä 2025 25	) 2041 41	9 2057 57	I 2073 73	Y 2089 89	i 2105 105	y 2121 121
2010 10	ö 2026 26	* 2042 42	: 2058 58	J 2074 74	Z 2090 90	j 2106 106	z 2122 122
2011 11	ü 2027 27	+ 2043 43	; 2059 59	K 2075 75	[2091 91	k 2107 107	{ 2123 123
2012 12	ß 2028 28	, 2044 44	< 2060 60	L 2076 76	\ 2092 92	l 2108 108	| 2124 124
2013 13	Ä 2029 29	– 2045 45	= 2061 61	M 2077 77	] 2093 93	m 2109 109	} 2125 125
2014 14	Ö 2030 30	. 2046 46	> 2062 62	N 2078 78	^ 2094 94	n 2110 110	2126 126
2015 15	Ü 2031 31	/ 2047 47	? 2063 63	O 2079 79	_ 2095 95	o 2111 111	2127 127

Roman Complex **Software Font 3**

3000 0	3016 16	3032 32	0 3048 48	@ 3064 64	P 3080 80	` 3096 96	p 3112 112
3001 1	3017 17	! 3033 33	1 3049 49	A 3065 65	Q 3081 81	a 3097 97	q 3113 113
3002 2	3018 18	" 3034 34	2 3050 50	B 3066 66	R 3082 82	b 3098 98	r 3114 114
3003 3	3019 19	# 3035 35	3 3051 51	C 3067 67	S 3083 83	c 3099 99	s 3115 115
3004 4	3020 20	$ 3036 36	4 3052 52	D 3068 68	T 3084 84	d 3100 100	t 3116 116
3005 5	3021 21	% 3037 37	5 3053 53	E 3069 69	U 3085 85	e 3101 101	u 3117 117
3006 6	3022 22	& 3038 38	6 3054 54	F 3070 70	V 3086 86	f 3102 102	v 3118 118
3007 7	3023 23	' 3039 39	7 3055 55	G 3071 71	W 3087 87	g 3103 103	w 3119 119
3008 8	3024 24	(3040 40	8 3056 56	H 3072 72	X 3088 88	h 3104 104	x 3120 120
3009 9	ä 3025 25	) 3041 41	9 3057 57	I 3073 73	Y 3089 89	i 3105 105	y 3121 121
3010 10	ö 3026 26	* 3042 42	: 3058 58	J 3074 74	Z 3090 90	j 3106 106	z 3122 122
3011 11	ü 3027 27	+ 3043 43	; 3059 59	K 3075 75	[3091 91	k 3107 107	{ 3123 123
3012 12	ß 3028 28	, 3044 44	< 3060 60	L 3076 76	\ 3092 92	l 3108 108	| 3124 124
3013 13	Ä 3029 29	– 3045 45	= 3061 61	M 3077 77	] 3093 93	m 3109 109	} 3125 125
3014 14	Ö 3030 30	. 3046 46	> 3062 62	N 3078 78	^ 3094 94	n 3110 110	3126 126
3015 15	Ü 3031 31	/ 3047 47	? 3063 63	O 3079 79	_ 3095 95	o 3111 111	3127 127

Roman Triplex **Software Font 4**

4000 0	4016 16	4032 32	0 4048 48	@ 4064 64	P 4080 80	` 4096 96	p 4112 112
4001 1	4017 17	! 4033 33	1 4049 49	A 4065 65	Q 4081 81	a 4097 97	q 4113 113
4002 2	4018 18	" 4034 34	2 4050 50	B 4066 66	R 4082 82	b 4098 98	r 4114 114
4003 3	4019 19	# 4035 35	3 4051 51	C 4067 67	S 4083 83	c 4099 99	s 4115 115
4004 4	4020 20	$ 4036 36	4 4052 52	D 4068 68	T 4084 84	d 4100 100	t 4116 116
4005 5	4021 21	% 4037 37	5 4053 53	E 4069 69	U 4085 85	e 4101 101	u 4117 117
4006 6	4022 22	& 4038 38	6 4054 54	F 4070 70	V 4086 86	f 4102 102	v 4118 118
4007 7	4023 23	' 4039 39	7 4055 55	G 4071 71	W 4087 87	g 4103 103	w 4119 119
4008 8	4024 24	(4040 40	8 4056 56	H 4072 72	X 4088 88	h 4104 104	x 4120 120
4009 9	ä 4025 25	) 4041 41	9 4057 57	I 4073 73	Y 4089 89	i 4105 105	y 4121 121
4010 10	ö 4026 26	* 4042 42	: 4058 58	J 4074 74	Z 4090 90	j 4106 106	z 4122 122
4011 11	ü 4027 27	+ 4043 43	; 4059 59	K 4075 75	[4091 91	k 4107 107	{ 4123 123
4012 12	β 4028 28	, 4044 44	< 4060 60	L 4076 76	\ 4092 92	l 4108 108	| 4124 124
4013 13	Ä 4029 29	- 4045 45	= 4061 61	M 4077 77	] 4093 93	m 4109 109	} 4125 125
4014 14	Ö 4030 30	. 4046 46	> 4062 62	N 4078 78	^ 4094 94	n 4110 110	4126 126
4015 15	Ü 4031 31	/ 4047 47	? 4063 63	O 4079 79	_ 4095 95	o 4111 111	🔔 4127 127

Italic Complex **Software Font 5**

5000 0	5016 16	5032 32	*0* 5048 48	@ 5064 64	*P* 5080 80	` 5096 96	*p* 5112 112
5001 1	5017 17	*!* 5033 33	*1* 5049 49	*A* 5065 65	*Q* 5081 81	*a* 5097 97	*q* 5113 113
5002 2	5018 18	" 5034 34	*2* 5050 50	*B* 5066 66	*R* 5082 82	*b* 5098 98	*r* 5114 114
5003 3	5019 19	# 5035 35	*3* 5051 51	*C* 5067 67	*S* 5083 83	*c* 5099 99	*s* 5115 115
5004 4	5020 20	$ 5036 36	*4* 5052 52	*D* 5068 68	*T* 5084 84	*d* 5100 100	*t* 5116 116
5005 5	5021 21	% 5037 37	*5* 5053 53	*E* 5069 69	*U* 5085 85	*e* 5101 101	*u* 5117 117
5006 6	5022 22	& 5038 38	*6* 5054 54	*F* 5070 70	*V* 5086 86	*f* 5102 102	*v* 5118 118
5007 7	5023 23	' 5039 39	*7* 5055 55	*G* 5071 71	*W* 5087 87	*g* 5103 103	*w* 5119 119
5008 8	5024 24	(5040 40	*8* 5056 56	*H* 5072 72	*X* 5088 88	*h* 5104 104	*x* 5120 120
5009 9	*ä* 5025 25	) 5041 41	*9* 5057 57	*I* 5073 73	*Y* 5089 89	*i* 5105 105	*y* 5121 121
5010 10	*ö* 5026 26	* 5042 42	: 5058 58	*J* 5074 74	*Z* 5090 90	*j* 5106 106	*z* 5122 122
5011 11	*ü* 5027 27	+ 5043 43	; 5059 59	*K* 5075 75	[5091 91	*k* 5107 107	{ 5123 123
5012 12	*β* 5028 28	, 5044 44	< 5060 60	*L* 5076 76	\ 5092 92	*l* 5108 108	| 5124 124
5013 13	*Ä* 5029 29	- 5045 45	= 5061 61	*M* 5077 77	] 5093 93	*m* 5109 109	} 5125 125
5014 14	*Ö* 5030 30	. 5046 46	> 5062 62	*N* 5078 78	^ 5094 94	*n* 5110 110	5126 126
5015 15	*Ü* 5031 31	/ 5047 47	? 5063 63	*O* 5079 79	_ 5095 95	*o* 5111 111	🔔 5127 127

Italic Triplex — **Software Font 6**

6000 0	6016 16	6032 32	6048 0 48	6064 @ 64	6080 P 80	6096 ` 96	6112 p 112
6001 1	6017 17	6033 ! 33	6049 1 49	6065 A 65	6081 Q 81	6097 a 97	6113 q 113
6002 2	6018 18	6034 " 34	6050 2 50	6066 B 66	6082 R 82	6098 b 98	6114 r 114
6003 3	6019 19	6035 # 35	6051 3 51	6067 C 67	6083 S 83	6099 c 99	6115 s 115
6004 4	6020 20	6036 $ 36	6052 4 52	6068 D 68	6084 T 84	6100 d 100	6116 t 116
6005 5	6021 21	6037 % 37	6053 5 53	6069 E 69	6085 U 85	6101 e 101	6117 u 117
6006 6	6022 22	6038 & 38	6054 6 54	6070 F 70	6086 V 86	6102 f 102	6118 v 118
6007 7	6023 23	6039 ' 39	6055 7 55	6071 G 71	6087 W 87	6103 g 103	6119 w 119
6008 8	6024 24	6040 (40	6056 8 56	6072 H 72	6088 X 88	6104 h 104	6120 x 120
6009 9	6025 ä 25	6041) 41	6057 9 57	6073 I 73	6089 Y 89	6105 i 105	6121 y 121
6010 10	6026 ö 26	6042 * 42	6058 : 58	6074 J 74	6090 Z 90	6106 j 106	6122 z 122
6011 11	6027 ü 27	6043 + 43	6059 ; 59	6075 K 75	6091 [91	6107 k 107	6123 { 123
6012 12	6028 β 28	6044 , 44	6060 < 60	6076 L 76	6092 \ 92	6108 l 108	6124 | 124
6013 13	6029 Ä 29	6045 - 45	6061 = 61	6077 M 77	6093] 93	6109 m 109	6125 } 125
6014 14	6030 Ö 30	6046 . 46	6062 > 62	6078 N 78	6094 ^ 94	6110 n 110	6126 126
6015 15	6031 Ü 31	6047 / 47	6063 ? 63	6079 O 79	6095 _ 95	6111 o 111	6127 🔔 127

Script Simplex — **Software Font 7**

7000 0	7016 16	7032 32	7048 0 48	7064 @ 64	7080 P 80	7096 ` 96	7112 p 112
7001 1	7017 17	7033 ! 33	7049 1 49	7065 A 65	7081 Q 81	7097 a 97	7113 q 113
7002 2	7018 18	7034 " 34	7050 2 50	7066 B 66	7082 R 82	7098 b 98	7114 r 114
7003 3	7019 19	7035 # 35	7051 3 51	7067 C 67	7083 S 83	7099 c 99	7115 s 115
7004 4	7020 20	7036 $ 36	7052 4 52	7068 D 68	7084 T 84	7100 d 100	7116 t 116
7005 5	7021 21	7037 % 37	7053 5 53	7069 E 69	7085 U 85	7101 e 101	7117 u 117
7006 6	7022 22	7038 & 38	7054 6 54	7070 F 70	7086 V 86	7102 f 102	7118 v 118
7007 7	7023 23	7039 ' 39	7055 7 55	7071 G 71	7087 W 87	7103 g 103	7119 w 119
7008 8	7024 24	7040 (40	7056 8 56	7072 H 72	7088 X 88	7104 h 104	7120 x 120
7009 9	7025 ä 25	7041) 41	7057 9 57	7073 I 73	7089 Y 89	7105 i 105	7121 y 121
7010 10	7026 ö 26	7042 * 42	7058 : 58	7074 J 74	7090 Z 90	7106 j 106	7122 z 122
7011 11	7027 ü 27	7043 + 43	7059 ; 59	7075 K 75	7091 [91	7107 k 107	7123 { 123
7012 12	7028 β 28	7044 , 44	7060 < 60	7076 L 76	7092 \ 92	7108 l 108	7124 | 124
7013 13	7029 Ä 29	7045 - 45	7061 = 61	7077 M 77	7093] 93	7109 m 109	7125 } 125
7014 14	7030 Ö 30	7046 . 46	7062 > 62	7078 N 78	7094 ^ 94	7110 n 110	7126 126
7015 15	7031 Ü 31	7047 / 47	7063 ? 63	7079 O 79	7095 _ 95	7111 o 111	7127 🔔 127

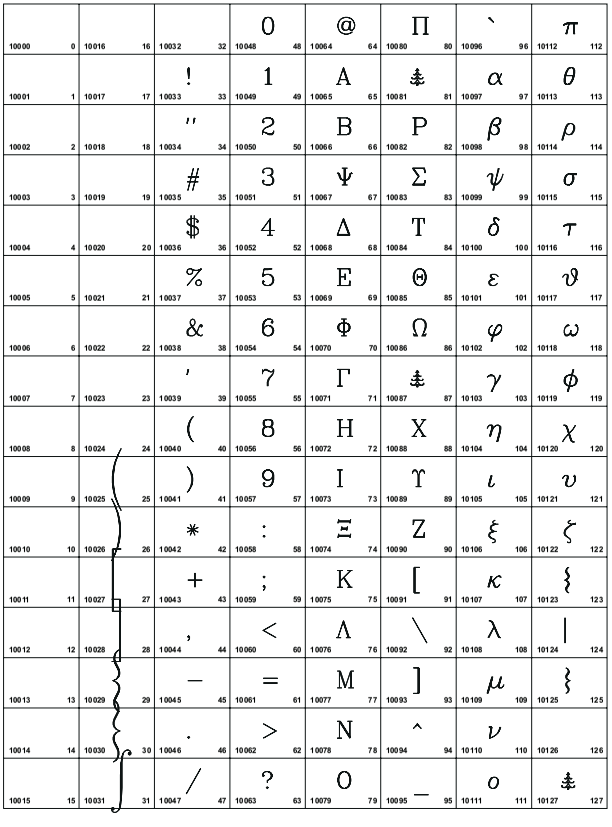
Script Complex **Software Font 8**

8000 0	8016 16	8032 32	0 8048 48	@ 8064 64	P 8080 80	` 8096 96	p 8112 112
8001 1	8017 17	! 8033 33	1 8049 49	A 8065 65	Q 8081 81	a 8097 97	q 8113 113
8002 2	8018 18	" 8034 34	2 8050 50	B 8066 66	R 8082 82	b 8098 98	r 8114 114
8003 3	8019 19	# 8035 35	3 8051 51	C 8067 67	S 8083 83	c 8099 99	s 8115 115
8004 4	8020 20	$ 8036 36	4 8052 52	D 8068 68	T 8084 84	d 8100 100	t 8116 116
8005 5	8021 21	% 8037 37	5 8053 53	E 8069 69	U 8085 85	e 8101 101	u 8117 117
8006 6	8022 22	& 8038 38	6 8054 54	F 8070 70	V 8086 86	f 8102 102	v 8118 118
8007 7	8023 23	' 8039 39	7 8055 55	G 8071 71	W 8087 87	g 8103 103	w 8119 119
8008 8	8024 24	(8040 40	8 8056 56	H 8072 72	X 8088 88	h 8104 104	x 8120 120
8009 9	ä 8025 25	) 8041 41	9 8057 57	I 8073 73	Y 8089 89	i 8105 105	y 8121 121
8010 10	ö 8026 26	* 8042 42	: 8058 58	J 8074 74	Z 8090 90	j 8106 106	z 8122 122
8011 11	ü 8027 27	+ 8043 43	; 8059 59	K 8075 75	[8091 91	k 8107 107	{ 8123 123
8012 12	ß 8028 28	, 8044 44	< 8060 60	L 8076 76	\ 8092 92	l 8108 108	| 8124 124
8013 13	Ä 8029 29	- 8045 45	= 8061 61	M 8077 77	] 8093 93	m 8109 109	} 8125 125
8014 14	Ö 8030 30	. 8046 46	> 8062 62	N 8078 78	^ 8094 94	n 8110 110	8126 126
8015 15	Ü 8031 31	/ 8047 47	? 8063 63	O 8079 79	_ 8095 95	o 8111 111	🔔 8127 127

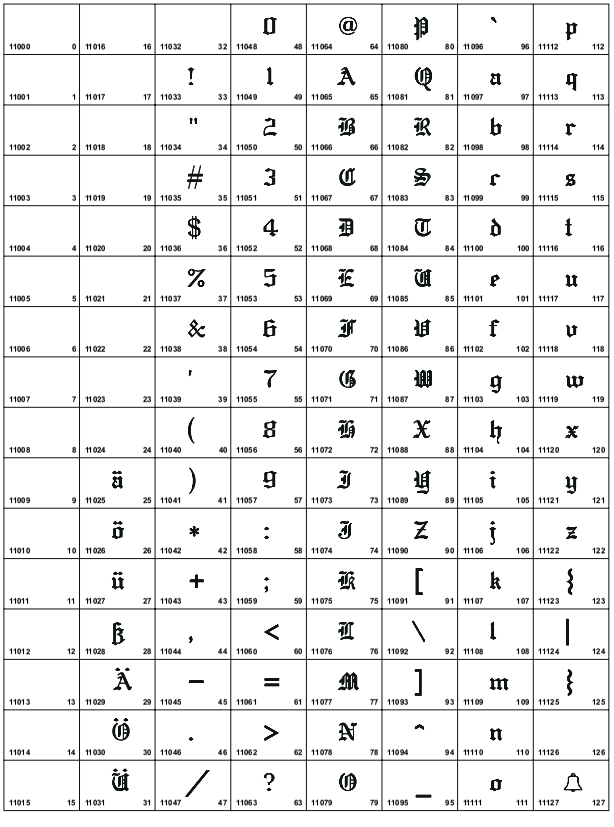
Greek Simplex **Software Font 9**

9000 0	9016 16	9032 32	0 9048 48	@ 9064 64	Π 9080 80	` 9096 96	π 9112 112
9001 1	9017 17	! 9033 33	1 9049 49	A 9065 65	♡ 9081 81	α 9097 97	θ 9113 113
9002 2	9018 18	" 9034 34	2 9050 50	B 9066 66	P 9082 82	β 9098 98	ρ 9114 114
9003 3	9019 19	# 9035 35	3 9051 51	Ψ 9067 67	Σ 9083 83	ψ 9099 99	σ 9115 115
9004 4	9020 20	$ 9036 36	4 9052 52	Δ 9068 68	T 9084 84	δ 9100 100	τ 9116 116
9005 5	9021 21	% 9037 37	5 9053 53	E 9069 69	Θ 9085 85	ε 9101 101	ϑ 9117 117
9006 6	9022 22	& 9038 38	6 9054 54	Φ 9070 70	Ω 9086 86	φ 9102 102	ω 9118 118
9007 7	9023 23	' 9039 39	7 9055 55	Γ 9071 71	♡ 9087 87	γ 9103 103	ϕ 9119 119
9008 8	9024 24	(9040 40	8 9056 56	H 9072 72	X 9088 88	η 9104 104	χ 9120 120
9009 9	9025 25	) 9041 41	9 9057 57	I 9073 73	ϒ 9089 89	ι 9105 105	υ 9121 121
9010 10	9026 26	* 9042 42	: 9058 58	Ξ 9074 74	Z 9090 90	ξ 9106 106	ζ 9122 122
9011 11	9027 27	+ 9043 43	; 9059 59	K 9075 75	[9091 91	κ 9107 107	{ 9123 123
9012 12	9028 28	, 9044 44	< 9060 60	Λ 9076 76	\ 9092 92	λ 9108 108	| 9124 124
9013 13	9029 29	- 9045 45	= 9061 61	M 9077 77	] 9093 93	μ 9109 109	} 9125 125
9014 14	9030 30	. 9046 46	> 9062 62	N 9078 78	^ 9094 94	ν 9110 110	9126 126
9015 15	9031 31	/ 9047 47	? 9063 63	O 9079 79	_ 9095 95	ο 9111 111	♡ 9127 127

Greek Complex **Software Font 10**



Gothic English **Software Font 11**



Gothic Italian

Software Font 13

0 13000	16 13016	32 13032	0 48 13048	@ 64 13064	P 80 13080	` 96 13096	p 112 13112
1 13001	17 13017	! 33 13033	1 49 13049	A 65 13065	Q 81 13081	a 97 13097	q 113 13113
2 13002	18 13018	" 34 13034	2 50 13050	B 66 13066	R 82 13082	b 98 13098	r 114 13114
3 13003	19 13019	# 35 13035	3 51 13051	C 67 13067	S 83 13083	c 99 13099	s 115 13115
4 13004	20 13020	$ 36 13036	4 52 13052	D 68 13068	T 84 13084	d 100 13100	t 116 13116
5 13005	21 13021	% 37 13037	5 53 13053	E 69 13069	U 85 13085	e 101 13101	u 117 13117
6 13006	22 13022	& 38 13038	6 54 13054	F 70 13070	V 86 13086	f 102 13102	v 118 13118
7 13007	23 13023	' 39 13039	7 55 13055	G 71 13071	W 87 13087	g 103 13103	w 119 13119
8 13008	24 13024	(40 13040	8 56 13056	H 72 13072	X 88 13088	h 104 13104	x 120 13120
9 13009	ä 25 13025	) 41 13041	9 57 13057	I 73 13073	Y 89 13089	i 105 13105	y 121 13121
10 13010	ö 26 13026	* 42 13042	: 58 13058	J 74 13074	Z 90 13090	j 106 13106	z 122 13122
11 13011	ü 27 13027	+ 43 13043	; 59 13059	K 75 13075	[91 13091	k 107 13107	{ 123 13123
12 13012	ß 28 13028	, 44 13044	< 60 13060	L 76 13076	\ 92 13092	l 108 13108	| 124 13124
13 13013	Ä 29 13029	- 45 13045	= 61 13061	M 77 13077	] 93 13093	m 109 13109	} 125 13125
14 13014	Ö 30 13030	. 46 13046	> 62 13062	N 78 13078	^ 94 13094	n 110 13110	126 13126
15 13015	Ü 31 13031	/ 47 13047	? 63 13063	O 79 13079	_ 95 13095	o 111 13111	🔔 127 13127

Gothic German

Software Font 12

0 12000	16 12016	32 12032	0 48 12048	@ 64 12064	P 80 12080	` 96 12096	p 112 12112
1 12001	17 12017	! 33 12033	1 49 12049	A 65 12065	Q 81 12081	a 97 12097	q 113 12113
2 12002	18 12018	" 34 12034	2 50 12050	B 66 12066	R 82 12082	b 98 12098	r 114 12114
3 12003	19 12019	# 35 12035	3 51 12051	C 67 12067	S 83 12083	c 99 12099	s 115 12115
4 12004	20 12020	$ 36 12036	4 52 12052	D 68 12068	T 84 12084	d 100 12100	t 116 12116
5 12005	21 12021	% 37 12037	5 53 12053	E 69 12069	U 85 12085	e 101 12101	u 117 12117
6 12006	22 12022	& 38 12038	6 54 12054	F 70 12070	V 86 12086	f 102 12102	v 118 12118
7 12007	23 12023	' 39 12039	7 55 12055	G 71 12071	W 87 12087	g 103 12103	w 119 12119
8 12008	24 12024	(40 12040	8 56 12056	H 72 12072	X 88 12088	h 104 12104	x 120 12120
9 12009	ä 25 12025	) 41 12041	9 57 12057	I 73 12073	Y 89 12089	i 105 12105	y 121 12121
10 12010	ö 26 12026	* 42 12042	: 58 12058	J 74 12074	Z 90 12090	j 106 12106	z 122 12122
11 12011	ü 27 12027	+ 43 12043	; 59 12059	K 75 12075	[91 12091	k 107 12107	{ 123 12123
12 12012	ß 28 12028	, 44 12044	< 60 12060	L 76 12076	\ 92 12092	l 108 12108	| 124 12124
13 12013	Ä 29 12029	- 45 12045	= 61 12061	M 77 12077	] 93 12093	m 109 12109	} 125 12125
14 12014	Ö 30 12030	. 46 12046	> 62 12062	N 78 12078	^ 94 12094	n 110 12110	126 12126
15 12015	Ü 31 12031	/ 47 12047	? 63 12063	O 79 12079	_ 95 12095	o 111 12111	🔔 127 12127

Cyrillic Complex — **Software Font 14**

14000 0	14016 16	14032 32	0 14048 48	@ 14064 64	П 14080 80	Я 14096 96	п 14112 112
14001 1	14017 17	! 14033 33	1 14049 49	А 14065 65	Р 14081 81	а 14097 97	р 14113 113
14002 2	14018 18	'' 14034 34	2 14050 50	Б 14066 66	С 14082 82	б 14098 98	с 14114 114
14003 3	14019 19	# 14035 35	3 14051 51	В 14067 67	Т 14083 83	в 14099 99	т 14115 115
14004 4	14020 20	$ 14036 36	4 14052 52	Г 14068 68	У 14084 84	г 14100 100	у 14116 116
14005 5	14021 21	% 14037 37	5 14053 53	Д 14069 69	Ф 14085 85	д 14101 101	ф 14117 117
14006 6	14022 22	& 14038 38	6 14054 54	Е 14070 70	Х 14086 86	е 14102 102	х 14118 118
14007 7	14023 23	' 14039 39	7 14055 55	Ж 14071 71	Ц 14087 87	ж 14103 103	ц 14119 119
14008 8	14024 24	(14040 40	8 14056 56	З 14072 72	Ч 14088 88	з 14104 104	ч 14120 120
14009 9	14025 25	) 14041 41	9 14057 57	И 14073 73	Ш 14089 89	и 14105 105	ш 14121 121
14010 10	14026 26	* 14042 42	: 14058 58	Й 14074 74	Щ 14090 90	й 14106 106	щ 14122 122
14011 11	14027 27	+ 14043 43	; 14059 59	К 14075 75	Ъ 14091 91	к 14107 107	ъ 14123 123
14012 12	я 14028 28	, 14044 44	< 14060 60	Л 14076 76	Ы 14092 92	л 14108 108	ы 14124 124
14013 13	14029 29	— 14045 45	= 14061 61	М 14077 77	Ь 14093 93	м 14109 109	ь 14125 125
14014 14	14030 30	. 14046 46	> 14062 62	Н 14078 78	Э 14094 94	н 14110 110	14126 126
14015 15	14031 31	/ 14047 47	? 14063 63	О 14079 79	Ю 14095 95	о 14111 111	ю 14127 127

Swiss Solid — **Software Font 15**

<table>
<tr><td>15000 0</td><td>15016 16</td><td>15032 32</td><td>0 15048 48</td><td>@ 15064 64</td><td>P 15080 80</td><td>‘ 15096 96</td><td>p 15112 112</td></tr>
<tr><td>15001 1</td><td>15017 17</td><td>! 15033 33</td><td>1 15049 49</td><td>A 15065 65</td><td>Q 15081 81</td><td>a 15097 97</td><td>q 15113 113</td></tr>
<tr><td>15002 2</td><td>15018 18</td><td>" 15034 34</td><td>2 15050 50</td><td>B 15066 66</td><td>R 15082 82</td><td>b 15098 98</td><td>r 15114 114</td></tr>
<tr><td>15003 3</td><td>15019 19</td><td># 15035 35</td><td>3 15051 51</td><td>C 15067 67</td><td>S 15083 83</td><td>c 15099 99</td><td>s 15115 115</td></tr>
<tr><td>15004 4</td><td>15020 20</td><td>$ 15036 36</td><td>4 15052 52</td><td>D 15068 68</td><td>T 15084 84</td><td>d 15100 100</td><td>t 15116 116</td></tr>
<tr><td>15005 5</td><td>15021 21</td><td>% 15037 37</td><td>5 15053 53</td><td>E 15069 69</td><td>U 15085 85</td><td>e 15101 101</td><td>u 15117 117</td></tr>
<tr><td>15006 6</td><td>15022 22</td><td>& 15038 38</td><td>6 15054 54</td><td>F 15070 70</td><td>V 15086 86</td><td>f 15102 102</td><td>v 15118 118</td></tr>
<tr><td>15007 7</td><td>15023 23</td><td>’ 15039 39</td><td>7 15055 55</td><td>G 15071 71</td><td>W 15087 87</td><td>g 15103 103</td><td>w 15119 119</td></tr>
<tr><td>15008 8</td><td>15024 24</td><td>(15040 40</td><td>8 15056 56</td><td>H 15072 72</td><td>X 15088 88</td><td>h 15104 104</td><td>x 15120 120</td></tr>
<tr><td>15009 9</td><td>ä 15025 25</td><td>) 15041 41</td><td>9 15057 57</td><td>I 15073 73</td><td>Y 15089 89</td><td>i 15105 105</td><td>y 15121 121</td></tr>
<tr><td>15010 10</td><td>ö 15026 26</td><td>* 15042 42</td><td>: 15058 58</td><td>J 15074 74</td><td>Z 15090 90</td><td>j 15106 106</td><td>z 15122 122</td></tr>
<tr><td>15011 11</td><td>ü 15027 27</td><td>+ 15043 43</td><td>; 15059 59</td><td>K 15075 75</td><td>[15091 91</td><td>k 15107 107</td><td>{ 15123 123</td></tr>
<tr><td>15012 12</td><td>ß 15028 28</td><td>, 15044 44</td><td>< 15060 60</td><td>L 15076 76</td><td>\ 15092 92</td><td>l 15108 108</td><td>| 15124 124</td></tr>
<tr><td>15013 13</td><td>Ä 15029 29</td><td>- 15045 45</td><td>= 15061 61</td><td>M 15077 77</td><td>] 15093 93</td><td>m 15109 109</td><td>} 15125 125</td></tr>
<tr><td>15014 14</td><td>Ö 15030 30</td><td>. 15046 46</td><td>> 15062 62</td><td>N 15078 78</td><td>^ 15094 94</td><td>n 15110 110</td><td>15126 126</td></tr>
<tr><td>15015 15</td><td>Ü 15031 31</td><td>/ 15047 47</td><td>? 15063 63</td><td>O 15079 79</td><td>_ 15095 95</td><td>o 15111 111</td><td>£ 15127 127</td></tr>
</table>

Dutch Solid **Software Font 16**

16000 0	16016 16	16032 32	0 16048 48	@ 16064 64	P 16080 80	‘ 16096 96	p 16112 112
16001 1	16017 17	! 16033 33	1 16049 49	A 16065 65	Q 16081 81	a 16097 97	q 16113 113
16002 2	16018 18	" 16034 34	2 16050 50	B 16066 66	R 16082 82	b 16098 98	r 16114 114
16003 3	16019 19	# 16035 35	3 16051 51	C 16067 67	S 16083 83	c 16099 99	s 16115 115
16004 4	16020 20	$ 16036 36	4 16052 52	D 16068 68	T 16084 84	d 16100 100	t 16116 116
16005 5	16021 21	% 16037 37	5 16053 53	E 16069 69	U 16085 85	e 16101 101	u 16117 117
16006 6	16022 22	& 16038 38	6 16054 54	F 16070 70	V 16086 86	f 16102 102	v 16118 118
16007 7	16023 23	’ 16039 39	7 16055 55	G 16071 71	W 16087 87	g 16103 103	w 16119 119
16008 8	16024 24	(16040 40	8 16056 56	H 16072 72	X 16088 88	h 16104 104	x 16120 120
16009 9	ä 16025 25	) 16041 41	9 16057 57	I 16073 73	Y 16089 89	i 16105 105	y 16121 121
16010 10	ö 16026 26	* 16042 42	: 16058 58	J 16074 74	Z 16090 90	j 16106 106	z 16122 122
16011 11	ü 16027 27	+ 16043 43	; 16059 59	K 16075 75	[16091 91	k 16107 107	{ 16123 123
16012 12	ß 16028 28	, 16044 44	< 16060 60	L 16076 76	\ 16092 92	l 16108 108	| 16124 124
16013 13	Ä 16029 29	- 16045 45	= 16061 61	M 16077 77	] 16093 93	m 16109 109	} 16125 125
16014 14	Ö 16030 30	. 16046 46	> 16062 62	N 16078 78	^ 16094 94	n 16110 110	16126 126
16015 15	Ü 16031 31	/ 16047 47	? 16063 63	O 16079 79	_ 16095 95	o 16111 111	£ 16127 127

Western **Software Font 17**

17000 0	17016 16	17032 32	0 17048 48	@ 17064 64	P 17080 80	` 17096 96	P 17112 112
17001 1	17017 17	! 17033 33	1 17049 49	A 17065 65	Q 17081 81	A 17097 97	Q 17113 113
17002 2	17018 18	˝ 17034 34	2 17050 50	B 17066 66	R 17082 82	B 17098 98	R 17114 114
17003 3	17019 19	# 17035 35	3 17051 51	C 17067 67	S 17083 83	C 17099 99	S 17115 115
17004 4	17020 20	$ 17036 36	4 17052 52	D 17068 68	T 17084 84	D 17100 100	T 17116 116
17005 5	17021 21	⁒ 17037 37	5 17053 53	E 17069 69	U 17085 85	E 17101 101	U 17117 117
17006 6	17022 22	& 17038 38	6 17054 54	F 17070 70	V 17086 86	F 17102 102	V 17118 118
17007 7	17023 23	´ 17039 39	7 17055 55	G 17071 71	W 17087 87	G 17103 103	W 17119 119
17008 8	17024 24	(17040 40	8 17056 56	H 17072 72	X 17088 88	H 17104 104	X 17120 120
17009 9	Ä 17025 25	) 17041 41	9 17057 57	I 17073 73	Y 17089 89	I 17105 105	Y 17121 121
17010 10	Ö 17026 26	* 17042 42	: 17058 58	J 17074 74	Z 17090 90	J 17106 106	Z 17122 122
17011 11	Ü 17027 27	+ 17043 43	; 17059 59	K 17075 75	[17091 91	K 17107 107	{ 17123 123
17012 12	ß 17028 28	, 17044 44	< 17060 60	L 17076 76	\ 17092 92	L 17108 108	| 17124 124
17013 13	Ä 17029 29	– 17045 45	= 17061 61	M 17077 77	] 17093 93	M 17109 109	} 17125 125
17014 14	Ö 17030 30	. 17046 46	> 17062 62	N 17078 78	^ 17094 94	N 17110 110	17126 126
17015 15	Ü 17031 31	/ 17047 47	? 17063 63	O 17079 79	_ 17095 95	O 17111 111	◎ 17127 127

Computer
Software Font 18
Display
Software Font 19

Latin **Software Font 20** **Greek/Symbol Font** **Software Font 21**

	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
20000	0	20016	16	20032	32	20048	48	20064	64	20080	80	20096	96	20112	112	
20001	1	20017	17	20033	33	20049	49	20065	65	20081	81	20097	97	20113	113	
20002	2	20018	18	20034	34	20050	50	20066	66	20082	82	20098	98	20114	114	
20003	3	20019	19	20035	35	20051	51	20067	67	20083	83	20099	99	20115	115	
20004	4	20020	20	20036	36	20052	52	20068	68	20084	84	20100	100	20116	116	
20005	5	20021	21	20037	37	20053	53	20069	69	20085	85	20101	101	20117	117	
20006	6	20022	22	20038	38	20054	54	20070	70	20086	86	20102	102	20118	118	
20007	7	20023	23	20039	39	20055	55	20071	71	20087	87	20103	103	20119	119	
20008	8	20024	24	20040	40	20056	56	20072	72	20088	88	20104	104	20120	120	
20009	9	20025	25	20041	41	20057	57	20073	73	20089	89	20105	105	20121	121	
20010	10	20026	26	20042	42	20058	58	20074	74	20090	90	20106	106	20122	122	
20011	11	20027	27	20043	43	20059	59	20075	75	20091	91	20107	107	20123	123	
20012	12	20028	28	20044	44	20060	60	20076	76	20092	92	20108	108	20124	124	
20013	13	20029	29	20045	45	20061	61	20077	77	20093	93	20109	109	20125	125	
20014	14	20030	30	20046	46	20062	62	20078	78	20094	94	20110	110	20126	126	
20015	15	20031	31	20047	47	20063	63	20079	79	20095	95	20111	111	20127	127	

	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
21000	0	21016	16	21032	32	21048	48	21064	64	21080	80	21096	96	21112	112	
21001	1	21017	17	21033	33	21049	49	21065	65	21081	81	21097	97	21113	113	
21002	2	21018	18	21034	34	21050	50	21066	66	21082	82	21098	98	21114	114	
21003	3	21019	19	21035	35	21051	51	21067	67	21083	83	21099	99	21115	115	
21004	4	21020	20	21036	36	21052	52	21068	68	21084	84	21100	100	21116	116	
21005	5	21021	21	21037	37	21053	53	21069	69	21085	85	21101	101	21117	117	
21006	6	21022	22	21038	38	21054	54	21070	70	21086	86	21102	102	21118	118	
21007	7	21023	23	21039	39	21055	55	21071	71	21087	87	21103	103	21119	119	
21008	8	21024	24	21040	40	21056	56	21072	72	21088	88	21104	104	21120	120	
21009	9	21025	25	21041	41	21057	57	21073	73	21089	89	21105	105	21121	121	
21010	10	21026	26	21042	42	21058	58	21074	74	21090	90	21106	106	21122	122	
21011	11	21027	27	21043	43	21059	59	21075	75	21091	91	21107	107	21123	123	
21012	12	21028	28	21044	44	21060	60	21076	76	21092	92	21108	108	21124	124	
21013	13	21029	29	21045	45	21061	61	21077	77	21093	93	21109	109	21125	125	
21014	14	21030	30	21046	46	21062	62	21078	78	21094	94	21110	110	21126	126	
21015	15	21031	31	21047	47	21063	63	21079	79	21095	95	21111	111	21127	127	

Greek Font 2 **Software Font 22**

22000 0	22016 16	22032 32	0 22048 48	@ 22064 64	Π 22080 80	` 22096 96	π 22112 112
22001 1	22017 17	! 22033 33	1 22049 49	Α 22065 65	◈ 22081 81	α 22097 97	θ 22113 113
22002 2	22018 18	" 22034 34	2 22050 50	Β 22066 66	Ρ 22082 82	β 22098 98	ρ 22114 114
22003 3	22019 19	# 22035 35	3 22051 51	Ψ 22067 67	Σ 22083 83	ϕ 22099 99	σ 22115 115
22004 4	22020 20	$ 22036 36	4 22052 52	Δ 22068 68	Τ 22084 84	δ 22100 100	τ 22116 116
22005 5	22021 21	% 22037 37	5 22053 53	Ε 22069 69	Θ 22085 85	ε 22101 101	ϑ 22117 117
22006 6	22022 22	& 22038 38	6 22054 54	Φ 22070 70	Ω 22086 86	φ 22102 102	ω 22118 118
22007 7	22023 23	' 22039 39	7 22055 55	Γ 22071 71	◈ 22087 87	γ 22103 103	ς 22119 119
22008 8	22024 24	(22040 40	8 22056 56	Η 22072 72	Χ 22088 88	η 22104 104	χ 22120 120
22009 9	◈ 22025 25	) 22041 41	9 22057 57	Ι 22073 73	Υ 22089 89	ι 22105 105	υ 22121 121
22010 10	◈ 22026 26	* 22042 42	: 22058 58	Ξ 22074 74	Ζ 22090 90	ξ 22106 106	ζ 22122 122
22011 11	◈ 22027 27	+ 22043 43	; 22059 59	Κ 22075 75	[22091 91	ϰ 22107 107	{ 22123 123
22012 12	◈ 22028 28	, 22044 44	< 22060 60	Λ 22076 76	\ 22092 92	λ 22108 108	| 22124 124
22013 13	◈ 22029 29	- 22045 45	= 22061 61	Μ 22077 77	] 22093 93	μ 22109 109	} 22125 125
22014 14	◈ 22030 30	. 22046 46	> 22062 62	Ν 22078 78	^ 22094 94	ν 22110 110	22126 126
22015 15	◈ 22031 31	/ 22047 47	? 22063 63	Ο 22079 79	_ 22095 95	ο 22111 111	◈ 22127 127

Greek Font 3 **Software Font 23**

23000 0	23016 16	23032 32	0 23048 48	@ 23064 64	Π 23080 80	` 23096 96	π 23112 112
23001 1	23017 17	! 23033 33	1 23049 49	Α 23065 65	⌖ 23081 81	α 23097 97	θ 23113 113
23002 2	23018 18	" 23034 34	2 23050 50	Β 23066 66	Ρ 23082 82	β 23098 98	ρ 23114 114
23003 3	23019 19	# 23035 35	3 23051 51	Ψ 23067 67	Σ 23083 83	ψ 23099 99	σ 23115 115
23004 4	23020 20	$ 23036 36	4 23052 52	Δ 23068 68	Τ 23084 84	δ 23100 100	τ 23116 116
23005 5	23021 21	% 23037 37	5 23053 53	Ε 23069 69	Θ 23085 85	ε 23101 101	ϑ 23117 117
23006 6	23022 22	& 23038 38	6 23054 54	Φ 23070 70	Ω 23086 86	φ 23102 102	ω 23118 118
23007 7	23023 23	' 23039 39	7 23055 55	Γ 23071 71	⌖ 23087 87	γ 23103 103	ϕ 23119 119
23008 8	23024 24	(23040 40	8 23056 56	Η 23072 72	Χ 23088 88	η 23104 104	χ 23120 120
23009 9	⌖ 23025 25	) 23041 41	9 23057 57	Ι 23073 73	Υ 23089 89	ι 23105 105	υ 23121 121
23010 10	⌖ 23026 26	* 23042 42	: 23058 58	Ξ 23074 74	Ζ 23090 90	ξ 23106 106	ζ 23122 122
23011 11	⌖ 23027 27	+ 23043 43	; 23059 59	Κ 23075 75	[23091 91	κ 23107 107	{ 23123 123
23012 12	⌖ 23028 28	, 23044 44	< 23060 60	Λ 23076 76	\ 23092 92	λ 23108 108	| 23124 124
23013 13	⌖ 23029 29	- 23045 45	= 23061 61	Μ 23077 77	] 23093 93	μ 23109 109	} 23125 125
23014 14	⌖ 23030 30	. 23046 46	> 23062 62	Ν 23078 78	^ 23094 94	ν 23110 110	23126 126
23015 15	⌖ 23031 31	/ 23047 47	? 23063 63	Ο 23079 79	_ 23095 95	ο 23111 111	⌖ 23127 127

Greek Font 4 Software Font 24

0 24000	16 24016	32 24032	48 24048	64 24064	80 Π 24080	96 24096	112 π 24112
1 24001	17 24017	33 24033	49 24049	65 A 24065	81 24081	97 α 24097	113 θ 24113
2 24002	18 24018	34 24034	50 24050	66 B 24066	82 P 24082	98 β 24098	114 ρ 24114
3 24003	19 24019	35 24035	51 24051	67 Ψ 24067	83 Σ 24083	99 ψ 24099	115 σ 24115
4 24004	20 24020	36 24036	52 24052	68 Δ 24068	84 T 24084	100 δ 24100	116 τ 24116
5 24005	21 24021	37 24037	53 24053	69 E 24069	85 Θ 24085	101 ε 24101	117 ϑ 24117
6 24006	22 24022	38 24038	54 24054	70 Φ 24070	86 Ω 24086	102 φ 24102	118 ω 24118
7 24007	23 24023	39 24039	55 24055	71 Γ 24071	87 24087	103 γ 24103	119 ϕ 24119
8 24008	24 24024	40 24040	56 24056	72 H 24072	88 X 24088	104 η 24104	120 χ 24120
9 24009	25 24025	41 24041	57 24057	73 I 24073	89 Y 24089	105 ι 24105	121 υ 24121
10 24010	26 24026	42 24042	58 24058	74 Ξ 24074	90 Z 24090	106 ξ 24106	122 ζ 24122
11 24011	27 24027	43 24043	59 24059	75 K 24075	91 24091	107 κ 24107	123 24123
12 24012	28 24028	44 24044	60 24060	76 Λ 24076	92 24092	108 λ 24108	124 24124
13 24013	29 24029	45 24045	61 24061	77 M 24077	93 24093	109 μ 24109	125 24125
14 24014	30 24030	46 24046	62 24062	78 N 24078	94 24094	110 ν 24110	126 24126
15 24015	31 24031	47 24047	63 24063	79 O 24079	95 24095	111 ο 24111	127 24127

Greek Font 5 Software Font 25

0 25000	16 25016	32 25032	48 25048	64 25064	80 Π 25080	96 25096	112 π 25112
1 25001	17 25017	33 25033	49 25049	65 A 25065	81 25081	97 α 25097	113 θ 25113
2 25002	18 25018	34 25034	50 25050	66 B 25066	82 P 25082	98 β 25098	114 ρ 25114
3 25003	19 25019	35 25035	51 25051	67 Ψ 25067	83 Σ 25083	99 ψ 25099	115 σ 25115
4 25004	20 25020	36 25036	52 25052	68 Δ 25068	84 T 25084	100 δ 25100	116 τ 25116
5 25005	21 25021	37 25037	53 25053	69 E 25069	85 Θ 25085	101 ε 25101	117 ϑ 25117
6 25006	22 25022	38 25038	54 25054	70 Φ 25070	86 Ω 25086	102 φ 25102	118 ω 25118
7 25007	23 25023	39 25039	55 25055	71 Γ 25071	87 25087	103 γ 25103	119 ϕ 25119
8 25008	24 25024	40 25040	56 25056	72 H 25072	88 X 25088	104 η 25104	120 χ 25120
9 25009	25 25025	41 25041	57 25057	73 I 25073	89 Y 25089	105 ι 25105	121 υ 25121
10 25010	26 25026	42 25042	58 25058	74 Ξ 25074	90 Z 25090	106 ξ 25106	122 ζ 25122
11 25011	27 25027	43 25043	59 25059	75 K 25075	91 25091	107 κ 25107	123 25123
12 25012	28 25028	44 25044	60 25060	76 Λ 25076	92 25092	108 λ 25108	124 25124
13 25013	29 25029	45 25045	61 25061	77 M 25077	93 25093	109 μ 25109	125 25125
14 25014	30 25030	46 25046	62 25062	78 N 25078	94 25094	110 ν 25110	126 25126
15 25015	31 25031	47 25047	63 25063	79 O 25079	95 25095	111 ο 25111	127 25127

Maths Symbols DIN 6776

Software Font 70

70000	0	70016	16	70032	32	70048	48	70064	64	70080	80	70096	96	70112	112
70001	1	70017	17	70033	33	70049	49	70065	65	70081	81	70097	97	70113	113
70002	2	70018	18	70034	34	70050	50	70066	66	70082	82	70098	98	70114	114
70003	3	70019	19	70035	35	70051	51	70067	67	70083	83	70099	99	70115	115
70004	4	70020	20	70036	36	70052	52	70068	68	70084	84	70100	100	70116	116
70005	5	70021	21	70037	37	70053	53	70069	69	70085	85	70101	101	70117	117
70006	6	70022	22	70038	38	70054	54	70070	70	70086	86	70102	102	70118	118
70007	7	70023	23	70039	39	70055	55	70071	71	70087	87	70103	103	70119	119
70008	8	70024	24	70040	40	70056	56	70072	72	70088	88	70104	104	70120	120
70009	9	70025	25	70041	41	70057	57	70073	73	70089	89	70105	105	70121	121
70010	10	70026	26	70042	42	70058	58	70074	74	70090	90	70106	106	70122	122
70011	11	70027	27	70043	43	70059	59	70075	75	70091	91	70107	107	70123	123
70012	12	70028	28	70044	44	70060	60	70076	76	70092	92	70108	108	70124	124
70013	13	70029	29	70045	45	70061	61	70077	77	70093	93	70109	109	70125	125
70014	14	70030	30	70046	46	70062	62	70078	78	70094	94	70110	110	70126	126
70015	15	70031	31	70047	47	70063	63	70079	79	70095	95	70111	111	70127	127

Hershey Maths Symbols

Software Font 71

71000	0	71016	16	71032	32	71048	48	71064	64	71080	80	71096	96	71112	112
71001	1	71017	17	71033	33	71049	49	71065	65	71081	81	71097	97	71113	113
71002	2	71018	18	71034	34	71050	50	71066	66	71082	82	71098	98	71114	114
71003	3	71019	19	71035	35	71051	51	71067	67	71083	83	71099	99	71115	115
71004	4	71020	20	71036	36	71052	52	71068	68	71084	84	71100	100	71116	116
71005	5	71021	21	71037	37	71053	53	71069	69	71085	85	71101	101	71117	117
71006	6	71022	22	71038	38	71054	54	71070	70	71086	86	71102	102	71118	118
71007	7	71023	23	71039	39	71055	55	71071	71	71087	87	71103	103	71119	119
71008	8	71024	24	71040	40	71056	56	71072	72	71088	88	71104	104	71120	120
71009	9	71025	25	71041	41	71057	57	71073	73	71089	89	71105	105	71121	121
71010	10	71026	26	71042	42	71058	58	71074	74	71090	90	71106	106	71122	122
71011	11	71027	27	71043	43	71059	59	71075	75	71091	91	71107	107	71123	123
71012	12	71028	28	71044	44	71060	60	71076	76	71092	92	71108	108	71124	124
71013	13	71029	29	71045	45	71061	61	71077	77	71093	93	71109	109	71125	125
71014	14	71030	30	71046	46	71062	62	71078	78	71094	94	71110	110	71126	126
71015	15	71031	31	71047	47	71063	63	71079	79	71095	95	71111	111	71127	127

Hershey Symbols 1 **Software Font 72**

72000	0	72016	16	72032	32	72048	48	72064	64	72080	80	72096	96	72112	112
72001	1	72017	17	72033	33	72049	49	72065	65	72081	81	72097	97	72113	113
72002	2	72018	18	72034	34	72050	50	72066	66	72082	82	72098	98	72114	114
72003	3	72019	19	72035	35	72051	51	72067	67	72083	83	72099	99	72115	115
72004	4	72020	20	72036	36	72052	52	72068	68	72084	84	72100	100	72116	116
72005	5	72021	21	72037	37	72053	53	72069	69	72085	85	72101	101	72117	117
72006	6	72022	22	72038	38	72054	54	72070	70	72086	86	72102	102	72118	118
72007	7	72023	23	72039	39	72055	55	72071	71	72087	87	72103	103	72119	119
72008	8	72024	24	72040	40	72056	56	72072	72	72088	88	72104	104	72120	120
72009	9	72025	25	72041	41	72057	57	72073	73	72089	89	72105	105	72121	121
72010	10	72026	26	72042	42	72058	58	72074	74	72090	90	72106	106	72122	122
72011	11	72027	27	72043	43	72059	59	72075	75	72091	91	72107	107	72123	123
72012	12	72028	28	72044	44	72060	60	72076	76	72092	92	72108	108	72124	124
72013	13	72029	29	72045	45	72061	61	72077	77	72093	93	72109	109	72125	125
72014	14	72030	30	72046	46	72062	62	72078	78	72094	94	72110	110	72126	126
72015	15	72031	31	72047	47	72063	63	72079	79	72095	95	72111	111	72127	127

Hershey Symbols 2 **Software Font 73**

73000	0	73016	16	73032	32	73048	48	73064	64	73080	80	73096	96	73112	112
73001	1	73017	17	73033	33	73049	49	73065	65	73081	81	73097	97	73113	113
73002	2	73018	18	73034	34	73050	50	73066	66	73082	82	73098	98	73114	114
73003	3	73019	19	73035	35	73051	51	73067	67	73083	83	73099	99	73115	115
73004	4	73020	20	73036	36	73052	52	73068	68	73084	84	73100	100	73116	116
73005	5	73021	21	73037	37	73053	53	73069	69	73085	85	73101	101	73117	117
73006	6	73022	22	73038	38	73054	54	73070	70	73086	86	73102	102	73118	118
73007	7	73023	23	73039	39	73055	55	73071	71	73087	87	73103	103	73119	119
73008	8	73024	24	73040	40	73056	56	73072	72	73088	88	73104	104	73120	120
73009	9	73025	25	73041	41	73057	57	73073	73	73089	89	73105	105	73121	121
73010	10	73026	26	73042	42	73058	58	73074	74	73090	90	73106	106	73122	122
73011	11	73027	27	73043	43	73059	59	73075	75	73091	91	73107	107	73123	123
73012	12	73028	28	73044	44	73060	60	73076	76	73092	92	73108	108	73124	124
73013	13	73029	29	73045	45	73061	61	73077	77	73093	93	73109	109	73125	125
73014	14	73030	30	73046	46	73062	62	73078	78	73094	94	73110	110	73126	126
73015	15	73031	31	73047	47	73063	63	73079	79	73095	95	73111	111	73127	127

Symbol type 1 - normal
Software Font 74
Symbol type 1 - thick
Software Font 75

Software Font 77 — Symbol type 2 - normal

0 77000	16 77016	32 77032	48 77048	64 77064	80 77080	96 77096	112 77112
1 77001	17 77017	33 77033	49 77049	65 77065	81 77081	97 77097	113 77113
2 77002	18 77018	34 77034	50 77050	66 77066	82 77082	98 77098	114 77114
3 77003	19 77019	35 77035	51 77051	67 77067	83 77083	99 77099	115 77115
4 77004	20 77020	36 77036	52 77052	68 77068	84 77084	100 77100	116 77116
5 77005	21 77021	37 77037	53 77053	69 77069	85 77085	101 77101	117 77117
6 77006	22 77022	38 77038	54 77054	70 77070	86 77086	102 77102	118 77118
7 77007	23 77023	39 77039	55 77055	71 77071	87 77087	103 77103	119 77119
8 77008	24 77024	40 77040	56 77056	72 77072	88 77088	104 77104	120 77120
9 77009	25 77025	41 77041	57 77057	73 77073	89 77089	105 77105	121 77121
10 77010	26 77026	42 77042	58 77058	74 77074	90 77090	106 77106	122 77122
11 77011	27 77027	43 77043	59 77059	75 77075	91 77091	107 77107	123 77123
12 77012	28 77028	44 77044	60 77060	76 77076	92 77092	108 77108	124 77124
13 77013	29 77029	45 77045	61 77061	77 77077	93 77093	109 77109	125 77125
14 77014	30 77030	46 77046	62 77062	78 77078	94 77094	110 77110	126 77126
15 77015	31 77031	47 77047	63 77063	79 77079	95 77095	111 77111	127 77127

Software Font 76 — Symbol type 1 - filled

0 76000	16 76016	32 76032	48 76048	64 76064	80 76080	96 76096	112 76112
1 76001	17 76017	33 76033	49 76049	65 76065	81 76081	97 76097	113 76113
2 76002	18 76018	34 76034	50 76050	66 76066	82 76082	98 76098	114 76114
3 76003	19 76019	35 76035	51 76051	67 76067	83 76083	99 76099	115 76115
4 76004	20 76020	36 76036	52 76052	68 76068	84 76084	100 76100	116 76116
5 76005	21 76021	37 76037	53 76053	69 76069	85 76085	101 76101	117 76117
6 76006	22 76022	38 76038	54 76054	70 76070	86 76086	102 76102	118 76118
7 76007	23 76023	39 76039	55 76055	71 76071	87 76087	103 76103	119 76119
8 76008	24 76024	40 76040	56 76056	72 76072	88 76088	104 76104	120 76120
9 76009	25 76025	41 76041	57 76057	73 76073	89 76089	105 76105	121 76121
10 76010	26 76026	42 76042	58 76058	74 76074	90 76090	106 76106	122 76122
11 76011	27 76027	43 76043	59 76059	75 76075	91 76091	107 76107	123 76123
12 76012	28 76028	44 76044	60 76060	76 76076	92 76092	108 76108	124 76124
13 76013	29 76029	45 76045	61 76061	77 76077	93 76093	109 76109	125 76125
14 76014	30 76030	46 76046	62 76062	78 76078	94 76094	110 76110	126 76126
15 76015	31 76031	47 76047	63 76063	79 76079	95 76095	111 76111	127 76127

Symbol type 2 - filled **Software Font 78** **GINO Dingbats** **Software Font 79**

Courier **Hardware Font 100** **Helvetica** **Software/Hardware Font 101**

Times **Software/Hardware Font 102**

102000 0	102016 16	102032 32	0 102048 48	@ 102064 64	P 102080 80	` 102096 96	p 102112 112
102001 1	102017 17	! 102033 33	1 102049 49	A 102065 65	Q 102081 81	a 102097 97	q 102113 113
102002 2	102018 18	" 102034 34	2 102050 50	B 102066 66	R 102082 82	b 102098 98	r 102114 114
102003 3	102019 19	# 102035 35	3 102051 51	C 102067 67	S 102083 83	c 102099 99	s 102115 115
102004 4	102020 20	$ 102036 36	4 102052 52	D 102068 68	T 102084 84	d 102100 100	t 102116 116
102005 5	102021 21	% 102037 37	5 102053 53	E 102069 69	U 102085 85	e 102101 101	u 102117 117
102006 6	102022 22	& 102038 38	6 102054 54	F 102070 70	V 102086 86	f 102102 102	v 102118 118
102007 7	102023 23	' 102039 39	7 102055 55	G 102071 71	W 102087 87	g 102103 103	w 102119 119
102008 8	102024 24	(102040 40	8 102056 56	H 102072 72	X 102088 88	h 102104 104	x 102120 120
102009 9	102025 25	) 102041 41	9 102057 57	I 102073 73	Y 102089 89	i 102105 105	y 102121 121
102010 10	102026 26	* 102042 42	: 102058 58	J 102074 74	Z 102090 90	j 102106 106	z 102122 122
102011 11	102027 27	+ 102043 43	; 102059 59	K 102075 75	[102091 91	k 102107 107	{ 102123 123
102012 12	102028 28	, 102044 44	< 102060 60	L 102076 76	\ 102092 92	l 102108 108	| 102124 124
102013 13	102029 29	- 102045 45	= 102061 61	M 102077 77	] 102093 93	m 102109 109	} 102125 125
102014 14	102030 30	. 102046 46	> 102062 62	N 102078 78	^ 102094 94	n 102110 110	102126 126
102015 15	102031 31	/ 102047 47	? 102063 63	O 102079 79	_ 102095 95	o 102111 111	^ 102127 127

AvantGarde **Software/Hardware Font 103**

103000 0	103016 16	103032 32	0 103048 48	@ 103064 64	P 103080 80	` 103096 96	p 103112 112
103001 1	103017 17	! 103033 33	1 103049 49	A 103065 65	Q 103081 81	a 103097 97	q 103113 113
103002 2	103018 18	" 103034 34	2 103050 50	B 103066 66	R 103082 82	b 103098 98	r 103114 114
103003 3	103019 19	# 103035 35	3 103051 51	C 103067 67	S 103083 83	c 103099 99	s 103115 115
103004 4	103020 20	$ 103036 36	4 103052 52	D 103068 68	T 103084 84	d 103100 100	t 103116 116
103005 5	103021 21	% 103037 37	5 103053 53	E 103069 69	U 103085 85	e 103101 101	u 103117 117
103006 6	103022 22	& 103038 38	6 103054 54	F 103070 70	V 103086 86	f 103102 102	v 103118 118
103007 7	103023 23	' 103039 39	7 103055 55	G 103071 71	W 103087 87	g 103103 103	w 103119 119
103008 8	103024 24	(103040 40	8 103056 56	H 103072 72	X 103088 88	h 103104 104	x 103120 120
103009 9	103025 25	) 103041 41	9 103057 57	I 103073 73	Y 103089 89	i 103105 105	y 103121 121
103010 10	103026 26	* 103042 42	: 103058 58	J 103074 74	Z 103090 90	j 103106 106	z 103122 122
103011 11	103027 27	+ 103043 43	; 103059 59	K 103075 75	[103091 91	k 103107 107	{ 103123 123
103012 12	103028 28	, 103044 44	< 103060 60	L 103076 76	\ 103092 92	l 103108 108	| 103124 124
103013 13	103029 29	- 103045 45	= 103061 61	M 103077 77	] 103093 93	m 103109 109	} 103125 125
103014 14	103030 30	. 103046 46	> 103062 62	N 103078 78	^ 103094 94	n 103110 110	103126 126
103015 15	103031 31	/ 103047 47	? 103063 63	O 103079 79	_ 103095 95	o 103111 111	^ 103127 127

LublinGraph **Software/Hardware Font 104**

104000 0	104016 16	104032 32	0 104048 48	@ 104064 64	P 104080 80	` 104096 96	p 104112 112
104001 1	104017 17	! 104033 33	1 104049 49	A 104065 65	Q 104081 81	a 104097 97	q 104113 113
104002 2	104018 18	" 104034 34	2 104050 50	B 104066 66	R 104082 82	b 104098 98	r 104114 114
104003 3	104019 19	# 104035 35	3 104051 51	C 104067 67	S 104083 83	c 104099 99	s 104115 115
104004 4	104020 20	$ 104036 36	4 104052 52	D 104068 68	T 104084 84	d 104100 100	t 104116 116
104005 5	104021 21	% 104037 37	5 104053 53	E 104069 69	U 104085 85	e 104101 101	u 104117 117
104006 6	104022 22	& 104038 38	6 104054 54	F 104070 70	V 104086 86	f 104102 102	v 104118 118
104007 7	104023 23	' 104039 39	7 104055 55	G 104071 71	W 104087 87	g 104103 103	w 104119 119
104008 8	104024 24	(104040 40	8 104056 56	H 104072 72	X 104088 88	h 104104 104	x 104120 120
104009 9	104025 25	) 104041 41	9 104057 57	I 104073 73	Y 104089 89	i 104105 105	y 104121 121
104010 10	104026 26	* 104042 42	: 104058 58	J 104074 74	Z 104090 90	j 104106 106	z 104122 122
104011 11	104027 27	+ 104043 43	; 104059 59	K 104075 75	(104091 91	k 104107 107	{ 104123 123
104012 12	104028 28	, 104044 44	< 104060 60	L 104076 76	\ 104092 92	l 104108 108	| 104124 124
104013 13	104029 29	- 104045 45	= 104061 61	M 104077 77	) 104093 93	m 104109 109	} 104125 125
104014 14	104030 30	. 104046 46	> 104062 62	N 104078 78	^ 104094 94	n 104110 110	104126 126
104015 15	104031 31	/ 104047 47	? 104063 63	O 104079 79	_ 104095 95	o 104111 111	^ 104127 127

NewCentury **Software/Hardware Font 105**

105000 0	105016 16	105032 32	0 105048 48	@ 105064 64	P 105080 80	` 105096 96	p 105112 112
105001 1	105017 17	! 105033 33	1 105049 49	A 105065 65	Q 105081 81	a 105097 97	q 105113 113
105002 2	105018 18	" 105034 34	2 105050 50	B 105066 66	R 105082 82	b 105098 98	r 105114 114
105003 3	105019 19	# 105035 35	3 105051 51	C 105067 67	S 105083 83	c 105099 99	s 105115 115
105004 4	105020 20	$ 105036 36	4 105052 52	D 105068 68	T 105084 84	d 105100 100	t 105116 116
105005 5	105021 21	% 105037 37	5 105053 53	E 105069 69	U 105085 85	e 105101 101	u 105117 117
105006 6	105022 22	& 105038 38	6 105054 54	F 105070 70	V 105086 86	f 105102 102	v 105118 118
105007 7	105023 23	' 105039 39	7 105055 55	G 105071 71	W 105087 87	g 105103 103	w 105119 119
105008 8	105024 24	(105040 40	8 105056 56	H 105072 72	X 105088 88	h 105104 104	x 105120 120
105009 9	105025 25	) 105041 41	9 105057 57	I 105073 73	Y 105089 89	i 105105 105	y 105121 121
105010 10	105026 26	* 105042 42	: 105058 58	J 105074 74	Z 105090 90	j 105106 106	z 105122 122
105011 11	105027 27	+ 105043 43	; 105059 59	K 105075 75	[105091 91	k 105107 107	{ 105123 123
105012 12	105028 28	, 105044 44	< 105060 60	L 105076 76	\ 105092 92	l 105108 108	| 105124 124
105013 13	105029 29	- 105045 45	= 105061 61	M 105077 77	] 105093 93	m 105109 109	} 105125 125
105014 14	105030 30	. 105046 46	> 105062 62	N 105078 78	^ 105094 94	n 105110 110	105126 126
105015 15	105031 31	/ 105047 47	? 105063 63	O 105079 79	_ 105095 95	o 105111 111	i 105127 127

Souvenir **Software/Hardware Font 106**

106000 0	106016 16	106032 32	0 106048 48	@ 106064 64	P 106080 80	` 106096 96	p 106112 112
106001 1	106017 17	! 106033 33	1 106049 49	A 106065 65	Q 106081 81	a 106097 97	q 106113 113
106002 2	106018 18	" 106034 34	2 106050 50	B 106066 66	R 106082 82	b 106098 98	r 106114 114
106003 3	106019 19	# 106035 35	3 106051 51	C 106067 67	S 106083 83	c 106099 99	s 106115 115
106004 4	106020 20	$ 106036 36	4 106052 52	D 106068 68	T 106084 84	d 106100 100	t 106116 116
106005 5	106021 21	% 106037 37	5 106053 53	E 106069 69	U 106085 85	e 106101 101	u 106117 117
106006 6	106022 22	& 106038 38	6 106054 54	F 106070 70	V 106086 86	f 106102 102	v 106118 118
106007 7	106023 23	' 106039 39	7 106055 55	G 106071 71	W 106087 87	g 106103 103	w 106119 119
106008 8	106024 24	(106040 40	8 106056 56	H 106072 72	X 106088 88	h 106104 104	x 106120 120
106009 9	106025 25	) 106041 41	9 106057 57	I 106073 73	Y 106089 89	i 106105 105	y 106121 121
106010 10	106026 26	* 106042 42	: 106058 58	J 106074 74	Z 106090 90	j 106106 106	z 106122 122
106011 11	106027 27	+ 106043 43	; 106059 59	K 106075 75	[106091 91	k 106107 107	{ 106123 123
106012 12	106028 28	, 106044 44	< 106060 60	L 106076 76	\ 106092 92	l 106108 108	| 106124 124
106013 13	106029 29	- 106045 45	= 106061 61	M 106077 77	] 106093 93	m 106109 109	} 106125 125
106014 14	106030 30	. 106046 46	> 106062 62	N 106078 78	^ 106094 94	n 106110 110	106126 126
106015 15	106031 31	/ 106047 47	? 106063 63	O 106079 79	_ 106095 95	o 106111 111	^ 106127 127

Palatino **Software/Hardware Font 107**

107000 0	107016 16	107032 32	0 107048 48	@ 107064 64	P 107080 80	` 107096 96	p 107112 112
107001 1	107017 17	! 107033 33	1 107049 49	A 107065 65	Q 107081 81	a 107097 97	q 107113 113
107002 2	107018 18	" 107034 34	2 107050 50	B 107066 66	R 107082 82	b 107098 98	r 107114 114
107003 3	107019 19	# 107035 35	3 107051 51	C 107067 67	S 107083 83	c 107099 99	s 107115 115
107004 4	107020 20	$ 107036 36	4 107052 52	D 107068 68	T 107084 84	d 107100 100	t 107116 116
107005 5	107021 21	% 107037 37	5 107053 53	E 107069 69	U 107085 85	e 107101 101	u 107117 117
107006 6	107022 22	& 107038 38	6 107054 54	F 107070 70	V 107086 86	f 107102 102	v 107118 118
107007 7	107023 23	' 107039 39	7 107055 55	G 107071 71	W 107087 87	g 107103 103	w 107119 119
107008 8	107024 24	(107040 40	8 107056 56	H 107072 72	X 107088 88	h 107104 104	x 107120 120
107009 9	107025 25	) 107041 41	9 107057 57	I 107073 73	Y 107089 89	i 107105 105	y 107121 121
107010 10	107026 26	* 107042 42	: 107058 58	J 107074 74	Z 107090 90	j 107106 106	z 107122 122
107011 11	107027 27	+ 107043 43	; 107059 59	K 107075 75	[107091 91	k 107107 107	{ 107123 123
107012 12	107028 28	, 107044 44	< 107060 60	L 107076 76	\ 107092 92	l 107108 108	| 107124 124
107013 13	107029 29	- 107045 45	= 107061 61	M 107077 77	] 107093 93	m 107109 109	} 107125 125
107014 14	107030 30	. 107046 46	> 107062 62	N 107078 78	^ 107094 94	n 107110 110	107126 126
107015 15	107031 31	/ 107047 47	? 107063 63	O 107079 79	_ 107095 95	o 107111 111	^ 107127 127

Chancery **Software/Hardware Font 108**

108000 0	108016 16	108032 32	0 108048 48	@ 108064 64	P 108080 80	` 108096 96	p 108112 112
108001 1	108017 17	! 108033 33	1 108049 49	A 108065 65	Q 108081 81	a 108097 97	q 108113 113
108002 2	108018 18	" 108034 34	2 108050 50	B 108066 66	R 108082 82	b 108098 98	r 108114 114
108003 3	108019 19	# 108035 35	3 108051 51	C 108067 67	S 108083 83	c 108099 99	s 108115 115
108004 4	108020 20	$ 108036 36	4 108052 52	D 108068 68	T 108084 84	d 108100 100	t 108116 116
108005 5	108021 21	% 108037 37	5 108053 53	E 108069 69	U 108085 85	e 108101 101	u 108117 117
108006 6	108022 22	& 108038 38	6 108054 54	F 108070 70	V 108086 86	f 108102 102	v 108118 118
108007 7	108023 23	' 108039 39	7 108055 55	G 108071 71	W 108087 87	g 108103 103	w 108119 119
108008 8	108024 24	(108040 40	8 108056 56	H 108072 72	X 108088 88	h 108104 104	x 108120 120
108009 9	108025 25	) 108041 41	9 108057 57	I 108073 73	Y 108089 89	i 108105 105	y 108121 121
108010 10	108026 26	* 108042 42	: 108058 58	J 108074 74	Z 108090 90	j 108106 106	z 108122 122
108011 11	108027 27	+ 108043 43	; 108059 59	K 108075 75	[108091 91	k 108107 107	{ 108123 123
108012 12	108028 28	, 108044 44	< 108060 60	L 108076 76	\ 108092 92	l 108108 108	| 108124 124
108013 13	108029 29	- 108045 45	= 108061 61	M 108077 77	] 108093 93	m 108109 109	} 108125 125
108014 14	108030 30	. 108046 46	> 108062 62	N 108078 78	^ 108094 94	n 108110 110	108126 126
108015 15	108031 31	/ 108047 47	? 108063 63	O 108079 79	_ 108095 95	o 108111 111	^ 108127 127

Appendix D

DEFAULTS

Defaults Introduction

These are GINO front-end supplied defaults. They are independent of the device driver. Defaults that may be overwritten by the device driver are marked [DD], meaning Device Dependent. The default values set at the GINO initialization are marked [G]; the values set on calling gNewDrawing() are marked [P]; the remaining, unmarked values are initialized at the device nomination.

GINO elements	Default Values	Corresponding GINO function calls
ERROR		
maximum	10 [G]	gSetMaxErrorLimit(10)
count	0 [G]	
message switch	on [G]	gSetErrorMode(GALLON)
error trapping	off [G]	gSetErrorTrap(GOFF)
TRACE SWITCH	off [G]	gSetTracerMode(GOFF)
WORKSPACE AREA	none [G]	gSetWorkspaceLimit(0,0)
DEVICE		
driver	DUMMY	
type	0	gSetDeviceFilename(,0) [DD]
DRAWING UNITS	millimetres	gDefinePictureUnits(1.0)
PAPER		

GINO elements	Default Values	Corresponding GINO function calls
size	200.0 x 200.0 mm	gSetDrawingLimits(200.0,200.0,0) [DD]
type	0	
POSITION		
space		gMoveTo3D(0.0,0.0,0.0)
picture	[P]	gMoveTo3D(0.0,0.0,0.0)
LINE ATTRIBUTES		
visibility	on	gSetLineVis(GVISIBLE)
broken line type	solid	gSetBrokenLine(GSOLID)
colour index	black	gSetLineColour(GBLACK) [DD]
line width	0.2 mm	gSetLineWidth(0.2) [DD]
line width scaling	1.0	gSetLineWidthScaling (1.0)
pen type	undefined	gSetPenType(GDEFAULT) [DD]
line end type	no ends	gSetLineEnd(GNONE)
line attributes table	each entry set to the current line attributes, except colour index is set equal to line style index (col=line)	GLINSTY rep={GVISIBLE, GSOLID, col, 0.2, GDEFAULT, GNONE}; gDefineLineStyle(line,&rep)
COLOUR TABLE		
0 (GBACKGROUND)	Background colour	[DD]
1 (GBLACK)	Black	
2 (GRED)	Red	
3 (GORANGE)	Orange	
4 (GYELLOW)	Yellow	
5 (GGREEN)	Green	
6 (GCYAN)	Cyan	
7 (GBLUE)	Blue	
8 (GMAGENTA)	Magenta	
9 (GBROWN)	Brown	
10 (GWHITE)	White	
Device capability		[DD]

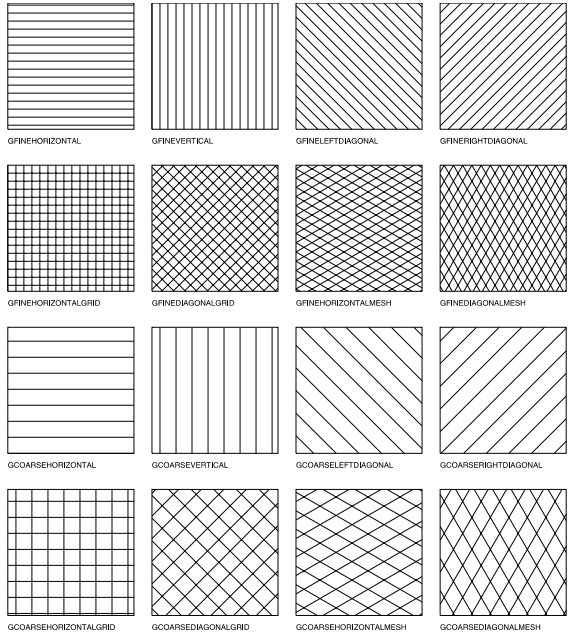
GINO elements	Default Values	Corresponding GINO function calls
CHARACTER		
mode		gSetMixedChars()
switch	untransformable	gSetCharTransformMode() (0)
font	0	gSetCharFont(GDEFAULT)
width and height	3.0 x 3.0 mm	gSetCharSize(3.0,3.0) [DD]
angle	0.0^{o}	gSetStrAngle(0.0)
slant	0.0^{o}	gSetItalicAngle(0.0)
underlining	off	gSetStrUnderscore(GOFF)
font fill style	solid	gSetFontFillStyle(1,0,0,-1,0)
font weight	normal	gSetFontWeight(0)
font spacing	as defined	gSetFontSpacing(0)
font representation	as requested	gSetFontForm(0)
escape character	an asterisk '*'	gSetEscapeChar('*')
justification	left	gSetStrJustify(GLEFT)
exponent/index size	0.6 character size	gSetStrExponent(0.6,0.6,*,*)
exponent position	0.6 above base line	gSetStrExponent(*,*,0.6,*)
index position	0.3 below base line	gSetStrExponent(*,*,*,0.3)
BROKEN LINE TABLE		gDefineBrokenLineStyle()
1 (GSHORTDASHED)	short dashed	1 { 1, 6.0, 4.0, 0.0}
2 (GSHORTDOTTED)	short dotted	2 { 1, 4.0, 0.8, 0.0}
3 (GSHORTCHAINED)	short chained	3 { 2, 10.0, 6.0, 1.0}
4 (GLONGDASHED)	long dashed	4 { 1, 12.0, 8.0, 0.0}
5 (GLONGDOTTED)	long dotted	5 { 1, 8.0, 1.6, 0.0}
6 (GLONGCHAINED)	long chained	6 { 2, 20.0,12.0, 2.0}
7 (GDOTTED)	dotted	7 { 1, 1.5, 0.5, 0.0}
8	dotted	8 { 1, 2.0, 1.0, 0.0}
9	dashed	9 { 1, 3.0, 2.0, 0.0}
10	dashed	10 { 1, 6.0, 5.0, 0.0}
11	dashed	11 { 1,10.0, 8.0, 0.0}
12	dashed	12 { 1,15.0,12.0, 0.0}
13	chained	13 { 2, 6.0, 3.0, 0.5}
14	chained	14 { 2, 8.0, 5.0, 0.5}
15	chained	15 { 2 12.0, 8.0, 0.5}

GINO elements	Default Values	Corresponding GINO function calls
16	chained	16 { 2 16.0 12.0, 1.0}
broken line switch	hardware	gSetBrokenLineMode(GHARD)
ARCS		
curve tolerance	0.1 mm	gSetArcTolerance(0.1) [DD]
arc switch	hardware	gSetArcMode(GHARD)
AREA FILL		
switch	hardware	gSetFillMode(GHARD)
HATCH TABLE		gDefineHatchStyle()
1 (GFINEHORIZONTAL)	fine horizontal hatch	1 { 2.0, 0.0,0.0,0.0, 0.0,0}
2 (GFINEVERTICAL)	fine vertical hatch	2 { 2.0, 90.0,0.0,0.0, 0.0,0}
3 (GFINELEFTDIAGONAL)	fine left diagonal hatch	3 { 2.0,-45.0,0.0,0.0,-45.0,0}
4 (GFINERIGHTDIAGONAL)	fine right diagonal hatch	4 { 2.0, 45.0,0.0,0.0, 45.0,0}
5 (GFINEHORIZONTALGRID)	fine horizontal hatch	5 { 2.0, 0.0,0.0,0.0, 0.0,1}
6 (GFINEDIAGONALGRID)	fine vertical hatch	6 { 2.0, 45.0,0.0,0.0, 0.0,1}
7 (GFINEHORIZONTALMESH)	fine left diagonal hatch	7 { 2.0,-30.0,0.0,0.0, 30.0,1}
8 (GFINEDIAGONALMESH)	fine right diagonal hatch	8 { 2.0, 60.0,0.0,0.0, 30.0,1}
9 (GCOARSEHORIZONTAL)	coarse horizontal hatch	9 { 4.0, 0.0,0.0,0.0, 0.0,0}
10 (GCOARSEVERTICAL)	coarse vertical hatch	10 { 4.0, 90.0,0.0,0.0, 0.0,0}
11 (GCOARSELEFTDIAGONAL)	coarse left diagonal hatch	11 { 4.0,-45.0,0.0,0.0,-45.0,0}
12 (GCOARSERIGHTDIAGONAL)	coarse right diagonal hatch	12 { 4.0, 45.0,0.0,0.0, 45.0,0}
13 (GCOARSEHORIZONTALGRID)	coarse horizontal grid	13 { 4.0, 0.0,0.0,0.0, 0.0,1}
14 (GCOARSEDIAGONALGRID)	coarse diagonal grid	14 { 4.0, 45.0,0.0,0.0, 0.0,1}
15 (GCOARSEHORIZONTALMESH)	coarse horizontal mesh	15 { 4.0,-30.0,0.0,0.0,-30.0,1}
16 (GCOARSEDIAGONALMESH)	coarse vertical mesh	16 { 4.0, 60.0,0.0,0.0, 30.0,1}
VIEW/TRANSFORM MODE		

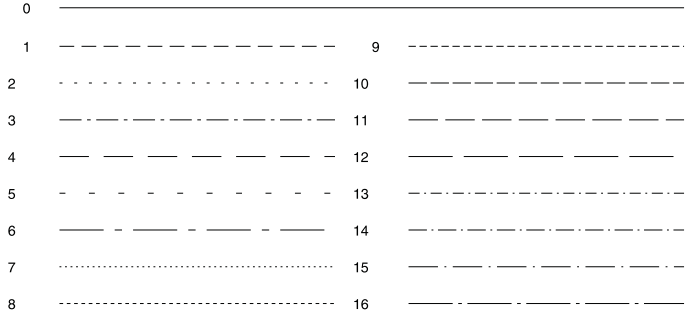
GINO elements	Default Values	Corresponding GINO function calls
mode	hardware/software	gSetViewTransformMode(DD)
VIEWING		
type	parallel view	
view centre	(0.0,0.0,0.0)	
view direction	(0.0,0.0,-1.0)	
position of view centre	centred on window or device limits	gInitView()
up direction	positive y-axis, or failing that, negative z-axis	
TRANSFORMATION		
switch	off	gSetTransform(GOFF)
matrix	unit matrix	
mode	space mode	gSetTransformMode(GSPACE)
saved state	unit matrix switched off, stack empty	
VIEWPORT		
limits	windowing limits	
transformation mode	maintain aspect ratio and centralize	gSetViewportMode(GCENTRAL)
WINDOWING		
switch	off [G]	gSetWindowMode(GOFF)
windowing limits	maximum and minimum possible floating point values. Normally overwritten by the device driver	
MASKING		
switch	off [G]	gSetMaskMode(GOFF)
limits	unset [G]	gSetMask2D({0.0,0.0,0.0,0.0})
POLYGON		

GINO elements	Default Values	Corresponding GINO function calls
workspace	none [G]	gDefinePolygonWorkspace(0)
polygons/vertices	none [G]	gClearPolygonWorkspace()
polygon identifier	0 [G]	gSetPolygonIdent(0)
current polygon list	no list [G]	gSelectPolygons(&list,0)
DISPLAY FILE		
switch	hardware [G]	gSetSegMode(GHARDWARE)
SDFBUF area	none [G]	gDefineSegWorkspace(0)
SDF marking colour	NDC	gSetSegMarkColour(ndc)
PICTURE SEGMENT		
default	0	gExtendSeg(0)
groups	group range	gDefineGroupRange(1,32767)
table	no groups defined	gRemoveSegGroup(-1)
visibility	on	gSetSegVis(*,GVISIBLE)
sensitivity	off	gSetSegHit(*,GNONSENSITIVE)
marked	off	gMarkSeg(*,GUNMARK)
CURSOR		
start position	centre of device limits [P]	
type	default	gSetCursorType(GDEFAULT)
EVENTS		
all disabled		gRemoveEventType(GALL)
LIGHTING AND SHADING		gSetShadingMode()
Shading Mode	off	GOFF,*,*,*
Culling	off	*,GOFF,*,*
Blending	off	*,*,GOFF,*
Winding	anticlockwise	*,*,*,GANTICLOCKWISE
LIGHT TABLE		
1	white ambient	gDefineLight(1,GWHITE)

GINO elements	Default Values	Corresponding GINO function calls
2	white directional at 0.0,0.0,ZMAX	
LIGHT SWITCHES		
1-8	off	gSetLightSwitch(*,GOFF)
MATERIAL TABLE		gDefineMaterial()
1	normal	1 {0.3,0.6,0.0,30.0,1.0}
2	plastic	2 {0.3,0.6,1.0,30.0,1.0}
3	shiny	3 {0.3,0.6,1.0,100.0,1.0}
MATERIAL INDEX		
setting	match GINO colours	gSetMaterialIndex(GOFF,GOFF)
FACET ATTRIBUTES		
filling	solid	gSetFacetFillStyle(GSOLID)
offset	off	gSetFacetOffsetMode(GOFF)
TEXTURE MAPPING		
mode	off	gSetTextureMappingMode(GOFF)
blend colour	background	
wrapping	repeat	
enlargement/reduction method	nearest neighbour	
border colour	background	
TEXTURE GENERATION		
mode	off	
FOG ATTRIBUTES		gDefineFog()
mode	off	GNONE
colour	black	
density	0.0025	
linear start/end values	0.0,1.0	



Hatch/Fill Styles



Line Styles

Appendix E

ERROR AND WARNING MESSAGES

Error and Warning Introduction

This Appendix is divided into four sections:

1. Errors and warnings that can be generated from calling GINO routines

2. CGM specific errors

3. System I/O errors generated from problems with file input or output

4. Configuration File errors

GINO Errors and Warnings

1 **Call to GINO routine requiring device to be nominated**
A GINO program must call a device nomination routine (from Appendix B) to generate graphics

2 **Cannot open file for device output**
A file open error has occurred opening the requested metafile - check access rights

3 **Attempt to close segment when one not open**
A call to gCloseSeg() made with no corresponding gOpenSeg()

4 **Error count exceeds maximum permitted**
More than 10 errors generated - see gSetMaxErrorLimit()

5 **GINO not initialized**
Graphics device failed to initialize - check environment or contact software support

6 **Routine used that will not be available in next release**
Check details in Appendix H

7 **Routine no longer available**
Check details in Appendix H

8 **Device not available**
Error transmitting data to device - check links or network

9 **Call to qualifying routine after device initialization**
Move offending call to immediately following nomination routine

10 **Cursor not available**
Self explanatory

12 **Perspective present in current transformation**
Need to use homogeneous coordinates

13 **Singular input transformation matrix**
Self explanatory

14 **3D terms present in current transformation**
Use 3D transformation routine

15 **Point lies behind eye point and does not transform**
Self explanatory

16 **Polygon vertex not recorded (polygon workspace full)**
Increase polygon workspace using gDefinePolygonWorkspace() or reduce complexity

18 **GINO font data file not available**
Check environment and installation notes

19 **Invalid workspace limits**
N1 > N2 - correct values

20 **Requested paper size too large**
Size larger than maximum on current device - check gEnqMaxDrawingLimits()

21 **Attempt to draw beyond lower X limit**
Check coordinates or switch on windowing

22 **Attempt to draw beyond upper X limit**
Check coordinates or switch on windowing

23 **Attempt to draw beyond lower Y limit**
Check coordinates or switch on windowing

24 **Attempt to draw beyond upper Y limit**
Check coordinates or switch on windowing

25 **Attempt to draw beyond lower Z limit**
Check coordinates or switch on windowing

26 **Attempt to draw beyond upper Z limit**
Check coordinates or switch on windowing

27 **Attempt to position segment anchor outside device limits**
First visible point of segment must lie within device limits

28 **Invalid segment operation on current segment**
Attempt to use gMoveSegBy2D() on current segment

29 **Negative workspace size requested**
Self explanatory

30 **Not enough space available for new workspace size**
Increase global workspace size with gSetWorkspaceLimit()

31 **New workspace size too small for existing data**
Can’t decrease workspace to less than current data requirements

33 **Invalid character follows escape character**
Probable single use of current escape character

34 **Negative polygon list size**
Self explanatory

35 **Character not available on output device**
Self explanatory

36 **Negative polygon identifier**
Self explanatory

37 **Attempt to define invalid escape character**
Outside range 32 - 127 or defined escape code - see gDisplayStr()

38 **Not enough room in workspace**
Increase global workspace using gSetWorkspaceLimit()

39 **Routine requires workspace to be defined**
Must declare global workspace using gSetWorkspaceLimit() first

40 **Arc rotation axis is undefined**
Zero length direction vector in 3D arc

41 **Arc end vector is undefined**
Centre and end point are coincident

42 **Negative number of points in multi-drawing routine**
Self explanatory

43 **Curve is undefined (all points coincide)**
Self explanatory

44 **Single non-zero vector incr. with inconsistent end conditions**
Curve only contains two points and without extra end points for full definition

45 **Negative broken line type**
Self explanatory

46 **Negative colour index**
Self explanatory

47 **Negative line width**
Self explanatory

48 **Negative pen type**
Self explanatory

49 **Negative line end type**
Self explanatory

50 **Negative gap length**
Self explanatory

51 **Broken line mode out of range**
Mode must be one of GCONTCHAIN, GCONTDASH, GDISCONTDASH or GDISCONTCHAIN

52 **Broken line type out of range**
Outside range 1 to 256

56 **Line style index out of range**
Outside range 1 to 256

57 **Fill style index out of range**
Outside range 1 to 256

58 **Negative hatch line separation**
Self explanatory

59 **Invalid component in colour definition**
Outside range 0.0 to 1.0

60 **Transformation stack empty**
Attempt to use gPopTransform() without preceding gPushTransform()

61 **Transformation stack full**
Attempt to stack more than 10 transformation matrices

62 **Rotation axis out of range**
Axis must be one of GXAXIS, GYAXIS or GZAXIS

63 **Shear axis out of range**
Shear axis must be one of GXAXIS, GYAXIS or GZAXIS

64 **3-D Shear direction out of range**
Shear direction must be one of GXAXIS, GYAXIS or GZAXIS

65 **Zero scale factor**
Self explanatory

66 **Axis out of range**
View axes must be one of GXAXIS, GYAXIS or GZAXIS

67 **Transformation switch out of range**
Transform switch must be one of GRESET, GOFF, GON or GINIT

68 **Negative or zero character dimension**
Self explanatory

69 **Negative symbol number**
Self explanatory

70 **Negative repeat, dash or dot length in broken line def.**
Self explanatory

79 **No hardware segment facilities - SDF initialised**
An attempt has been made to open a picture segment on a device that does not handle segments. Use gSetSegMode() to define the required segment mode for the application

80 **Invalid segment number**
Attempt to define/use segment outside permitted range

81 **Segment does not exist**
Attempt to use segment not yet defined

82 **Invalid Event type**
Event type must be of permitted type

85 **Invalid ASCII code**
Character code outside range 0-127

88 **Invalid fill style in fill routine**
Attempt to use negative or invalid fill style

89 **Negative line style in fill routine**
Attempt to use negative line style

92 **Not enough space for temporary area fill workspace**
Increase global workspace using gSetWorkspaceLimit()

93 **Segment group number out of range**
Outside range 1-50

94 **Invalid segment range for segment group**
Start > end or either out or range

95 **Invalid range for segment group numbers**
Min > Max

96 **Segment group table full**
Reduce number of segment groups

97 **Negative dot radius**
Self explanatory

98 **Negative character count**
Self explanatory

99 **Numeric field width > 32**
Self explanatory

100 **Too many points in Bezier curve - truncated**
A limits of 200 points is imposed on any Bezier curve routine

102 **Attempt to switch masking on with no limits set**
Define mask using gSetMask2D() before switching on

103 **Failed to open file for metafile output**
File open error - check disk space or access

104 **Failed to open file for metafile input**
File open error - check file name and/or access rights

105 **Cannot read SAVPIC file**
Use gGetPicture() to read SAVPIC file

106 **Failed to rewind file for metafile input**
System cannot reset pointer to start of file - I/O error

107 **Metafile error in segment header data**
If file not corrupted, contact software support

108 **Metafile error in segment data**
If file not corrupted, contact software support

109 **Metafile error between segments**
If file not corrupted, contact software support

110 **Incorrect metafile header**
If file not corrupted, contact software support

111 **Picture segment not found in metafile**
Check list of segments in file using gEnqSavdraSegList()

112 **Incorrect metafile data**
If file not corrupted, contact software support

113 **Corrupt SAVDRA code**
If file not corrupted, contact software support

114 **Invalid SAVDRA metafile**
If file not corrupted, contact software support

116 **Invalid delimiter in metafile**
If file not corrupted, contact software support

117 **Incorrect character count for metafile input record**
If file not corrupted, contact software support

118 **Incorrect metafile segment header**
If file not corrupted, contact software support

119 **Invalid file unit**
File unit outside implementation range - use gFopen()

120 **Window/Mask switch out of range**
Switch must be either GON or GOFF

121 **Zero view UP vector**
Self explanatory

122 **Zero rotation axis vector**
Self explanatory

123 **Zero perspective distance**
Self explanatory

124 **Plane out of range**
Plane must be one of GYZPLANE, GXZPLANE or GXYPLANE

125 **Zero perspective distance**
Self explanatory

126 **Zero window dimension**
Self explanatory

127 **Zero line of sight vector in view**
Self explanatory

128 **Zero or negative sphere radius**
Self explanatory

129 **Line of sight vector and view UP vector coincide**
Self explanatory

130 **View with perspective required**
Self explanatory

131 **Window/Mask limits are outside device limits**
Self explanatory

133 **Invalid file unit**
File unit outside implementation range - use gFopen()

134 **Not enough space for clipped polygon data**
Increase global workspace using gSetWorkspaceLimit()

135 **Device driver error - incorrect access to polygon data**
Contact device driver writer or customer support

136 **Not enough space for metafile polygon data**
Increase global workspace using gSetWorkspaceLimit()

137 **Negative hit radius**
Self explanatory

138 **Negative character size index**
Self explanatory

139 **Input coordinates and current transformation are inconsistent**
Self explanatory

140 **Invalid maximum error limit**
Must set positive value

141 **Negative array size**
Self explanatory

142 **Attempt to position cursor outside device limits**
Self explanatory

143 **Segment identifier does not exist**
Self explanatory

144 **Negative or zero scale factor**
Self explanatory

145 **Negative number of decimal places in numeric field**
Self explanatory

146 **Invalid file unit**
Metafile file unit outside implementation range - check config file & Appendix B

148 **Workspace area is internally defined**
Global workspace size is predefined in library - this cannot be changed

150 **Current picture position is undefined**
Define specific pen position using gMove() routine before displaying string

151 **Too many vertices in polygon boundary**
Program has exceeded 2048 limit

152 **Request for memory buffer when already using file storage**
Must use gDefineSegWorkspace() before any segments are opened

153 **Segment Reference depth > 10**
Self explanatory

154 **Cannot open scratch software display file**
Internal file open error - check access rights

155 **Segment display file switch out of range**
Switch must be one of GSOFTWARE, GMIXWARE or GHARDWARE

156 **Call to change SDF mode while in segment**
Must close segment before using gSetSegMode()

157 **SDF File pointer corrupted**
Contact software support

158 **No polygon workspace**
Must call gDefinePolygonWorkspace() to use this routine

159 **Segment cannot reference itself**
Self explanatory

160 **Negative dialogue area origin**
Increase global workspace using gSetWorkspaceLimit()

161 **Negative number of rows or columns in dialogue area**
Self explanatory

165 **No room for polygonal window/mask**
Increase global workspace using gSetWorkspaceLimit()

166 **Invalid character justification**
Justification must be one of GLEFT, GCENTRE or GRIGHT

167 **Invalid argument**
Null type must be one of GNOSLASH, GSLASH or GTICK

168 **Negative font style**
Must use GOUTLINE, GFILLED, GOUTFIL or system dependent +'ve setting

169 **Invalid fitting switch**
Must use one of GB2P or GSIZE as fitting switch

170 **Invalid underscore switch**
Must use one of GON or GOFF

171 **Negative argument**
Must use positive value for all real arguments

172 **Not enough points for spline curve**
Must provide at least 3 points to define spline curve

173 **Invalid font number**
Font number is negative or unknown value

174 **No room in SDF memory buffer**
Increase size of segment workspace using gDefineSegWorkspace()

175 **Unable to open SDF archive file**
File open error - check access rights

176 **SDF mode not set**
Must call gSetSegMode() before retrieving segment archive

177 **SDF memory buffer not large enough for archive file**
Must increase segment workspace or use file mode

178 **New line requested outside text block**
The *N escape sequence must be used in conjunction with gStartTextBlock()

179 **Invalid font representation**
Must be in range 0-7

180 **Illegal cursor type**
Must be one of predefined cursor types

181 **Invalid number of points in cursor polyline**
Number of vertices must be positive

182 **Negative number of polygons in polygon set**
Self explanatory

183 **Invalid viewport limits**
Limits have zero range or outside device limits

184 **Invalid viewport scaling switch**
Mode must be one of GCENTRAL, GBOTTOMLEFT or GDEFORMED

185 **Negative or zero argument**
Self explanatory

186 **Incorrect sub-array size**
Sub-array dimension is less than or equal to zero

187 **Pixel rectangle outside device limits**
Check pixel dimensions using gEnqPixelResolution()

188 **Pixel rectangle not contained within device limits**
Check pixel dimensions using gEnqPixelResolution()

189 **Negative or zero scale factor**
Self explanatory

190 **Orientation variable out of range**
Orientation must be in range 0-3

191 **Pixel data definition out of range**
Arguments are either negative or outside range of machine architecture

192 **Pixel data definition incompatible**
No. of relevant bits (nrb) > Bits per pixel (nbp) or total greater than number of bits per word or device working in direct colour mode

193 **Pixel information out of range**
Pixel data is outside colour range of device

194 **Invalid SDF highlight colour index**
Check number of colours available with gEnqColourInfo()

195 **Invalid pixel display switch**
Switch must be one of GOFF, GON or GBOUNDARY

196 **Pixel buffer overflow**
Pixel area width > internal buffer - contact software support

197 **Invalid drawing area identifier**
Identifier greater than number of auxiliary areas available on device

198 **Cannot open auxiliary drawing area**
Device error - check resources

199 **Invalid clipping mode**
Mode must be one of GNOCLIP, GHARD or GSOFT

200 **Empty GINO state stack**
Attempt to call gRestoreGinoState() without corresponding gSaveGinoState()

201 **Error accessing GINO state stack file**
Internal file read/write error - check access rights or contact software support

202 **Invalid thick line generation mode**
Mode must be one of GHARDWARE, GMIXWARE or GSOFTWARE

203 **Invalid image file type**
Unrecognized metafile image type

204 **Metafile too large for allocated workspace**
Increase local image array size - check dimensions with gEnqImageFile()

205 **Invalid colour definition flag**
Attempt to read 24bit image on indexed display in mode 0 - use mode 1

206 **Invalid shading mode**
Mode must be one of GNONE, GFLAT, GGOURAUD or GPHONG

207 **Invalid culling mode**
Culling must be either GON or GOFF

208 **Invalid blending mode**
Blending must be either GON or GOFF

209 **Invalid winding mode**
Winding must be either GANTICLOCKWISE or GCLOCKWISE

210 **Negative or too many points in facet**
There must be at least 3 and less 1024 points in a facet

211 **Illegal light number**
Must be in range 1-8

212 **Light not defined**
Attempt to switch light on before defined - use gDefineLightSource() first

213 **Illegal material index**
Must be range 1-256

214 **Invalid depth test mode**
Must be one of recognized modes

215 **Invalid facet filling style**
Style must be either GHOLLOW or GSOLID

216 **Invalid texture mapping mode**
Mode must be one of GOFF, GOVERLAY, GMODULATE or GBLEND

217 **Invalid texture mapping wrap mode**
Mode must be either GREPEAT or GCLAMP

218 **Invalid texture mapping filter**
Must be one of GOFF, GOVERLAY, GMODULATE or GBLEND

219 **Invalid texture coordinate generation mode**
Must be one of GOFF, GOBJECT or GSPHERICAL

220 **Texture map too large**
Texture map exceeds 1024x1024 limit

221 **Invalid fog mode**
Fog mode must be one of GNONE or GLINEAR, GEXP1 or GEXP2

222 **Invalid facet face**
Face must be either GFRONT or GBACK

223 **Invalid facet offset mode**
Use one of documented modes

224 **Invalid point storage mode**
Point storage mode must be one of GOFF, GSPACE, GPICTURE or GCLEAR

225 **Invalid task priority**
Priority must be one of GREALTIME, GHIGH, GNORMAL, GLOW or GIDLE

226 **Failed to allocate memory for metafile creation**
An attempt to allocate memory failed due to insufficient resources. Either increase amount of virtual memory or reduce size of metafile

227 **Fatal error creating metafile**
Contact software support

228 **Too few or too many points in shaded polyline**
There must be at least 2 and less than 1024 points in a shaded polyline

229 **Too few data points for interpolation**
There must be at least 2 points to carry out an interpolation

230 **Invalid interpolation switch**
Interpolation switch must be either of GXDATA, GYDATA or GZDATA

231 **No. of interpolated values exceeds output array size**
There are more intersections for the supplied data value than can be returned in the supplied output arrays. Increase the value of nptout and size of the ptout1 (and ptout2) arrays.

CGM Errors

The error number is followed by the error message which is accompanied by a brief description of the possible cause of the error.

1 **CGM not in proper state. Should be in state KMFCL**
An attempt has been made to begin an already open metafile

2 **CGM not in proper state. Should be KMFDS, KPIDS, KPIOP or KPICL**
An attempt has been made to process an external element in an invalid state

3 **CGM not in proper state. Should be in state KMFDS**
An attempt has been made to process a Metafile Descriptor element while not in Metafile Descriptor state

4 **CGM not in proper state. Should be in state KPIDS**
An attempt has been made to process a Picture Descriptor element while not in Picture Descriptor state

5 **CGM not in proper state. Should be in state KMFDS or KPICL**
An attempt has been made to end a metafile or open a picture from within an open picture

6 **CGM not in proper state. Should be in state KPIOP**
An attempt has been made to process a graphical primitive or attribute element while picture is not open

7 **CGM not in proper state. Should be in state KPIDS or KPIOP**
An attempt has been made to close a picture while picture is not open

8 **CGM not in proper state. Should be in state KPATX**
An attempt has been made to append while text final/not final flag is set to final

9 **CGM not in proper state. Should be in state KPIOP or KPATX**
As error number 6 except covering special case of elements which can occur in both states

10 **CGM not in proper state. Should be in state KMFDEF**
An attempt has been made to end metafile defaults replacement before a begin metafile defaults replacement has been encountered

20 **Invalid metafile**
An attempt has been made to leave the Metafile Descriptor state without writing the elements Metafile Version or Metafile Element List

30 **Unknown element**
An op-code has been encountered which is not included in the standard

45 **Invalid rectangle definition. Zero area**
A zero area rectangle has been defined within VDC Extent, Clip Rectangle or Rectangle Elements

50 **Element not included in metafile element list**
Self explanatory

100 **Invalid line bundle index**
A line bundle index which is negative or greater than indicated by Index Precision has been specified

101 **Invalid marker bundle index**
A marker bundle index which is negative or greater than indicated by Index Precision has been specified

102 **Invalid text bundle index**
A text bundle index which is negative or greater than indicated by Index Precision has been specified

103 **Invalid text font index**
A text font index which is negative or greater than indicated by Font List or Index Precision has been specified

104 **Invalid character set index**
A character set index which is negative or greater than indicated by Character Set List or Index Precision has been specified

105 **Invalid alternative character set index**
An alternative character set index which is negative or greater than indicated by Character Set List or Index Precision has been specified

106 **Invalid fill bundle index**
A fill bundle index which is negative or greater than indicated by Index Precision has been specified

107 **Invalid pattern index**
A pattern index which is negative or greater than indicated by Index Precision or Pattern Table has been specified

108 **Invalid edge bundle index**
An edge bundle index which is negative or greater than indicated by Index Precision has been specified

130 **Invalid line colour**
A line colour whose index is negative or greater than Maximum Colour Index or at least one of whose components lies outside the range indicated by Colour Value Extent has been specified

131 **Invalid marker colour**
A marker colour whose index is negative or greater than Maximum Colour Index or at least one of whose components lies outside the range indicated by Colour Value Extent has been specified

132 **Invalid text colour**
A text colour whose index is negative or greater than Maximum Colour Index or at least one of whose components lies outside the range indicated by Colour Value Extent has been specified

133 **Invalid fill colour**
A fill colour whose index is negative or greater than Maximum Colour Index or at least one of whose components lies outside the range indicated by Colour Value Extent has been specified

134 **Invalid edge colour**
An edge colour whose index is negative or greater than Maximum Colour Index or at least one of whose components lies outside the range indicated by Colour Value Extent has been specified

200 **Loss of integer precision**
An integer value outside the range indicated by Integer Precision has been specified

202 **Loss of real precision**
A real value outside the range indicated by Real Precision has been specified

204 **Real VDC lost precision**
A real VDC value outside the range indicated by VDC Real Precision has been specified

205 **Real point lost precision**
One of x,y in (a real point) lies outside the range indicated by VDC Real Precision

206 **Integer VDC lost precision**
An integer VDC value outside the range indicated by VDC Integer Precision has been specified

207 **Integer point lost precision**
One of x,y in (an integer point) lies outside the range indicated by VDC Integer Precision

210 **Invalid VDC distance**
A negative or ‘zero’ distance has been specified. e.g. Circle Radius, Character Height

220 **Point outside VDC Extent**
At least one of the coordinates of a point is greater (or less) than the relevant VDC EXTENT coordinate

230 **Invalid colour list**
An error list has been encountered when processing a colour list. Perhaps invalid type, colour specification greater than maximum

302 **Invalid line width**
A negative or ‘zero’ line width has been specified

304 **Invalid marker size**
A negative or ‘zero’ marker size has been specified

306 **Invalid edge width**
A negative or ‘zero’ edge width has been specified

310 **Invalid character expansion factor**
A negative character expansion factor has been specified

312 **Invalid character spacing**
A negative character spacing width has been specified

320 **Invalid test alignment**
A value outside the enumerated range for text alignment [0,4] & [0,6] has been specified

332 **Invalid text final/not-final flag**
A value other than 0 or 1 for either a closure flag or an edge visibility flag was specified

333 **Invalid closure/edge visibility flag**
A value other than 0 or 1 for either a closure flag or an edge visibility flag was specified

334 **Invalid line type**
A value greater than 5 for line type was specified

336 **Invalid marker type**
A value greater than 5 for marker type was specified

338 **Invalid text precision**
A value outside the range [0,2] for text precision was specified

340 **Invalid text path**
A value outside the range [0,2] for text path was specified

350 **Invalid interior style**
A value greater than 4 for interior style was specified

352 **Invalid hatch index**
A value greater than 6 for hatch index was specified

354 **Invalid edge type**
A value greater than 5 for edge type was specified

360 **Aspect source flag error**
An unacceptable value for aspect [0,17] or [506,511] or a value other than 0 or 1 for aspect source was specified

370 **Invalid message action flag**
A value other than 0 or 1 was encountered as a message action flag

401 **Invalid VDC type**
A value other than 0 or 1 has been specified

402 **Invalid maximum colour index**
A maximum colour index which is either negative or greater than that indicated by Colour Index Precision

403 **Invalid colour value extent**
A colour value extent where either minimum colour value is greater than maximum colour value or a component lies outside the range indicated by Colour Precision has been specified

404 **Invalid real precision**
An inconsistent set of parameters for real precision (e.g. smallest real code greater than largest real code) or a value other than 0 or 1 for exponents allowed flag has been specified. An invalid combination of real precision for binary encoding

405 **Invalid integer precision**
An invalid value for integer precision was encountered. A number less than 1. Or an unspecified precision for binary encoding

406 **Invalid background colour**
A background colour at least one of whose components lies outside the range indicated by Colour Value Extent has been specified

407 **Invalid auxiliary colour**
An auxiliary colour whose index is negative or greater than Maximum Colour Index or at least one of whose components lies outside the range indicated by Colour Value Extent has been specified

408 **Invalid index precision**
An invalid value for index precision was encountered. A number less than 1. Or an Unspecified precision for binary encoding

409 **Invalid colour precision**
An invalid value for colour precision was encountered. A number less than 1. Or an unspecified precision for binary encoding

410 **Invalid colour index precision**
An invalid value for colour index precision was encountered. A number less than 1. Or an unspecified precision for binary encoding

411 **Invalid metafile element list**
Either an invalid op-code or a value greater than 1 has been specified in the element list

412 **Invalid character set type**
A value other than the range [0,4] has been specified as a character set type

413 **Invalid character coding announcer**
A value greater than 3 as a character coding announcer has been specified

414 **Invalid scaling mode**
A value other than 0 or 1 for scaling mode has been specified

415 **Invalid colour selection mode**
A value other than 0 or 1 for colour selection mode has been specified

416 **Invalid line width selection mode**
A value other than 0 or 1 for line width selection mode has been specified

417 **Invalid marker size selection mode**
A value other than 0 or 1 for marker size selection mode has been specified

418 **Invalid edge width selection mode**
A value other than 0 or 1 for edge width selection mode has been specified

419 **Invalid scale factor**
A negative or ‘zero’ value for metric scale factor has been specified

422 **Invalid VDC real precision**
As for error number 404

423 **Invalid VDC integer precision**
As for error number 405

424 **Invalid transparency indicator**
A value other than 0 or 1 for the transparency indicator was specified

426 **Invalid clip indicator**
A value other than 0 or 1 for the clip indicator was specified

700 **Invalid CGM coding type**
CGM file must be character or binary coding

708 **No file opened for interpreter**
A partial interpretation routine called before gOpenCGMFile

709 **Error opening CGM file**
Self explanatory

710 **Error while writing to metafile**
Self explanatory

711 **Error while reading from CGM file**
Self explanatory

712 **End of CGM metafile reached**
Self explanatory

720 **Input buffer overflow**
While storing data to send to (or read from) a file, the maximum has been exceeded. This value is specified by the implementor

740 **Invalid parameter data found**
Some redundant parameter data at the end of an element was encountered (i.e. An op-code was not found immediately following the end of an element)

750 **String data overflow**
A string variable has a length greater than the maximum string length implemented (255)

760 **Colour buffer overflow**
A colour definition or cell array definition exceeds implementation maximum (2048)

770 **Polyline/polygon buffer overflow**
Number of points in polyline or polygon exceeds implemented maximum (1024)

999 **Unknown error**

System Input and Output Errors

GINO System error - File unit number out of range (NDEVIC=nnnn)

A device file configuration setting has been set greater than the highest permitted file unit number for your machine or less than -8.

GINO System error - Cannot open file (NDEVIC=nnnn, ERROR CODE =nnnn)

A system error has occurred when opening the output file or port.

GINO System error - Cannot input from output only file

A GINO input routine has been used when outputting to a disk file.

GINO System error - Error (nnnn) reading from device

A system error has occurred during input from the desired device.

GINO System error - Error (nnnn) writing to output file

A system error has occurred during output to the desired device.

For OpenVMS and UNIX error codes, please refer to the relevant system documentation.

Configuration File Errors

***** Missing or Empty GINO Configuration File - Program Aborted! *****

Check the *Getting Started* guide for the correct location of the Configuration File. If it is missing, either it has been accidentally deleted or you have been supplied with an update pack assuming that you have a configuration file from a previous release. Contact Bradly Associates or your dealer if this error continues.

***** Incorrect GINO-C/F Serial No. - Program Aborted! *****

Check the contents of your configuration file. There should be a serial number line beginning with FSERIAL= or CSERIAL = followed by a string of ASCII characters. Contact Bradly Associates or your dealer if this error continues.

***** GINO-C/F Evaluation period expired! *****

You have a temporary licence which has now expired - Contact Bradly Associates or your dealer.

Appendix F

GINO STRUCTURES

Structures Introduction

There follows an alphabetical listing of the GINO structures for reference purposes. These structures are defined within the GINO-C header file <gino-c.h>. Equivalent structures exist for the Fortran 90 implementation of GINO using the derived type construct and are included in the Fortran 90 interface module for GINO.

GBRKSTY

```
typedef struct {
   int    mode;
   float  repeat;
   float  dash;
   float  dot;
} GBRKSTY;
```

```
type GBRKSTY
   integer:: mode
   real   :: repeat
   real   :: dash
   real   :: dot
end type
```

GCHASTY

```
typedef struct {
   int   type;
   float width;
   float height;
   int   size;
   float slant;
   float angle;
} GCHASTY;
```

```
type GCHASTY
   integer:: type
   real   :: width
   real   :: height
   integer:: size
   real   :: slant
   real   :: angle
end type
```

GDATE

```
typedef struct {
   int  year;
   int  month;
   int  day;
} GDATE;
```

```
type GDATE
   integer:: year
   integer:: month
   integer:: day
end type
```

GDEVSTATE

```
typedef struct {
   char   name[8];
   int    ddver;
   int    ndevty;
   GFILE *ndev;
   int    ntype;
   float  xmm;
   float  xdu;
   float  unitsc;
   int    atrib;
   int    ahard;
   float  arctol;
   int    cwt;
   int    nbrk;
   int    nlend;
   int    thick;
   int    chard;
   int    nchard;
   int    cangm;
   int    rectfi;
   int    npolmx;
   int    nvermx;
   int    ndcmax;
   int    ndtmax;
   int    whard;
   int    nsegmx;
   int    ncurtp;
   int    nevetp;
   int    nquemx;
   float  xpixel;
   float  ypixel;
   int    npixdp;
   int    nxpix;
   int    nypix;
   int    maxaux;
   int    dddim;
   int    mhard;
   int    dialog;
} GDEVSTATE;
```

```
type GDEVSTATE
   character    :: name*8
   integer:: ddver
   integer:: ndevty
   integer:: ndev
   integer:: ntype
   real   :: xmm
   real   :: xdu
   real   :: unitsc
   integer:: atrib
   integer:: ahard
   real   :: arctol
   integer:: cwt
   integer:: nbrk
   integer:: nlend
   integer:: thick
   integer:: chard
   integer:: nchard
   integer:: cangm
   integer:: rectfi
   integer:: npolmx
   integer:: nvermx
   integer:: ndcmax
   integer:: ndtmax
   integer:: whard
   integer:: nsegmx
   integer:: ncurtp
   integer:: nevetp
   integer:: nquemx
   real   :: xpixel
   real   :: ypixel
   integer:: npixdp
   integer:: nxpix
   integer:: nypix
   integer:: maxaux
   integer:: dddim
   integer:: mhard
   integer:: dialog
end type
```

GDIM

```
typedef struct {
   float  xpap;
   float  ypap;
} GDIM;
```

```
type GDIM
   real   :: xpap
   real   :: ypap
end type
```

GEVEREC

```
typedef struct {
   int    key;
   int    impkey;
   int    impdat;
   int    nseg;
   GPOINT pos;
   int    nargs;
   float  args[80];
   int    iargs[80];
} GEVEREC;
```

```
type GEVEREC
   integer:: key
   integer:: impkey
   integer:: impdat
   integer:: nseg
   type (GPOINT):: pos
   integer:: nargs
   real   :: args ( 80 )
   integer:: iargs ( 80 )
end type
```

GFILE

```
typedef struct ginofile {
   int unit;
   struct ginofile *nextfcb;
} GFILE;
```

GFNTFILSTY

```
typedef struct {
   int  type;
   int  ffill;
   int  fline;
   int  bfill;
   int  bline;
} GFNTFILSTY;
```

```
type GFNTFILSTY
   integer:: type
   integer:: ffill
   integer:: fline
   integer:: bfill
   integer:: bline
end type
```

GFOGATT

```
typedef struct {
   int    mode;
   int    colour;
   float  density;
   float  start;
   float  end;
} GFOGATT;
```

```
type GFOGATT
   integer:: mode
   integer:: colour
   real   :: density
   real   :: start
   real   :: end
end type
```

GHATSTY

```
typedef struct {
   float  pitch;
   float  angle;
   float  xshift;
   float  yshift;
   float  xshear;
   int    xhatch;
} GHATSTY;
```

```
type GHATSTY
   real   :: pitch
   real   :: angle
   real   :: xshift
   real   :: yshift
   real   :: xshear
   integer:: xhatch
end type
```

GHLSSTY

```
typedef struct {
   float  hue;
   float  light;
   float  sat;
} GHLSSTY;
```

```
type GHLSSTY
   real   :: hue
   real   :: light
   real   :: sat
end type
```

GHSVSTY

```
typedef struct {
   float  hue;
   float  sat;
   float  value;
} GHSVSTY;
```

```
type GHSVSTY
   real   :: hue
   real   :: sat
   real   :: value
end type
```

GIMPLEMENTATION

```
typedef struct {
   float  rmin;
   float  rmax;
   float  rsmall;
   float  rsig;
   int    imin;
   int    imax;
   int    nipr;
   int    nfmax;
   int    nbits;
   int    nbmask;
   int    iwtres;
   int    nbyter;
   int    nfumin;
   int    nfumax;
   int    ndevdf;
   int    ndsavf;
   int    nfdinp;
   int    nfdout;
   int    nfertr;
   int    nfmess;
   int    nfsdf;
   int    nffont;
   int    nficon;
   int    nfstat;
   int    nfimpl;
   int    nflice;
   int    iso;
   char   dsep;
} GIMPLEMENTATION;
```

```
type GIMPLEMENTATION
   real   :: rmin
   real   :: rmax
   real   :: rsmall
   real   :: rsig
   integer:: imin
   integer:: imax
   integer:: nipr
   integer:: nfmax
   integer:: nbits
   integer:: nbmask
   integer:: iwtres
   integer:: nbyter
   integer:: nfumin
   integer:: nfumax
   integer:: ndevdf
   integer:: ndsavf
   integer:: nfdinp
   integer:: nfdout
   integer:: nfertr
   integer:: nfmess
   integer:: nfsdf
   integer:: nffont
   integer:: nficon
   integer:: nfstat
   integer:: nfimpl
   integer:: nflice
   integer:: iso
   character:: dsep*1
end type
```

GLIBSTATE

```
typedef struct {
   int  gino;
   int  graf;
   int  surf;
   int  menu
} GLIBSTATE;
```

```
type GLIBSTATE
   integer:: gino
   integer:: graf
   integer:: surf
   integer:: menu
end type
```

GLIMIT

```
typedef struct {
   float  xmin;
   float  xmax;
   float  ymin;
   float  ymax;
} GLIMIT;
```

```
type GLIMIT
   real   :: xmin
   real   :: xmax
   real   :: ymin
   real   :: ymax
end type
```

GLIMIT3

```
typedef struct {
   float  xmin;
   float  xmax;
   float  ymin;
   float  ymax;
   float  zmin;
   float  zmax;
} GLIMIT3;
```

```
type GLIMIT3
   real   :: xmin
   real   :: xmax
   real   :: ymin
   real   :: ymax
   real   :: zmin
   real   :: zmax
end type
```

GLINSTY

```
typedef struct {
   int    vis;
   int    brk;
   int    col;
   float  width;
   int    type;
   int    end;
} GLINSTY;
```

```
type GLINSTY
   integer:: vis
   integer:: brk
   integer:: col
   real   :: width
   integer:: type
   integer:: end
end type
```

GLITATT

```
typedef struct {
   int     state;
   int     type;
   int     col;
   int     spec;
   GPOINT3 pos;
   GPOINT3 dir;
   float   att1;
   float   att2;
   float   conc;
   float   spang;
} GLITATT;
```

```
type GLITATT
   integer:: state
   integer:: type
   integer:: col
   integer:: spec
   type (GPOINT3):: pos
   type (GPOINT3):: dir
   real   :: att1
   real   :: att2
   real   :: conc
   real   :: spang
end type
```

GMAT2D

```
typedef  float  GMAT2D [2][3];
```

GMAT3D

```
typedef  float  GMAT3D [4][4];
```

GMATSTY

```
typedef struct {
   float ambient;
   float diffuse;
   float specular;
   float shine;
   float trans;
} GMATSTY;
```

```
type GMATSTY
   real :: ambient
   real :: diffuse
   real :: specular
   real :: shine
   real :: trans
end type
```

GMATV

```
typedef  float  GMATV [15];
```

GPICATT

```
typedef struct {
   int     exist;
   int     vis;
   int     sens;
   int     mark;
   GPOINT3 anchor;
} GPICATT;
```

```
type GPICATT
   integer:: exist
   integer:: vis
   integer:: sens
   integer:: mark
   type (GPOINT3)  :: anchor
end type
```

GPIXEL

```
typedef struct {
   int  ix;
   int  iy;
} GPIXEL;
```

```
type GPIXEL
   integer:: ix
   integer:: iy
end type
```

GPOINT

```
typedef struct {
   float  x;
   float  y;
} GPOINT;
```

```
type GPOINT
   real   :: x
   real   :: y
end type
```

GPOINT3

```
typedef struct {
   float  x;
   float  y;
   float  z;
} GPOINT3;
```

```
type GPOINT3
   real   :: x
   real   :: y
   real   :: z
end type
```

GPOLYGON

```
typedef struct {
   int     nvert;
   GPOINT *verts;
} GPOLYGON;
```

```
type GPOLYGON
   integer:: nvert
   type (GPOINT),dimension(:), &
                 pointer :: verts
end type
```

GPOLYGON3

```
typedef struct {
   int      nvert;
   GPOINT3 *verts;
} GPOLYGON3;
```

```
type GPOLYGON3
  integer:: nvert
  type (GPOINT3),dimension(:), &
                 pointer ::verts
end type
```

GRGBSTY

```
typedef struct {
   float  red;
   float  green;
   float  blue;
} GRGBSTY;
```

```
type GRGBSTY
   real   :: red
   real   :: green
   real   :: blue
end type
```

GSHADING

```
typedef struct {
   int  mode;
   int  culling;
   int  blending;
   int  winding;
} GSHADING;
```

```
type GSHADING
   integer:: mode
   integer:: culling
   integer:: blending
   integer:: winding
end type
```

GTEXATT

```
typedef struct {
   int mode;
   int blendcol;
   int wraps;
   int wrapt;
   int maxfil;
   int minfil;
   int bordercol;
} GTEXATT;
```

```
type GTEXATT
   integer :: mode
   integer :: blendcol
   integer :: wraps
   integer :: wrapt
   integer :: maxfil
   integer :: minfil
   integer :: bordercol
end type
```

GTIME

```
typedef struct {
   int  hour;
   int  min;
   int  sec;
   int  millsec;
} GTIME;
```

```
type GTIME
   integer:: hour
   integer:: min
   integer:: sec
   integer:: millsec
end type
```

GVIEWSTATE

```
typedef struct {
   int     mode;
   int     cflag;
   int     upflag;
   GPOINT3 dir;
   GPOINT3 centre;
   float   dist;
   GPOINT  shift;
   GPOINT3 upvec;
} GVIEWSTATE;
```

```
type GVIEWSTATE
   integer        :: mode
   integer        :: cflag
   integer        :: upflag
   type (GPOINT3) :: dir
   type (GPOINT3) :: centre
   real           :: dist
   type (GPOINT)  :: shift
   type (GPOINT3) :: upvec
end type
```

Appendix G

CROSS REFERENCES

Cross References Introduction

GINO is supplied as either a C library or FORTRAN library. The FORTRAN library includes two bindings; an F77 binding using short names and simple arguments and an F90 binding using long names and structures/optional arguments as appropriate. This Appendix gives the cross-references from both short name to long name and vice-versa.

F77-F90 Cross-Reference

F77 names	F90 names
AKIBY2	gDrawAkimaBy2D
AKITO2	gDrawAkimaTo2D
ARCBY2	gDrawArcBy2D
ARCBY3	gDrawArcBy3D
ARCENQ	gEnqArcState
ARCINC	gSetArcIncrement
ARCSWI	gSetArcMode
ARCTO2	gDrawArcTo2D
ARCTO3	gDrawArcTo3D
ARCTOL	gSetArcTolerance
AUXCLO	gCloseAuxDrawingArea
AUXOPN	gOpenAuxDrawingArea
AUXSEL	gSelectDrawingArea
BEZBY2	gDrawBezierBy2D

BEZBY3	gDrawBezierBy3D
BEZ|EL2	gElevateBezierTo2D
BEZ|EL3	gElevateBezier3D
BEZRE2	gReduceBezier2D
BEZRE3	gReduceBezier3D
BEZSPH	gDrawBezierSphere
BEZSUR	gDrawBezierSurface
BEZ|TO2	gDrawBezierTo2D
BEZTO3	gDrawBezierTo3D
BEZVOL	gDrawBezierVolume
BLDTM2	gBuildMatrix2D
BLDTM3	gBuildMatrix3D
BOX	gDrawBox
BOXBY3	gDrawBox
BOXTO3	gDrawBox
BRKDEF	gDefineBrokenLineStyle
BRKENQ	gEnqBrokenLineStyle
BRKMOD	gSwitchBrokenLineStyles
BRKSWI	gSetBrokenLineMode
BROENQ	gEnqBrokenLine
BROKEN	gSetBrokenLine
BUFLIM	gSetWorkspaceLimit
BUFLNQ	gEnqWorkspaceLimit
CALNOR	gReturnPlanarNormal
CELDRA	gDrawCellArray
CHAANG	gSetStrAngle
CHAASC	gDisplayAsciiChar
CHABEG	gStartTextBlock
CHABY2	gDisplayStrPolylineBy2D
CHAENQ	gEnqCharAttribs
CHAEXI	gSetStrExponent
CHAEXP	gDisplayRealExponent
CHAFIT	gFitCharStr
CHAFIX	gDisplayRealFixed
CHAFLO	gDisplayRealFloat
CHAFNT	gSetCharFont

CHAHAR	gSetHardCharSize
CHAILS	gSetInterlineSpace
CHAINT	gDisplayInteger
CHAJUS	gSetStrJustify
CHAMOD	gSetAlphaMode
CHANUL	gDefineNullChar
CHANXT	gMoveToNextLine
CHAPNT	gSetCharSizePoint
CHASIZ	gSetCharSize
CHASTR	gDisplayStr
CHASWI	gSetCharTransformMode
CHATO2	gDisplayStrPolylineTo2D
CHATRA	gEnqCharTransform
CHAUND	gSetStrUnderscore
CLPENQ	gEnqClippingMode
CLPSWI	gSetClippingMode
COLENQ	gEnqLineColour
COLINF	gEnqColourInfo
COLSET	gSetColourInfo
COMTM2	gCombineMatrix2D
COMTM3	gCombineMatrix3D
CONBY3	gDrawCone
CONE	gDrawCone
CONTO3	gDrawCone
CUBE	gDrawCube
CURACQ	gEnqCursorAction
CURACT	gSetCursorAction
CURBY2	gDrawCurveBy2D
CURENQ	gEnqCurveAttribs2D
CUREQ3	gEnqCurveAttribs3D
CURPOS	gSetCursorPos
CURSET	gSetCurveAttribs2D
CURSOR	gGetCursorEvent
CURST3	gSetCurveAttribs3D
CURTO2	gDrawCurveTo2D
CURTYP	gSetCursorType

CURTYQ	gEnqCursorType
CYLBY3	gDrawCylinder
CYLIND	gDrawCylinder
CYLTO3	gDrawCylinder
DEBUG	gDebug
DEBUGT	gSetDebugSwitch
DEPENQ	gEnqDepthMode
DEPMOD	gSetDepthMode
DEVEND	gCloseDevice
DEVFIL	gSetDeviceFilename
DEVINF	gEnqDeviceState
DEVPAP	gSetDrawingLimits
DEVSUS	gSuspendDevice
DEVTTL	gSetDeviceTitle
DIAVIS	gSetDialogueVis
DIRDAT	gReturnDirDate
DIRDEL	gRemoveDir
DIRENQ	gEnqWorkingDir
DIRFUL	gGetFullDirList
DIRLIS	gGetDirList
DIRMK	gMakeDir
DIRSET	gSetWorkingDir
DRAG	gDragSeg
EDITR2	gEditSeg2D
EDITR3	gEditSeg3D
ENDENQ	gEnqLineEnd
ERRDEV	gSetErrorFile
ERRENQ	gEnqLastErrors
ERRGET	gEnqNumberOfErrors
ERRMAX	gSetMaxErrorLimit
ERRSET	gSetErrorTrap
ERRSWI	gSetErrorMode
EVEDEL	gRemoveEventType
EVEENQ	gGetEventRecord
EVENT	gWaitForEvent
EVESET	gAddEventType

EXIENQ	gEnqStrExponent
FACET	gDrawFacet
FACETC	gDrawFacet
FACETN	gDrawFacet
FACETT	gDrawFacet
FACFIL	gSetFacetFillStyle
FACFIQ	gEnqFacetFillStyle
FACMAQ	gEnqFacetMaterialProps
FACMAT	gSetFacetMaterialProps
FACOSM	gSetFacetOffsetMode
FACOSQ	gEnqFacetOffsetMode
FILCLS	gFclose
FILCOP	gCopyFile
FILDEL	gRemoveFile
FILOPN	gFopen
FILREN	gRenameFile
FILSWI	gSetFillMode
FLUSHG	gFlushGraphics
FNTENQ	gEnqFontStyle
FNTLIS	gEnqHardFontList
FNTREP	gSetFontForm
FNTSPA	gSetFontSpacing
FNTSTY	gSetFontFillStyle
FNTWGT	gSetFontWeight
FOGATT	gDefineFog
FOGENQ	gEnqFog
FOGMOD	gDefineFog
GDELAY	gTimeDelay
GETDRA	gGetDrawing
GETIMG	gGetImageFile
GETPIC	gGetPicture
GETPNT	gGetPixel
GETRAN	gGetRand
GETSEQ	gEnqSavdraSegAttribs
GETSNQ	gEnqSavdraSegList
GFILL	gFillSelectedPolygons

GINCON	gEnqConfigStatus
GINEND	gCloseGino
GINENQ	gEnqGinoState
GINO	gOpenGino
GINRES	gRestoreGinoState
GINSAV	gSaveGinoState
GRAVIS	gSetGraphicsVis
GRPENQ	gEnqSegGroup
GSOUND	gPlaySound
HARCHA	gSetHardChars
HATDEF	gDefineHatchStyle
HATENQ	gEnqHatchStyle
HLSDEF	gDefineHLS
HLSENQ	gEnqHLS
HSVDEF	gDefineHSV
HSVENQ	gEnqHSV
IMGENQ	gEnqImageFile
ITALIC	gSetItalicAngle
JUSENQ	gEnqStrJustify
KEYSTA	gEnqKeyState
LINBY2	gDrawLineBy2D
LINBY3	gDrawLineBy3D
LINCOL	gSetLineColour
LINDEF	gDefineLineStyle
LINEND	gSetLineEnd
LINENQ	gEnqLineStyle
LINSAV	gSaveLineStyle
LINSEL	gSetLineStyle
LINTO2	gDrawLineTo2D
LINTO3	gDrawLineTo3D
LINVIS	gSetLineVis
LINWID	gSetLineWidth
LITAMB	gDefineLightSource
LITDIR	gDefineLightSource
LITENQ	gEnqLightAttribs
LITPNT	gDefineLightSource

LITSPC	gDefineLightSource
LITSPT	gDefineLightSource
LITSWI	gSetLightSwitch
MARKER	gDrawMarker
MASENQ	gEnqMaskState
MASK2	gSetMask2D
MASSWI	gSetMaskMode
MATATQ	gEnqMaterialAttribs
MATCOL	gSetMaterialColour
MATDEF	gDefineMaterial
MATENQ	gEnqMaterial
MIXCHA	gSetMixedChars
MODBEG	gStartBatchUpdate
MODEND	gEndBatchUpdate
MOUENQ	gEnqMousePos
MOUSET	gSetMousePos
MOVBY2	gMoveBy2D
MOVBY3	gMoveBy3D
MOVTO2	gMoveTo2D
MOVTO3	gMoveTo3D
OBJANG	-
OBJCOM	-
PAPENQ	gEnqDrawingLimits
PAPMAX	gEnqMaxDrawingLimits
PENENQ	gEnqSelectedPen
PENTYP	gSetPenType
PICBEG	gOpenSeg
PICBY	gMoveSegBy2D
PICCOP	gCopySeg
PICDEL	gDeleteSeg
PICDGP	gRemoveSegGroup
PICDRA	gDrawSeg
PICEND	gCloseSeg
PICENQ	gEnqSegAttribs
PICEXT	gExtendSeg
PICFGP	gDefineSegGroup

PICHIT	gEnqSegHit
PICMAR	gMarkSeg
PICNUM	gEnqOpenSeg
PICREF	gInsertSegRef
PICREN	gRenameSeg
PICSEN	gSetSegHit
PICTAG	gInsertSegTag
PICTO	gMoveSegTo2D
PICTQ2	gEnqSegTransform2D
PICTR2	gSetSegTransform2D
PICTRA	gSetSegTransform
PICTRQ	gEnqSegTransform
PICVIS	gSetSegVis
PIXATQ	gEnqPixelAttribs
PIXCOP	gCopyPixelArea
PIXDEF	gDefinePixelPacking
PIXDEQ	gEnqPixelPacking
PIXDRA	gDrawPixelArea
PIXENQ	gEnqPixelResolution
PIXGET	gGetPixelArea
PIXPNT	gDrawPixel
PIXPOS	gEnqPosOfPixel
PIXREP	gSetPixelReplication
PIXSWI	gSetPixelDisplayMode
PIXTRA	gSetPixelTransform
PLSTO2	gDrawPolylineSet2D
PLSTO3	gDrawPolylineSet3D
PNTBUF	gDefinePointWorkspace
PNTENQ	gEnqPointMode
PNTSWI	gSetPointMode
POFBY2	gFillPolygonBy2D
POFBY3	gFillPolygonBy3D
POFTO2	gFillPolygonTo2D
POFTO3	gFillPolygonTo3D
POLBEG	gStartPolygon
POLBND	gDrawPolygonBound

POLBUF	gDefinePolygonWorkspace
POLBY2	gDrawPolylineBy2D
POLBY3	gDrawPolylineBy3D
POLCLE	gClearPolygonWorkspace
POLEND	gEndPolygon
POLENQ	gEnqPolygonWorkspace
POLHIT	gPolygonHit
POLIDN	gSetPolygonIdent
POLLIS	gEnqPolygonList
POLMAS	gSetPolygonMask
POLSEL	gSelectPolygons
POLSWI	gSetPolygonMode
POLTO2	gDrawPolylineTo2D
POLTO3	gDrawPolylineTo3D
POLWIN	gSetPolygonWindow
POMLIS	gEnqPolygonMaskList
POSPIC	gEnqPicturePos
POSPIX	gEnqPixelPos
POSSPA	gEnqSpacePos
POSTO3	gDrawShadedPolylineTo3D
POTTO3	gDrawShadedPolylineTo3D
POWLIS	gEnqPolygonWindowList
PSFTO2	gFillPolygonSet2D
PSFTO3	gFillPolygonSet3D
PT2ENQ	gReturnInternalPoints2D
PT3ENQ	gReturnInternalPoints3D
PT2INT	gInterpolateData2D
PT3INT	gInterpolateData3D
QUEDEL	gDeleteEventQueue
QUEUE	gEnqQueueLength
RANGGP	gDefineGroupRange
RECT3D	gDrawRect3D
RGBDEF	gDefineRGB
RGBENQ	gEnqRGB
ROTAT2	gRotate2D
ROTAT3	gRotate3D

RTFILL	gFillRect
RULBEZ	gDrawRuledBezierSurface
SAVENQ	gEnqSavdraDimension
SCALE2	gScale2D
SCALE3	gScale3D
SCRCLE	gNewDrawing
SDFARC	gArchiveSegs
SDFBFQ	gEnqSegWorkspace
SDFBUF	gDefineSegWorkspace
SDFMAR	gSetSegMarkColour
SDFRES	gRetrieveSegs
SDFSWI	gSetSegMode
SETMAT	gSetMaterialIndex
SETPRI	gSetSysPriority
SETRAN	gSetRandSeed
SETVP2	gSetViewport2D
SETVP3	gSetViewport3D
SHADOW	gCreatePlanarShadowMatrix
SHAENQ	gEnqShadingMode
SHAMOD	gSetShadingMode
SHEAR2	gShear2D
SHEAR3	gShear3D
SHIFT2	gShift2D
SHIFT3	gShift3D
SOFCHA	gSetSoftChars
SPHERE	gDrawSphere
SPLBY2	gDrawSplineBy2D
SPLBY3	gDrawSplineBy3D
SPLSUR	gDrawSplineSurface
SPLTEN	gSetSplineTension
SPLTO2	gDrawSplineTo2D
SPLTO3	gDrawSplineTo3D
STRESC	gSetEscapeChar
STRESQ	gEnqEscapeChar
STREXP	gConvertRealExponent
STRFIX	gConvertRealFixed

STRFLO	gConvertRealFloat
STRINF	gReturnStrInfo
STRINT	gConvertInteger
SWEBEZ	gDrawSweptBezierSurface
SYMBOL	gDrawMarker
SYMBY2	gDrawPolymarkerBy2D
SYMBY3	gDrawPolymarkerBy3D
SYMTO2	gDrawPolymarkerTo2D
SYMTO3	gDrawPolymarkerTo3D
SYSARG	gEnqSysArgs
SYSCOM	gExecuteSysCommand
SYSDAS	gEnqSysDateStr
SYSDAT	gEnqSysDate
SYSENV	gEnqSysEnviron
SYSNAM	gEnqSysUsername
SYSPRI	gEnqSysPriority
SYSTIM	gEnqSysTime
SYSTTM	gEnqSysTime
TABBEZ	gDrawTabulatedBezierSurface
TBKENQ	gEnqTextBlockAttribs
TENENQ	gEnqSplineTension
TRABEG	gPushTransform
TRACER	gSetTracerMode
TRAEND	gPopTransform
TRAENQ	gEnqTransformState
TRAMU2	gModifyTransform2D
TRAMU3	gModifyTransform3D
TRANS2	gTransformPoint2D
TRANS3	gTransformPoint3D
TRANS4	gTransformHomogPoint3D
TRANSF	gSetTransform
TRAPIC	gSetTransformMode
TRARES	gRestoreTransform
TRASA2	gGetTransform2D
TRASA3	gGetTransform3D
TRASAV	gSaveTransform

TRASE2	gSetTransform2D
TRASE3	gSetTransform3D
TRASEQ	gEnqViewTransformMode
TRASWI	gSetViewTransformMode
TRUCOL	gTrueCol
TRULEN	gTrueLen
TXMATQ	gEnqTextureMappingMode
TXMATT	gSetTextureMappingMode
TXMENQ	gEnqTextureMappingMode
TXMGEN	gSetTextureCoordGeneration
TXMGEQ	gEnqTextureCoordGeneration
TXMMAP	gDefineTexture
TXMMOD	gSetTextureMappingMode
TYPENQ	gEnqPenType
UNDENQ	gEnqStrUnderscore
UNITS	gDefinePictureUnits
UNTRA2	gUntransformPoint2D
UNTRA3	gUntransformPoint3D
UNTRA4	gUntransformHomogPoint3D
VALONG	gMoveViewCentre
VIEW	gGenerateView
VIEWSE	gSetViewAxis
VIEWUP	gUpdateView
VINIT	gInitView
VISENQ	gEnqLineVis
VMULT	gModifyView
VOLBY3	gDrawVolume
VOLTO3	gDrawVolume
VOLUME	gDrawVolume
VP2CLE	gClearViewport
VP2ENQ	gEnqViewport2D
VP3ENQ	gEnqViewport3D
VPARAL	gDefineParallelView
VPERSP	gSetViewEyeDistance
VPOINT	gDefinePerspView
VPOSIT	gPosViewCentre

VPTCLP	gSetViewportClipSwitch
VPTENQ	gEnqViewportState
VPTSWI	gSetViewportMode
VPTSWQ	gEnqViewportMode
VRESET	gSetViewParams, gSetViewState
VROTAT	gViewRotate
VSAVE	gGetViewParams, gGetViewState
VSHIFT	gViewShift
VSPHER	gDefineSphericalView
VTURN	gViewTurn
VUPDIR	gSetViewUpDirection
VZOOM	gSetViewPlaneDistance
WEDGE	gDrawWedge
WEGBY3	gDrawWedge
WEGTO3	gDrawWedge
WIDENQ	gEnqLineWidth
WIDSCA	gSetLineWidthScaling
WIDSCQ	gEnqLineWidthScaling
WIDSWI	gSetLineWidthMode
WIDSWQ	gEnqLineWidthMode
WINDO2	gSetWindow2D
WINDO3	gSetWindow3D
WINDOW	gSetWindowMode
WINENQ	gEnqWindowState
WINSWI	gSetWindowMode

F90-F77 Cross-Reference

F90 names	F77 names
gAddEventType	EVESET
gArchiveSegs	SDFARC
gBuildMatrix2D	BLDTM2
gBuildMatrix3D	BLDTM3
gClearPolygonWorkspace	POLCLE
gClearViewport	VP2CLE
gCloseAuxDrawingArea	AUXCLO

gCloseDevice	DEVEND
gCloseGino	GINEND
gCloseSeg	PICEND
gCombineMatrix2D	COMTM2
gCombineMatrix3D	COMTM3
gConvertInteger	STRINT
gConvertRealExponent	STREXP
gConvertRealFixed	STRFIX
gConvertRealFloat	STRFLO
gCopyPixelArea	PIXCOP
gCopyFile	FILCOP
gCopySeg	PICCOP
gCreatePlanarShadowMatrix	SHADOW
gDebug	DEBUG
gDefineBrokenLineStyle	BRKDEF
gDefineFog	FOGATT, FOGMOD
gDefineGroupRange	RANGGP
gDefineHatchStyle	HATDEF
gDefineHLS	HLSDEF
gDefineHSV	HSVDEF
gDefineLightSource	LITAMB, LITDIR, LITPNT, LITSPC, LITSPT
gDefineLineStyle	LINDEF
gDefineMaterial	MATDEF
gDefineNullChar	CHANUL
gDefineParallelView	VPARAL
gDefinePerspView	VPOINT
gDefinePictureUnits	UNITS
gDefinePixelPacking	PIXDEF
gDefinePointWorkspace	PNTBUF
gDefinePolygonWorkspace	POLBUF
gDefineRGB	RGBDEF
gDefineSegGroup	PICFGP
gDefineSegWorkspace	SDFBUF
gDefineSphericalView	VSPHER
gDefineTexture	TXMMAP

gDeleteEventQueue	QUEDEL
gDeleteSeg	PICDEL
gDisplayAsciiChar	CHAASC
gDisplayInteger	CHAINT
gDisplayRealExponent	CHAEXP
gDisplayRealFixed	CHAFIX
gDisplayRealFloat	CHAFLO
gDisplayStr	CHASTR
gDisplayStrPolylineBy2D	CHABY2
gDisplayStrPolylineTo2D	CHATO2
gDragSeg	DRAG
gDrawAkimaBy2D	AKIBY2
gDrawAkimaTo2D	AKITO2
gDrawArcBy2D	ARCBY2
gDrawArcBy3D	ARCBY3
gDrawArcTo2D	ARCTO2
gDrawArcTo3D	ARCTO3
gDrawBezierBy2D	BEZBY2
gDrawBezierBy3D	BEZBY3
gDrawBezierSphere	BEZSPH
gDrawBezierSurface	BEZSUR
gDrawBezierTo2D	BEZTO2
gDrawBezierTo3D	BEZTO3
gDrawBezierVolume	BEZVOL
gDrawBox	BOX, BOXBY3, BOXTO3
gDrawCellArray	CELDRA
gDrawCone	CONBY3, CONE, CONTO3
gDrawCube	CUBE
gDrawCurveBy2D	CURBY2
gDrawCurveTo2D	CURTO2
gDrawCylinder	CYLBY3, CYLIND, CYLTO3
gDrawFacet	FACET, FACETC, FACETN, FACETT,FACETX
gDrawLineBy2D	LINBY2
gDrawLineBy3D	LINBY3
gDrawLineTo2D	LINTO2

gDrawLineTo3D	LINTO3
gDrawMarker	MARKER, SYMBOL
gDrawPixel	PIXPNT
gDrawPixelArea	PIXDRA
gDrawPolygonBound	POLBND
gDrawPolylineBy2D	POLBY2
gDrawPolylineBy3D	POLBY3
gDrawPolylineSet2D	PLSTO2
gDrawPolylineSet3D	PLSTO3
gDrawPolylineTo2D	POLTO2
gDrawPolylineTo3D	POLTO3
gDrawPolymarkerBy2D	SYMBY2
gDrawPolymarkerBy3D	SYMBY3
gDrawPolymarkerTo2D	SYMTO2
gDrawPolymarkerTo3D	SYMTO3
gDrawRect3D	RECT3D
gDrawRuledBezierSurface	RULBEZ
gDrawSeg	PICDRA
gDrawShadedPolyline3D	POSTO3,POTTO3
gDrawSphere	SPHERE
gDrawSplineBy2D	SPLBY2
gDrawSplineBy3D	SPLBY3
gDrawSplineSurface	SPLSUR
gDrawSplineTo2D	SPLTO2
gDrawSplineTo3D	SPLTO3
gDrawSweptBezierSurface	SWEBEZ
gDrawTabulatedBezierSurface	TABBEZ
gDrawVolume	VOLBY3, VOLTO3, VOLUME
gDrawWedge	WEDGE, WEGBY3, WEGTO3
gEditSeg2D	EDITR2
gEditSeg3D	EDITR3
gEndBatchUpdate	MODEND
gEndPolygon	POLEND
gElevateBezier2D	BEZEL2
gElevateBezier3D	BEZEL3
gEnqArcState	ARCENQ

gEnqBrokenLine	BROENQ
gEnqBrokenLineStyle	BRKENQ
gEnqCharAttribs	CHAENQ
gEnqCharTransform	CHATRA
gEnqClippingMode	CLPENQ
gEnqColourInfo	COLINF
gEnqConfigStatus	GINCON
gEnqCursorAction	CURACQ
gEnqCursorType	CURTYQ
gEnqCurveAttribs2D	CURENQ
gEnqCurveAttribs3D	CUREQ3
gEnqDepthMode	DEPENQ
gEnqDeviceState	DEVINF
gEnqDrawingLimits	PAPENQ
gEnqEscapeChar	STRESQ
gEnqFacetFillStyle	FACFIQ
gEnqFacetMaterialProps	FACMAQ
gEnqFacetOffsetMode	FACOSQ
gEnqFog	FOGENQ
gEnqFontStyle	FNTENQ
gEnqGinoState	GINENQ
gEnqHardFontList	FNTLIS
gEnqHatchStyle	HATENQ
gEnqHLS	HLSENQ
gEnqHSV	HSVENQ
gEnqImageFile	IMGENQ
gEnqKeyState	KEYSTA
gEnqLastErrors	ERRENQ
gEnqLightAttribs	LITENQ
gEnqLineColour	COLENQ
gEnqLineEnd	ENDENQ
gEnqLineStyle	LINENQ
gEnqLineVis	VISENQ
gEnqLineWidth	WIDENQ
gEnqLineWidthMode	WIDSWQ
gEnqLineWidthScaling	WIDSCQ

gEnqMaskState	MASENQ
gEnqMaterial	MATENQ
gEnqMaterialAttribs	MATATQ
gEnqMaxDrawingLimits	PAPMAX
gEnqMousePos	MOUENQ
gEnqNumberOfErrors	ERRGET
gEnqOpenSeg	PICNUM
gEnqPenType	TYPENQ
gEnqPicturePos	POSPIC
gEnqPixelAttribs	PIXATQ
gEnqPixelPacking	PIXDEQ
gEnqPixelPos	POSPIX
gEnqPixelResolution	PIXENQ
gEnqPointMode	PNTENQ
gEnqPolygonList	POLLIS
gEnqPolygonMaskList	POMLIS
gEnqPolygonWindowList	POWLIS
gEnqPolygonWorkspace	POLENQ
gEnqPosOfPixel	PIXPOS
gEnqQueueLength	QUEUE
gEnqRGB	RGBENQ
gEnqSavdraDimension	SAVENQ
gEnqSavdraSegAttribs	GETSEQ
gEnqSavdraSegList	GETSNQ
gEnqSegAttribs	PICENQ
gEnqSegGroup	GRPENQ
gEnqSegHit	PICHIT
gEnqSegTransform	PICTRQ
gEnqSegTransform2D	PICTQ2
gEnqSegWorkspace	SDFBFQ
gEnqSelectedPen	PENENQ
gEnqShadingMode	SHAENQ
gEnqSpacePos	POSSPA
gEnqSplineTension	TENENQ
gEnqStrExponent	EXIENQ
gEnqStrJustify	JUSENQ

gEnqStrUnderscore	UNDENQ
gEnqSysArgs	SYSARG
gEnqSysDate	SYSDAT
gEnqSysDateStr	SYSDAS
gEnqSysEnviron	SYSENV
gEnqSysPriority	SYSPRI
gEnqSysTime	SYSTIM
gEnqSysTime	SYSTTM
gEnqSysUsername	SYSNAM
gEnqTextBlockAttribs	TBKENQ
gEnqTextureCoordGeneration	TXMGEQ
gEnqTextureMappingMode	TXMENQ
gEnqTransformState	TRAENQ
gEnqViewport2D	VP2ENQ
gEnqViewport3D	VP3ENQ
gEnqViewportMode	VPTSWQ
gEnqViewportState	VPTENQ
gEnqViewTransformMode	TRASEQ
gEnqWindowState	WINENQ
gEnqWorkingDir	DIRENQ
gEnqWorkspaceLimit	BUFLNQ
gExecuteSysCommand	SYSCOM
gExtendSeg	PICEXT
gFclose	FILCLS
gFillPolygonBy2D	POFBY2
gFillPolygonBy3D	POFBY3
gFillPolygonSet2D	PSFTO2
gFillPolygonSet3D	PSFTO3
gFillPolygonTo2D	POFTO2
gFillPolygonTo3D	POFTO3
gFillRect	RTFILL
gFillSelectedPolygons	GFILL
gFitCharStr	CHAFIT
gFlushGraphics	FLUSHG
gFopen	FILOPN
gGenerateView	VIEW

gGetCursorEvent	CURSOR
gGetDirList	DIRLIS
gGetDrawing	GETDRA
gGetEventRecord	EVEENQ
gGetFullDirList	DIRFUL
gGetImageFile	GETIMG
gGetPicture	GETPIC
gGetPixel	GETPNT
gGetPixelArea	PIXGET
gGetRand	GETRAN
gGetTransform2D	TRASA2
gGetTransform3D	TRASA3
gGetViewParams	VSAVE
gGetViewState	VSAVE
gInitView	VINIT
gInsertSegRef	PICREF
gInsertSegTag	PICTAG
gInterpolateData2D	PT2INT
gInterpolateData3D	PT3INT
gMakeDir	DIRMK
gMarkSeg	PICMAR
gModifyTransform2D	TRAMU2
gModifyTransform3D	TRAMU3
gModifyView	VMULT
gMoveBy2D	MOVBY2
gMoveBy3D	MOVBY3
gMoveSegBy2D	PICBY
gMoveSegTo2D	PICTO
gMoveTo2D	MOVTO2
gMoveTo3D	MOVTO3
gMoveToNextLine	CHANXT
gMoveViewCentre	VALONG
gNewDrawing	SCRCLE
gOpenAuxDrawingArea	AUXOPN
gOpenGino	GINO
gOpenSeg	PICBEG

gPlaySound	GSOUND
gPolygonHit	POLHIT
gPopTransform	TRAEND
gPosViewCentre	VPOSIT
gPushTransform	TRABEG
gReduceBezier2D	BEZRE2
gReduceBezier3D	BEZRE3
gRemoveDir	DIRDEL
gRemoveEventType	EVEDEL
gRemoveFile	FILDEL
gRemoveSegGroup	PICDGP
gRenameFile	FILREN
gRenameSeg	PICREN
gRestoreGinoState	GINRES
gRestoreTransform	TRARES
gRetrieveSegs	SDFRES
gReturnDirDate	DIRDAT
gReturnInternalPoints2D	PT2ENQ
gReturnInternalPoints3D	PT3ENQ
gReturnPlanarNormal	CALNOR
gReturnStrInfo	STRINF
gRotate2D	ROTAT2
gRotate3D	ROTAT3
gSaveGinoState	GINSAV
gSaveLineStyle	LINSAV
gSaveTransform	TRASAV
gScale2D	SCALE2
gScale3D	SCALE3
gSelectDrawingArea	AUXSEL
gSelectPolygons	POLSEL
gSetAlphaMode	CHAMOD
gSetArcIncrement	ARCINC
gSetArcMode	ARCSWI
gSetArcTolerance	ARCTOL
gSetBrokenLine	BROKEN
gSetBrokenLineMode	BRKSWI

gSetCharFont	CHAFNT
gSetCharSize	CHASIZ
gSetCharSizePoint	CHAPNT
gSetCharTransformMode	CHASWI
gSetClippingMode	CLPSWI
gSetColourInfo	COLSET
gSetCursorAction	CURACT
gSetCursorPos	CURPOS
gSetCursorType	CURTYP
gSetCurveAttribs2D	CURSET
gSetCurveAttribs3D	CURST3
gSetDebugSwitch	DEBUGT
gSetDepthMode	DEPMOD
gSetDeviceFilename	DEVFIL
gSetDeviceTitle	DEVTTL
gSetDialogueVis	DIAVIS
gSetDrawingLimits	DEVPAP
gSetErrorFile	ERRDEV
gSetErrorMode	ERRSWI
gSetErrorTrap	ERRSET
gSetEscapeChar	STRESC
gSetFacetFillStyle	FACFIL
gSetFacetMaterialProps	FACMAT
gSetFacetOffsetMode	FACOSM
gSetFillMode	FILSWI
gSetFontFillStyle	FNTSTY
gSetFontForm	FNTREP
gSetFontSpacing	FNTSPA
gSetFontWeight	FNTWGT
gSetGraphicsVis	GRAVIS
gSetHardChars	HARCHA
gSetHardCharSize	CHAHAR
gSetInterlineSpace	CHAILS
gSetItalicAngle	ITALIC
gSetLightSwitch	LITSWI
gSetLineColour	LINCOL

gSetLineEnd	LINEND
gSetLineStyle	LINSEL
gSetLineVis	LINVIS
gSetLineWidth	LINWID
gSetLineWidthMode	WIDSWI
gSetLineWidthScaling	WIDSCA
gSetMask2D	MASK2
gSetMaskMode	MASSWI
gSetMaterialColour	MATCOL
gSetMaterialIndex	SETMAT
gSetMaxErrorLimit	ERRMAX
gSetMixedChars	MIXCHA
gSetMousePos	MOUSET
gSetPenType	PENTYP
gSetPixelDisplayMode	PIXSWI
gSetPixelReplication	PIXREP
gSetPixelTransform	PIXTRA
gSetPointMode	PNTSWI
gSetPolygonIdent	POLIDN
gSetPolygonMask	POLMAS
gSetPolygonMode	POLSWI
gSetPolygonWindow	POLWIN
gSetRandSeed	SETRAN
gSetSegHit	PICSEN
gSetSegMarkColour	SDFMAR
gSetSegMode	SDFSWI
gSetSegTransform	PICTRA
gSetSegTransform2D	PICTR2
gSetSegVis	PICVIS
gSetShadingMode	SHAMOD
gSetSoftChars	SOFCHA
gSetSplineTension	SPLTEN
gSetStrAngle	CHAANG
gSetStrExponent	CHAEXI
gSetStrJustify	CHAJUS
gSetStrUnderscore	CHAUND

gSetSysPriority	SETPRI
gSetTextureCoordGeneration	TXMGEN
gSetTextureMappingMode	TXMMOD
gSetTracerMode	TRACER
gSetTransform	TRANSF
gSetTransform2D	TRASE2
gSetTransform3D	TRASE3
gSetTransformMode	TRAPIC
gSetViewAxis	VIEWSE
gSetViewEyeDistance	VPERSP
gSetViewParams	VRESET
gSetViewState	VRESET
gSetViewPlaneDistance	VZOOM
gSetViewport2D	SETVP2
gSetViewport3D	SETVP3
gSetViewportClipSwitch	VPTCLP
gSetViewportMode	VPTSWI
gSetViewTransformMode	TRASWI
gSetViewUpDirection	VUPDIR
gSetWindow2D	WINDO2
gSetWindow3D	WINDO3
gSetWindowMode	WINDOW, WINSWI
gSetWorkingDir	DIRSET
gSetWorkspaceLimit	BUFLIM
gShear2D	SHEAR2
gShear3D	SHEAR3
gShift2D	SHIFT2
gShift3D	SHIFT3
gStartBatchUpdate	MODBEG
gStartPolygon	POLBEG
gStartTextBlock	CHABEG
gSuspendDevice	DEVSUS
gSwitchBrokenLineStyles	BRKMOD
gTimeDelay	GDELAY
gTransformHomogPoint3D	TRANS4
gTransformPoint2D	TRANS2

gTransformPoint3D	TRANS3
gTrueCol	TRUCOL
gTrueLen	TRULEN
gUntransformHomogPoint3D	UNTRA4
gUntransformPoint2D	UNTRA2
gUntransformPoint3D	UNTRA3
gUpdateView	VIEWUP
gViewRotate	VROTAT
gViewShift	VSHIFT
gViewTurn	VTURN
gWaitForEvent	EVENT

Appendix H

DEPRECATED ROUTINES

Deprecated Routines Introduction

This appendix contains routines that are being deprecated because of the developing nature of GINO as it keeps in line with changes in the graphics and general computing environment.

A routine will go through two intermediate stages prior to being removed from the GINO library:

Stage 0:	Routine has no further use in GINO library. Documentation will be removed from the reference section and temporarily placed in 'DEPRECATED ROUTINES'. The routine will not however be removed from the library.
Stage 1:	Routine will generate a warning message but will function correctly. Documentation will be removed from the reference section and placed in 'DEPRECATED ROUTINES'.
Stage 2:	Routine will generate an error message and will not have any effect on a user's program. Its routine specification and arguments will remain in 'DEPRECATED ROUTINES', but without the description. Alternative functions (where applicable) will be indicated.
Stage 3:	The routine will be removed from the GINO library.

Each stage represents one major release of GINO which gives about 2-3 years in order to facilitate changes to an application program to reflect any deprecations of a routine.

However, it is stressed that no routine will be deprecated without alternatives being provided, unless it is offering a facility that has fallen out of use, and in both cases discussed in the GINO Technical Committee. If there are any problems due to routine deprecation then they should be addressed to the Product Development Manager of **Bradly Associates Limited**.

Due to a rationalisation, the following F77 routines have not been included in the F90 interface due to duplication or lack of use. The following table lists an alternative F90 routine if appropriate.

If users of these routines have a pressing requirement for an equivalent routine in Fortran 90, they should address their case to the Product Development Manager as above.

Fortran-77	**F90 alternative**
ASCII	gDisplayAsciiChar
CENTRE	gShift2D
CHAPOS	gMoveTo2D and gSetAlphaMode
CURSTR	
DASENQ	gEnqBrokenLineStyle
DASHED	gDefineBrokenLineStyle
DATENQ	
DEVATT	
DEVENQ	gEnqDeviceState
DEVICE	gSetDeviceFilename
DEVSPE	
DIAATT	
DIACLE	
DIASTC	
DIASTP	
DOT	gDrawMarker
DRAW2	gMoveBy2D/To2D gDrawLineBy2D/To2D
DRAW3	gMoveBy3D/To3D gDrawLineBy3D/To3D
DSEENQ	
ESCAPE	
ESCIN	
ESCOUT	
GETEND	
IRCBY2	gSetLineVis and gDrawArcBy2D
IRCBY3	gSetLineVis and gDrawArcBy3D
IRCTO2	gSetLineVis and gDrawArcTo2D
IRCTO3	gSetLineVis and gDrdawArcTo3D

Fortran-77	**F90 alternative**
KEYSTR	
LINPEN	
LINPNQ	
NUMENQ	gEnqLineColour
PAPHEA	
PENDEF	
PENEND	
PENNUM	
PENSEL	gSetLineColour
PENSPE	
PICCLE	gNewDrawing and gDeleteSeg
RFILL	gFillRect
SCALE	gScale2D
STRCEN	gDrawMarker

Appendix I

TECHNICAL INFORMATION

Homogeneous Coordinate Transformations

Includes information on:

- 2-D Transformations
- 2-D Matrices
- 2-D Homogeneous Transformations
- Combining Transformations
- 2-D Summary
- Extending 2-D Operations
- Perspective Transformations

2-D Transformations

The 2-D transformations may be described algebraically as follows:

Let (X,Y) be the coordinate of any point in MAN.

Then the transformed coordinates (X',Y') are:

Null transformation

X'=X

Y'=Y

Shifting

X'=X+DX

Y'=Y+DY

Rotating

X'=X*COS(ANGLE)-Y*SIN(ANGLE)

Y'=X*SIN(ANGLE)+Y*COS(ANGLE)

Permutating

X'=Y

Y'=X

Scaling

X'=Sx*X

Y'=Sy*Y

Shearing

X'=X+A*Y
Y'=Y

X'=X
Y'=Y+A*X

2-D Matrices

All these operations can be rewritten in a consistent form if matrix rotation is used:

$$\begin{bmatrix} AB \\ CD \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} A*X+B*Y \\ C*X+D*Y \end{bmatrix} = \begin{bmatrix} X' \\ Y' \end{bmatrix}$$

Then:

Null transformation:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

Rotating:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} \text{COS(ANGLE)} & -\text{SIN(ANGLE)} \\ \text{SIN(ANGLE)} & \text{COS(ANGLE)} \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

Permutating

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

Scaling:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

Shearing:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} 1 & A \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ A & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

3-D Homogeneous Transformations

To incorporate shifts into this simple scheme, the 3-D plane of Z=1 is used as XY plane, and all 2-D operations are treated as 3-D operations on this plane. Thus:

Null transformation:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Rotating:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} COS(ANGLE) & -SIN(ANGLE) & 0 \\ SIN(ANGLE) & COS(ANGLE) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Permutating:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Scaling:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Shearing:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & A & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ A & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Shifting:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & DX \\ 0 & 1 & DY \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Note: This 2-D shift is achieved by means of a 3-D shear.

The third component of the vector is called the homogeneous item. The transformations generated by treating the 2-D plane as the 3-D plane of Z=1 are called homogeneous transformations.

Combining Multiple Transformations

All matrix and vector multiplications are associative. It is not necessary to hold a series of transformations separately - they may be multiplied together and the composite matrix applied to each point.

For example, the result of:

```
gShift2D(10.0,20.0);
gRotate2D(30.0);
gScale2D(1.0,2.0);
gMoveTo2D(x,y);
```

```
call gShift2D(10.0,20.0)
call gRotate2D(30.0)
call gScale2D(1.0,2.0)
call gMoveTo2D(x,y)
```

is:

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 10 \\ 0 & 1 & 20 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 0.0 & 0.0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

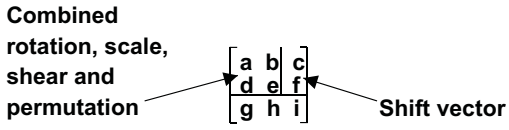
$$= \begin{bmatrix} 0.866 & -1.0 & 10 \\ 0.5 & 1.732 & 20 \\ 0.0 & 0.0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

So the statements above are equivalent to:

gMoveTo2D(0.866*X-Y+10.0,0.5*X+1.732*Y+20.0)

2-D Summary

To summarize, for 2-D transformations, two parts of the 3*3 matrix are used:



Extending 2-D Operations

All the 2-D operations can be simply extended to 3-D by using 4x4 matrices:-

Shifting:

$$\begin{bmatrix} 1 & 0 & 0 & DX \\ 0 & 1 & 0 & DY \\ 0 & 0 & 1 & DZ \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Rotating about the X axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \text{COS(A)} & \text{-SIN(A)} & 0 \\ 0 & \text{SIN(A)} & \text{COS(A)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotating about the Y axis:

$$\begin{bmatrix} \text{COS(A)} & 0 & \text{SIN(A)} & 0 \\ 0 & 1 & 0 & 0 \\ \text{-SIN(A)} & 0 & \text{COS(A)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

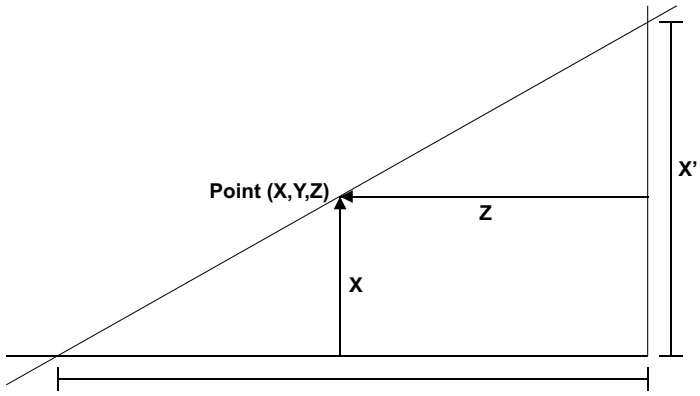
Rotating about the Z axis:

$$\begin{bmatrix} \text{COS(A)} & \text{-SIN(A)} & 0 & 0 \\ \text{SIN(A)} & \text{COS(A)} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling:

$$\begin{bmatrix} \text{Sx} & 0 & 0 & 0 \\ 0 & \text{Sy} & 0 & 0 \\ 0 & 0 & \text{Sz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Transformations



Consider the eye to be at a point S along the Z axis. Let the point (X,Y,Z) be projected on to the point (X',Y') in the Z=0 plane to give 3-D point projection onto this plane.

By similar triangles:

$$\frac{S}{S-Z} = \frac{X'}{X} \quad \text{and} \quad \frac{S}{S-Z} = \frac{Y'}{Y}$$

Thus:

$$X' = \frac{X*S}{S-Z}$$

And

$$Y' = \frac{Y*S}{S-Z}$$

It would be convenient if we could do the same with 4x4 matrix.

Consider:

$$\begin{bmatrix} x^* \\ y^* \\ z^* \\ h^* \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/s & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1-z/s \end{bmatrix}$$

This no longer yields a homogeneous term of 1. In order to restore the homogeneous term to 1, we adopt the convention of dividing through by it, hence:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x^*/h^* \\ y^*/h^* \\ z^*/h^* \\ 1 \end{bmatrix} = \begin{bmatrix} x^*/(1-z) \\ y^*/(1-z) \\ z^*/(1-z) \\ 1 \end{bmatrix}$$

Ignoring the Z' term gives the required point projection. By combining this matrix with rotations, shifts, etc. arbitrary perspective transformations may be generated. In general these will have non zero terms in the bottom row.

Index

!

2D
- devices · 42
- drawing · 77

3D
- devices · 42,271,371
- drawing · 277
- graphics · 269
- objects · 305
- primitives · 307 - 311

A

Adobe Pagemaker - importing GINO files into · · · 62
Adobe Photoshop - importing GINO files into · · · 62
Alphanumeric mode · · · · · · · · · · · · · · · · · · 51
Ambient light · 330
Animation · 49
Application name · · · · · · · · · · · · · · · · · · 463
Archiving segments · · · · · · · · · · · · · · · · · 442
Arcs
- 2D · 86
- 3D · 284 - 287
- chord control · · · · · · · · · · · · · · · · · · · 89

Array transformation · · · · · · · · · · · · · · 375 - 377
ASCII character output · · · · · · · · · · · · · · · 137
Auxiliary Drawing Areas · · · · · · · · · · · · · · 49
Axes
- 2D · 78
- 3D · 278
- swapping · 362

B

Background · 118
Backing store · 49
Bezier
- sphere · 322
- surface · 317
- volume · 323

Bitmaps · 57
Blending · 326
Blending textures · · · · · · · · · · · · · · · · · · 354
BMP · 825
BMP files · 58
Boxes · 307
Building transformations · · · · · · · · · · · 378 - 380

C

Calcomp plotters · · · · · · · · · · · · · · · 799 - 801
Cell array · 191
CGM · 819 - 823
- elements · 820
- errors · · · · · · · · · · · · · · · · · · · 881 - 887
- files · 58

Character
- Attributes · · · · · · · · · · · · · · · · · 147 - 152
- country specific · · · · · · · · · · · · · · · · 160
- default settings · · · · · · · · · · · · · · · · 147
- Escape character · · · · · · · · · · · · · · · 158
- Font enquiry · · · · · · · · · · · · · · · · · · 147
- Font fill style · · · · · · · · · · · · · · · · · · 144
- Font pitch control · · · · · · · · · · · · · · · 146
- Font representation · · · · · · · · · · · · · · 146
- Font weight · · · · · · · · · · · · · · · · · · · 145
- Fonts · · · · · · · · · · · · · · · · 141 - 146,156
- Output · 137
- output ASCII characters · · · · · · · · · · · · 137
- output strings · · · · · · · · · · · · · · · · · 138
- positioning along curve · · · · · · · · · · · · 159
- positioning exponents & indices · · · · · · · 155
- size · 147
- string angle · · · · · · · · · · · · · · · · · · · 149
- string enquiry · · · · · · · · · · · · · · · 159 - 160
- string fitting · · · · · · · · · · · · · · · · · · · 158
- text blocks · 154
- zero representation · · · · · · · · · · · · · · 152

Character transforming · · · · · · · · · · · · 239 - 240
Characters · 135
Circles
- 2-D · 88
- hardware · 89
- number of chords · · · · · · · · · · · · · · · · 89

Clamping · 354
Clipping

mode · 222 - 223
Colour · 205
conversion between systems · · · · · · · · · · 210
coordinate system · · · · · · · · · · · · 207 - 210
definition · 205
dynamic · 47
facet · 299
matching of facets · · · · · · · · · · · · · · · 340
setting and enquiring · · · · · · · · · · · · · · 117
static · 47
Combining transformations
2D · 232 - 237
3D · 364
Command-line · · · · · · · · · · · · · · · · · · · 463
Cones · 309
Configuration file · · · · · · · · · · · · · · · · 26,741
Control arc smoothness · · · · · · · · · · · · · · 89
Converstion of numbers to strings · · · · · · · · · 140
Coordinate systems · · · · · · · · · · · · · · · 36 - 38
Coordinates
2D · 78
3D · 278
homogeneous · · · · · · · · · · · · · · · · · · 369
picture · · · · · · · · · · · · · · · · · · · 238,368
space · 238,368
texture · 349
Corel Ventura - importing GINO files into · · · · · 62
Corel WordPerfect - importing GINO files into · · · 62
CorelDraw - importing GINO files into · · · · · · · 62
Culling · 326
Current point · · · · · · · · · · · · · · · · 79,238,368
Cursor
action types · · · · · · · · · · · · · · · · · · · 243
input · 241
shapes · 242
shapes with GLX driver · · · · · · · · · · · · 750
shapes with X driver · · · · · · · · · · · · · · 795
Curves
akima · 93 - 97
Bezier · · · · · · · · · · · · · · · · 101 - 102,290
end conditions · · · · · · · · · · · · · · · · · · 94
spline · · · · · · · · · · · · · · · · · 98 - 102,290
spline tension · · · · · · · · · · · · · · · · · · 100
Cylinders · 309

D

Dashed Lines · · · · · · · · · · · · · · · · · · 116,124
Debugging · 31
DEC LA100/LN03 · · · · · · · · · · · · · · · · · · 812
Defining colour · · · · · · · · · · · · · · · · · · · 205
Deprecated routines · · · · · · · · · · · · · 927 - 929
Depth-cueing · 337
Device
attributes · 43
defaults · · · · · · · · · · · · · · · · 43,859 - 866
dependent routine · · · · · · · · · · · · · 49 - 51
filename · 44
initialization · · · · · · · · · · · · · · · · · · · 48
nomination · 42
qualification · · · · · · · · · · · · · · · · · 43 - 47
release and suspension · · · · · · · · · · · · · 52
Device Drivers · · · · · · · · · · · · · · · · 739 - 740
Diagnostics · · · · · · · · · · · · · · · · · · · 29 - 32
Dialogue area visibility · · · · · · · · · · · · · · 51
Direct colour · 47
Direction vectors (arcs) · · · · · · · · · · · · · · 288
Directional light · · · · · · · · · · · · · · · · · · 330
Directory enquiry and setting · · · · · · · · · · · 460
DOS VGA/SVGA Driver · · · · · · · · · · 755 - 758
Double Buffering · · · · · · · · · · · · · · · · · · 49
Dragging · 440
Drawing
2D · 77
3D · 277
arcs · 86 - 92
limits · 45
new · 48
units · 44
Drawing Mode · · · · · · · · · · · · · · · · · · · 120
Drawing routine names
2D · 79
3D · 278
DTP packages · 57
DUMMY (device driver) · · · · · · · · · · · · · · 741
DXF · 824
DXF files · 59

E

Environment mapping · · · · · · · · · · · · · · · 353
Environment Variables - system · · · · · · · · · · 464
Error handling · · · · · · · · · · · · · · · · · · · 29
Errors and Warnings · · · · · · · · · · · · · 867 - 880
Escape characters · · · · · · · · · · · · · · · · · 156
Euro symbol · 161
Event
data · 451
programming · · · · · · · · · · · · · · · 454 - 455
Events · 447 - 448
Execute System Command · · · · · · · · · · · · · 465
Exporting · 57
External files · 28
External Images · · · · · · · · · · · · · · · · · · · 73
Eye position · 386

F

F77-F90 Cross Reference · · · · · · · · · · 901 - 912
F90-F77 Cross Reference · · · · · · · · · · 913 - 925
Facet · 295
attributes · · · · · · · · · · · · · · · · · 301 - 303
coloured · 299
face · 296
normal · 297
offset · 302
textured · 299
Field width of numbers · · · · · · · · · · · · · · · · 139
File handling · 460
Fill
hardware/software · · · · · · · · · · · · · · · 171
single polygons · · · · · · · · · · · · · · · · · 257
Filtering textures · · · · · · · · · · · · · · · · · · · 355
Flat shading · 326
Fog · 337 - 338
Font tables · 835
Fonts · 141 - 146

G

gAddEventType Usage · · · · · · · · · · · · · · · · 450
gArchiveSegs Usage · · · · · · · · · · · · · · · · · 442
gBuildMatrix2D Usage · · · · · · · · · · · · · · · · 378
gBuildMatrix3D Usage · · · · · · · · · · · · · · · · 378
gCGMInterpreter Usage · · · · · · · · · · · · · · · · 69
gClearPolygonWorkspace Usage · · · · · · · · · · 250
gClearViewport Usage · · · · · · · · · · · · · · · · 221
gCloseAuxDrawingArea Usage · · · · · · · · · · · · 50
gCloseCGMFile Usage · · · · · · · · · · · · · · · · 70
gCloseDevice Usage · · · · · · · · · · · · · · · · · 52
gCloseGino Usage · · · · · · · · · · · · · · · · · · 26
gCloseSeg Usage · · · · · · · · · · · · · · · · · · 426
gCombineMatrix2D Usage · · · · · · · · · · · · · · 378
gCombineMatrix3D Usage · · · · · · · · · · · · · · 378
gConvertInteger Usage · · · · · · · · · · · · · · · · 140
gConvertRealExponent Usage · · · · · · · · · · · · 140
gConvertRealFixed Usage · · · · · · · · · · · · · · 140
gConvertRealFloat Usage · · · · · · · · · · · · · · 140
gCopyFile Usage · · · · · · · · · · · · · · · · · · · 461
gCopyPixelArea Usage · · · · · · · · · · · · · · · · 203
gCopySeg Usage · · · · · · · · · · · · · · · · · · · 434
gCreateDir Usage · · · · · · · · · · · · · · · · · · · 461
gCreatePlanarShadowMatrix Usage · · · · · · · · · 342
gDebug Usage · 31
gDefineBrokenLineStyle Usage · · · · · · · · · · · 124
gDefineFog Usage · · · · · · · · · · · · · · · · · · 337
gDefineGroupRange Usage · · · · · · · · · · · · · · 438
gDefineHatchStyle Usage · · · · · · · · · · · · · · 172
gDefineHLS Usage · · · · · · · · · · · · · · · · · · 216
gDefineHSV Usage · · · · · · · · · · · · · · · · · · 214
gDefineLightSource Usage · · · · · · · · · · · · · · 329
gDefineLineStyle Usage · · · · · · · · · · · · · · · 129
gDefineMaterial Usage · · · · · · · · · · · · · · · · 340
gDefineNullChar Usage · · · · · · · · · · · · · · · · 152
gDefineParallelView Usage · · · · · · · · · · · · · · 396
gDefinePerspView Usage · · · · · · · · · · · · · · · 393
gDefinePictureUnits Usage · · · · · · · · · · · · · · 44
gDefinePixelPacking Usage · · · · · · · · · · · · · · 194
gDefinePointWorkspace Usage · · · · · · · · · · · · 104
gDefinePolygonWorkspace Usage · · · · · · · 104,245
gDefineRGB Usage · · · · · · · · · · · · · · · · · · 212
gDefineSegGroup Usage · · · · · · · · · · · · · · · 437
gDefineSegWorkspace Usage · · · · · · · · · · · · · 426
gDefineSphericalView Usage · · · · · · · · · · · · · 388
gDefineTexture Usage · · · · · · · · · · · · · · · · 346
gDeleteEventQueue Usage · · · · · · · · · · · · · · 456
gDeleteSeg Usage · · · · · · · · · · · · · · · · · · 427
gDisplayAsciiChar Usage · · · · · · · · · · · · · · · 137
gDisplayInteger Usage · · · · · · · · · · · · · · · · 138
gDisplayRealExponent Usage · · · · · · · · · · · · 138
gDisplayRealFixed Usage · · · · · · · · · · · · · · 138
gDisplayRealFloat Usage · · · · · · · · · · · · · · 138
gDisplayStr Usage · · · · · · · · · · · · · · · · · · 138
gDisplayStrPolylineBy2D Usage · · · · · · · · · · · 159
gDisplayStrPolylineTo2D Usage · · · · · · · · · · · 159
gDragSeg Usage · · · · · · · · · · · · · · · · · · · 440
gDrawAkimaBy2D Usage · · · · · · · · · · · · · · · 93
gDrawAkimaTo2D Usage · · · · · · · · · · · · · · · 93
gDrawArcBy2D Usage · · · · · · · · · · · · · · · · 86
gDrawArcBy3D Usage · · · · · · · · · · · · · · · · 284
gDrawArcTo2D Usage · · · · · · · · · · · · · · · · 86
gDrawArcTo3D Usage · · · · · · · · · · · · · · · · 284
gDrawBezierBy2D Usage · · · · · · · · · · · · · · · 101
gDrawBezierBy3D Usage · · · · · · · · · · · · · · · 290
gDrawBezierSphere Usage · · · · · · · · · · · · · · 323
gDrawBezierSurface Usage · · · · · · · · · · · · · · 317
gDrawBezierTo2D Usage · · · · · · · · · · · · · · · 101
gDrawBezierTo3D Usage · · · · · · · · · · · · · · · 290
gDrawBezierVolume Usage · · · · · · · · · · · · · · 324
gDrawBox Usage · · · · · · · · · · · · · · · · · · · 307
gDrawCellArray Usage · · · · · · · · · · · · · · · · 191
gDrawCone Usage · · · · · · · · · · · · · · · · · · 309
gDrawCube Usage · · · · · · · · · · · · · · · · · · 307
gDrawCurveBy2D Usage · · · · · · · · · · · · · · · 93
gDrawCurveTo2D Usage · · · · · · · · · · · · · · · 93
gDrawCylinder Usage · · · · · · · · · · · · · · · · 309
gDrawFacet Usage · · · · · · · · · · · · · · · · · · 296
gDrawLineBy2D Usage · · · · · · · · · · · · · · · · 80
gDrawLineBy3D Usage · · · · · · · · · · · · · · · 280
gDrawLineTo2D Usage · · · · · · · · · · · · · · · · 80
gDrawLineTo3D Usage · · · · · · · · · · · · · · · 280
gDrawMarker Usage · · · · · · · · · · · · · · · · · 161

gDrawPixel Usage · 191
gDrawPixelArea Usage · · · · · · · · · · · · · · · · · · 191
gDrawPolygonBound Usage · · · · · · · · · · · · · · · 251
gDrawPolylineBy2D Usage · · · · · · · · · · · · · · · · 82
gDrawPolylineBy3D Usage · · · · · · · · · · · · · · · 280
gDrawPolylineSet2D Usage · · · · · · · · · · · · · · · 85
gDrawPolylineSet3D Usage · · · · · · · · · · · · · · 283
gDrawPolylineTo2D Usage · · · · · · · · · · · · · · · 82
gDrawPolylineTo3D Usage · · · · · · · · · · · · · · · 280
gDrawPolymarkerBy2D Usage · · · · · · · · · · · · · 163
gDrawPolymarkerBy3D Usage · · · · · · · · · · · · · 163
gDrawPolymarkerTo2D Usage · · · · · · · · · · · · · 163
gDrawPolymarkerTo3D Usage · · · · · · · · · · · · · 163
gDrawRect3D Usage · · · · · · · · · · · · · · · · · · · 307
gDrawRuledBezierSurface Usage · · · · · · · · · · · 322
gDrawSeg Usage · 433
gDrawShadedPolylineTo3D Usage · · · · · · · · · · 307
gDrawSphere Usage · · · · · · · · · · · · · · · · · · · 310
gDrawSplineBy2D Usage · · · · · · · · · · · · · · · · 98
gDrawSplineBy3D Usage · · · · · · · · · · · · · · · · 288
gDrawSplineSurface Usage · · · · · · · · · · · · · · 313
gDrawSplineTo2D Usage · · · · · · · · · · · · · · · · 98
gDrawSplineTo3D Usage · · · · · · · · · · · · · · · · 288
gDrawSweptBezierSurface Usage · · · · · · · · · · · 321
gDrawTabulatedBezierSurface Usage · · · · · · · · · 320
gDrawVolume Usage · · · · · · · · · · · · · · · · · · · 311
gDrawWedge Usage · · · · · · · · · · · · · · · · · · · 309
gEditSeg2D Usage · 436
gEditSeg3D Usage · 436
gElevateBezier2D Usage · · · · · · · · · · · · · · · · 103
gElevateBezier3D Usage · · · · · · · · · · · · · · · · 290
gEndBatchUpdate Usage · · · · · · · · · · · · · · · · 50
gEndPolygon Usage · · · · · · · · · · · · · · · · · · · 247
gEnqArcState Usage · · · · · · · · · · · · · · · · · · · 91
gEnqBrokenLine Usage · · · · · · · · · · · · · · · · 116
gEnqBrokenLineStyle Usage · · · · · · · · · · · · · 124
gEnqCharAttribs Usage · · · · · · · · · · · · · · · · 151
gEnqCharTransform Usage · · · · · · · · · · · · · · 152
gEnqClippingMode Usage · · · · · · · · · · · · · · · 222
gEnqColourInfo Usage · · · · · · · · · · · · · · · 46,206
gEnqConfigStatus Usage · · · · · · · · · · · · · · · · 26
gEnqCursorAction Usage · · · · · · · · · · · · · · · 243
gEnqCursorType Usage · · · · · · · · · · · · · · · · 242
gEnqCurveAttribs2D Usage · · · · · · · · · · · · · · 100
gEnqCurveAttribs3D Usage · · · · · · · · · · · · · · 289
gEnqDepthMode Usage · · · · · · · · · · · · · · · · 329
gEnqDeviceState Usage · · · · · · · · · · · · · · · · · 43
gEnqDrawingLimits Usage · · · · · · · · · · · · 45,272
gEnqEscapeChar Usage · · · · · · · · · · · · · · · · 158
gEnqFacetFillStyle Usage · · · · · · · · · · · · · · 302
gEnqFacetMaterialProps Usage · · · · · · · · · · · 342
gEnqFacetOffsetMode Usage · · · · · · · · · · · · 303
gEnqFog Usage · 338
gEnqFontStyle Usage · · · · · · · · · · · · · · · · · 147
gEnqGinoState Usage · · · · · · · · · · · · · · · · · · 26
gEnqHardFontList Usage · · · · · · · · · · · · · · · 147
gEnqHatchStyle Usage · · · · · · · · · · · · · · 172,182
gEnqHLS Usage · 216
gEnqHSV Usage · 214
gEnqImageFile Usage · · · · · · · · · · · · · · · · · · 74
gEnqKeyState Usage · · · · · · · · · · · · · · · · · · 457
gEnqLastErrors Usage · · · · · · · · · · · · · · · · · 30
gEnqLightAttribs Usage · · · · · · · · · · · · · · · · 333
gEnqLineColour Usage · · · · · · · · · · · · · · · · · 117
gEnqLineEnd Usage · · · · · · · · · · · · · · · · · · 120
gEnqLineStyle Usage · · · · · · · · · · · · · · · · · 129
gEnqLineVis Usage · · · · · · · · · · · · · · · · · · · 116
gEnqLineWidth Usage · · · · · · · · · · · · · · · · · 119
gEnqLineWidthMode Usage · · · · · · · · · · · · · · 119
gEnqLineWidthScaling Usage · · · · · · · · · · · · 120
gEnqMaskState Usage · · · · · · · · · · · · · · · · · 225
gEnqMaterial Usage · · · · · · · · · · · · · · · · · · 341
gEnqMaterialAttribs Usage · · · · · · · · · · · · · · 341
gEnqMaxDrawingLimits Usage · · · · · · · · · · · · 46
gEnqMousePos Usage · · · · · · · · · · · · · · · · · 456
gEnqNumberOfErrors Usage · · · · · · · · · · · · · 30
gEnqOpenSeg Usage · · · · · · · · · · · · · · · · · · 427
gEnqPenType Usage · · · · · · · · · · · · · · · · · · 120
gEnqPicturePos Usage · · · · · · · · · · · · · 238,368
gEnqPixelAttribs Usage · · · · · · · · · · · · · · · · 202
gEnqPixelPacking Usage · · · · · · · · · · · · · · · 202
gEnqPixelPos Usage · · · · · · · · · · · · · · · · · · 198
gEnqPixelResolution Usage · · · · · · · · · · · · · · 190
gEnqPointMode Usage · · · · · · · · · · · · · · · · · 104
gEnqPolygonList Usage · · · · · · · · · · · · · · · · 257
gEnqPolygonMaskList Usage · · · · · · · · · · · · · 266
gEnqPolygonWindowList Usage · · · · · · · · · · · 266
gEnqPolygonWorkspace Usage · · · · · · · · · · · 250
gEnqPosOfPixel Usage · · · · · · · · · · · · · · · · · 198
gEnqQueueLength Usage · · · · · · · · · · · · · · · 456
gEnqRGB Usage · 212
gEnqSavdraDimension Usage · · · · · · · · · · · · · 64
gEnqSavdraSegAttribs Usage · · · · · · · · · · · · · 65
gEnqSavdraSegList Usage · · · · · · · · · · · · · · · 65
gEnqSegAttribs Usage · · · · · · · · · · · · · · · · · 432
gEnqSegGroup Usage · · · · · · · · · · · · · · · · · 438
gEnqSegHit Usage · · · · · · · · · · · · · · · · · · · 439
gEnqSegTransform Usage · · · · · · · · · · · · · · · 432
gEnqSegTransform2D Usage · · · · · · · · · · · · · 432
gEnqSegWorkspace Usage · · · · · · · · · · · · · · 426
gEnqSelectedPen Usage · · · · · · · · · · · · · · · · 113
gEnqShadingMode Usage · · · · · · · · · · · · · · · 327
gEnqSpacePos Usage · · · · · · · · · · · · · · 238,368
gEnqSplineTension Usage · · · · · · · · · · · · · · · 100
gEnqStrExponent Usage · · · · · · · · · · · · · · · · 155
gEnqStrJustify Usage · · · · · · · · · · · · · · · · · 153
gEnqStrUnderscore Usage · · · · · · · · · · · · · · · 152
gEnqSysArgs Usage · · · · · · · · · · · · · · · · · · 463

gEnqSysDate Usage · · · · · · · · · · · · · · · · · · 463
gEnqSysDateStr Usage · · · · · · · · · · · · · · · · · 463
gEnqSysEnviron Usage · · · · · · · · · · · · · · · · · 464
gEnqSysPriority Usage · · · · · · · · · · · · · · · · · 466
gEnqSysTime Usage · · · · · · · · · · · · · · · · · · · 463
gEnqSysUsername Usage · · · · · · · · · · · · · · · · 464
gEnqTextBlockAttribs Usage · · · · · · · · · · · · · · 155
gEnqTextureCoordGeneration Usage · · · · · · · · · · 351
gEnqTextureMappingMode Usage · · · · · · · · · · · · 358
gEnqTransformState Usage · · · · · · · · · · · · · · · 381
gEnqViewport2D Usage · · · · · · · · · · · · · · · · · 221
gEnqViewport3D Usage · · · · · · · · · · · · · · · · · 274
gEnqViewportMode Usage · · · · · · · · · · · · · · · · 220
gEnqViewportState Usage · · · · · · · · · · · · · · · · 221
gEnqViewTransformMode Usage · · · · · · · · · · · · 371
gEnqWindowState Usage · · · · · · · · · · · · · 224,275
gEnqWorkingDir Usage · · · · · · · · · · · · · · · · · 460
gEnqWorkspaceLimit Usage · · · · · · · · · · · · · · · 36
gExecuteSysCommand Usage · · · · · · · · · · · · · · 465
gExtendSeg Usage · · · · · · · · · · · · · · · · · · · 427
gFclose Usage · 28
gFillPolygonBy2D Usage · · · · · · · · · · · · · · · · 167
gFillPolygonBy3D Usage · · · · · · · · · · · · · · · · 290
gFillPolygonSet2D Usage · · · · · · · · · · · · · · · · 170
gFillPolygonSet3D Usage · · · · · · · · · · · · · · · · 290
gFillPolygonTo2D Usage · · · · · · · · · · · · · · · · 167
gFillPolygonTo3D Usage · · · · · · · · · · · · · · · · 290
gFillRect Usage · 165
gFillSelectedPolygons Usage · · · · · · · · · · · · · · 257
gFitCharStr Usage · · · · · · · · · · · · · · · · · · · 158
gFlushGraphics Usage · · · · · · · · · · · · · · · · · · 49
gFopen Usage · 28
gGenerateView Usage · · · · · · · · · · · · · · · · · 399
gGetCGMElement Usage · · · · · · · · · · · · · · · · · 70
gGetCursorEvent Usage · · · · · · · · · · · · · · · · 241
gGetDirList Usage · · · · · · · · · · · · · · · · · · · 461
gGetDrawing Usage · · · · · · · · · · · · · · · · · · · 65
gGetEventRecord Usage · · · · · · · · · · · · · · · · 451
gGetFullDirList Usage · · · · · · · · · · · · · · · · · 461
gGetImageFile Usage · · · · · · · · · · · · · · · 74,346
gGetPicture Usage · · · · · · · · · · · · · · · · · · · 67
gGetPixel Usage · 191
gGetPixelArea Usage · · · · · · · · · · · · · · · · · · 203
gGetRand Usage · 466
gGetTransform2D Usage · · · · · · · · · · · · · · · · 377
gGetTransform3D Usage · · · · · · · · · · · · · · · · 377
gGetViewParams Usage · · · · · · · · · · · · · · · · · 416
gGetViewState Usage · · · · · · · · · · · · · · · · · · 416
gInitView Usage · 399
GINO
  Closing · 26
  facilities · 24
  general description · · · · · · · · · · · · · · · · 23
  State · 54
  States · 26 - 27
  structures · · · · · · · · · · · · · · · · · 891 - 899
gInsertSegRef Usage · · · · · · · · · · · · · · · · · · 435
gInsertSegTag Usage · · · · · · · · · · · · · · · · · · 436
gInterpolateData2D Usage · · · · · · · · · · · · · · · 107
gInterpolateData3D Usage · · · · · · · · · · · · · · · 294
gInterpretCGMElement Usage · · · · · · · · · · · · · · 70
GLX · 743
gMarkSeg Usage · 431
gModify View Usage · · · · · · · · · · · · · · · · · · 417
gModifyTransform2D Usage · · · · · · · · · · · · · · · 377
gModifyTransform3D Usage · · · · · · · · · · · · · · · 377
gModifyView Usage · · · · · · · · · · · · · · · · · · · 342
gMoveBy2D Usage · 80
gMoveBy3D Usage · · · · · · · · · · · · · · · · · · · 279
gMoveSegBy2D Usage · · · · · · · · · · · · · · · · · · 431
gMoveSegTo2D Usage · · · · · · · · · · · · · · · · · · 431
gMoveTo2D Usage · 80
gMoveTo3D Usage · · · · · · · · · · · · · · · · · · · 279
gMoveToNextLine Usage · · · · · · · · · · · · · · · · 154
gMoveViewCentre Usage · · · · · · · · · · · · · · · · 404
gNewDrawing Usage · · · · · · · · · · · · · · · · · · · 48
gOpenAuxDrawingArea Usage · · · · · · · · · · · · · · 49
gOpenCGMFile Usage · · · · · · · · · · · · · · · · · · 70
gOpenGino Usage · 25
gOpenSeg Usage · 426
Gouraud shading · 326
gPlaySound Usage · · · · · · · · · · · · · · · · · · · 466
gPolygonHit Usage · · · · · · · · · · · · · · · · · · · 261
gPopTransform Usage · · · · · · · · · · · · · · · · · 376
gPosViewCentre Usage · · · · · · · · · · · · · · · · · 402
gPrintf Usage · 138
gPushTransform Usage · · · · · · · · · · · · · · · · · 376
Graphics buffer · 49
gReduceBezier2D Usage · · · · · · · · · · · · · · · · 103
gRemoveDir Usage · · · · · · · · · · · · · · · · · · · 461
gRemoveEventType Usage · · · · · · · · · · · · · · · 450
gRemoveFile Usage · · · · · · · · · · · · · · · · · · · 461
gRemoveSegGroup Usage · · · · · · · · · · · · · · · · 437
gRenameFile Usage · · · · · · · · · · · · · · · · · · · 461
gRenameSeg Usage · · · · · · · · · · · · · · · · · · · 427
gRestoreGinoState Usage · · · · · · · · · · · · · · · · 54
gRestoreTransform Usage · · · · · · · · · · · · · · · · 376
gRetrieveSegs Usage · · · · · · · · · · · · · · · · · · 442
gReturnDirDate Usage · · · · · · · · · · · · · · · · · 463
gReturnInternalPoints2D Usage · · · · · · · · · · · · 104
gReturnInternalPoints3D Usage · · · · · · · · · · · · 293
gReturnPlanarNormal Usage · · · · · · · · · · · · · · 297
gReturnStrInfo Usage · · · · · · · · · · · · · · · · · 159
gRotate2D Usage · 228
gRotate3D Usage · 360
gSaveGinoState Usage · · · · · · · · · · · · · · · · · · 54
gSaveLineStyle Usage · · · · · · · · · · · · · · · · · 131
gSaveTransform Usage · · · · · · · · · · · · · · · · · 376

gScale2D Usage · 229
gScale3D Usage · 363
gSelectDrawingArea Usage · · · · · · · · · · · · · · · 50
gSelectPolygons · 256
gSelectPolygons Usage · · · · · · · · · · · · · · · · · 254
gSetAlphaMode Usage · · · · · · · · · · · · · · · · · · 51
gSetArcIncrement Usage · · · · · · · · · · · · · · · · 89
gSetArcMode Usage · · · · · · · · · · · · · · · · · · · 89
gSetArcTolerance Usage · · · · · · · · · · · · · · · · 89
gSetBrokenLine Usage · · · · · · · · · · · · · · · · · 116
gSetBrokenLineMode Usage · · · · · · · · · · · · · · 116
gSetCharFont Usage · · · · · · · · · · · · · · · · · · 141
gSetCharSize Usage · · · · · · · · · · · · · · · · · · 147
gSetCharSizePoint Usage · · · · · · · · · · · · · · · 149
gSetCharTransformMode Usage · · · · · · · · · · · · 239
gSetClippingMode Usage · · · · · · · · · · · · · · · 222
gSetColourInfo Usage · · · · · · · · · · · · · · · · · 47
gSetCursorAction Usage · · · · · · · · · · · · · · · · 243
gSetCursorPos Usage · · · · · · · · · · · · · · · · · · 242
gSetCursorType Usage · · · · · · · · · · · · · · · · · 242
gSetCurveAttribs2D Usage · · · · · · · · · · · · 94,100
gSetCurveAttribs3D Usage · · · · · · · · · · · · · · 289
gSetDebugSwitch Usage · · · · · · · · · · · · · · · · 32
gSetDepthMode Usage · · · · · · · · · · · · · · · · · 328
gSetDeviceFilename Usage · · · · · · · · · · · · · · 44
gSetDeviceTitle Usage · · · · · · · · · · · · · · · · · 52
gSetDialogueVis Usage · · · · · · · · · · · · · · · · 51
gSetDrawingLimits Usage · · · · · · · · · · · · · · · 45
gSetErrorFile Usage · · · · · · · · · · · · · · · · · · 31
gSetErrorMode Usage · · · · · · · · · · · · · · · · · 30
gSetErrorTrap Usage · · · · · · · · · · · · · · · · · · 30
gSetEscapeChar Usage · · · · · · · · · · · · · · · · · 158
gSetFacetFillStyle Usage · · · · · · · · · · · · · · · 301
gSetFacetMaterialProps Usage · · · · · · · · · · · · 342
gSetFacetOffsetMode Usage · · · · · · · · · · · · · 302
gSetFillMode Usage · · · · · · · · · · · · · · · · · · 171
gSetFontFillStyle Usage · · · · · · · · · · · · · · · 144
gSetFontForm Usage · · · · · · · · · · · · · · · · · · 146
gSetFontSpacing Usage · · · · · · · · · · · · · · · · 146
gSetFontWeight Usage · · · · · · · · · · · · · · · · · 145
gSetGraphicsVis Usage · · · · · · · · · · · · · · · · 51
gSetInterlineSpace Usage · · · · · · · · · · · · · · · 154
gSetItalicAngle Usage · · · · · · · · · · · · · · · · · 150
gSetLightSwitch Usage · · · · · · · · · · · · · · · · · 332
gSetLineColour Usage · · · · · · · · · · · · · · · · · 117
gSetLineEnd Usage · · · · · · · · · · · · · · · · · · · 120
gSetLineStyle Usage · · · · · · · · · · · · · · · · · · 131
gSetLineVis Usage · · · · · · · · · · · · · · · · · · · 116
gSetLineWidth Usage · · · · · · · · · · · · · · · · · · 119
gSetLineWidthMode Usage · · · · · · · · · · · · · · 119
gSetLineWidthScaling Usage · · · · · · · · · · · · · 119
gSetMask2D Usage · · · · · · · · · · · · · · · · · · · 224
gSetMaskMode Usage · · · · · · · · · · · · · · · · · 225
gSetMaterialColour Usage · · · · · · · · · · · · · · · 341
gSetMaterialIndex Usage · · · · · · · · · · · · · · · 341
gSetMaxErrorLimit Usage · · · · · · · · · · · · · · · 30
gSetMousePos Usage · · · · · · · · · · · · · · · · · · 456
gSetPenType Usage · · · · · · · · · · · · · · · · · · · 120
gSetPixelDisplayMode Usage · · · · · · · · · · · · · 197
gSetPixelReplication Usage · · · · · · · · · · · · · · 201
gSetPixelTransform Usage · · · · · · · · · · · · · · · 198
gSetPointMode Usage · · · · · · · · · · · · · · · · · 104
gSetPolygonIdent Usage · · · · · · · · · · · · · · · · 249
gSetPolygonMask Usage · · · · · · · · · · · · · · · · 265
gSetPolygonMode Usage · · · · · · · · · · · · · · · · 247
gSetPolygonWindow Usage · · · · · · · · · · · · · · 264
gSetRandSeed Usage · · · · · · · · · · · · · · · · · · 466
gSetSegHit Usage · 431
gSetSegMarkColour Usage · · · · · · · · · · · · · · · 431
gSetSegMode Usage · · · · · · · · · · · · · · · · · · 424
gSetSegTransform Usage · · · · · · · · · · · · · · · 431
gSetSegTransform2D Usage · · · · · · · · · · · · · · 431
gSetSegVis Usage · · · · · · · · · · · · · · · · · · · 430
gSetShadingMode Usage · · · · · · · · · · · · · · · · 325
gSetSplineTension Usage · · · · · · · · · · · · · · · 100
gSetStrAngle Usage · · · · · · · · · · · · · · · · · · 149
gSetStrExponent Usage · · · · · · · · · · · · · · · · 155
gSetStrJustify Usage · · · · · · · · · · · · · · · · · 153
gSetStrUnderscore Usage · · · · · · · · · · · · · · · 152
gSetSysPriority Usage · · · · · · · · · · · · · · · · · 466
gSetTextureCoordGeneration Usage · · · · · · · · · 351
gSetTextureMappingMode Usage · · · · · · · 345,354
gSetTracerMode Usage · · · · · · · · · · · · · · · · 31
gSetTransform Usage · · · · · · · · · · · · · · · · · 372
gSetTransform2D Usage · · · · · · · · · · · · · · · · 377
gSetTransform3D Usage · · · · · · · · · · · · · · · · 377
gSetTransformMode Usage · · · · · · · · · · 239,382
gSetViewAxis Usage · · · · · · · · · · · · · · · · · · 362
gSetViewEyeDistance Usage · · · · · · · · · · · · · 404
gSetViewParams Usage · · · · · · · · · · · · · · · · 416
gSetViewPlaneDistance Usage · · · · · · · · · · · · 404
gSetViewport2D Usage · · · · · · · · · · · · · · · · · 219
gSetViewport3D Usage · · · · · · · · · · · · · · · · · 273
gSetViewportClipSwitch Usage · · · · · · · · · · · · 221
gSetViewportMode Usage · · · · · · · · · · · 220,274
gSetViewState Usage · · · · · · · · · · · · · · · · · · 416
gSetViewTransformMode Usage · · · · · · · · · · · 371
gSetViewUpDirection Usage · · · · · · · · · · · · · 402
gSetWindow2D Usage · · · · · · · · · · · · · · · · · 223
gSetWindow3D Usage · · · · · · · · · · · · · · · · · 275
gSetWindowMode Usage · · · · · · · · · · · · · · · · 222
gSetWorkingDir Usage · · · · · · · · · · · · · · · · · 460
gSetWorkspaceLimit Usage · · · · · · · · · · · · · · 33
gShear2D Usage · 231
gShear3D Usage · 364
gShift2D Usage · 228
gShift3D Usage · 360
gSkipCGMElement Usage · · · · · · · · · · · · · · · 70

gStartBatchUpdate Usage · · · · · · · · · · · · · · · 50
gStartPolygon Usage · · · · · · · · · · · · · · · · 247
gStartTextBlock Usage · · · · · · · · · · · · · · · 154
gSuspendDevice Usage · · · · · · · · · · · · · · · 52
gSwitchBrokenLineStyles Usage · · · · · · · · · · 128
gTimeDelay Usage · · · · · · · · · · · · · · · · · 463
gTransformHomogPoint3D Usage · · · · · · · · · · 369
gTransformPoint2D Usage · · · · · · · · · · · · · 238
gTransformPoint3D Usage · · · · · · · · · · · · · 369
gTrueCol Usage · · · · · · · · · · · · · · · · · · 217
gTrueLen Usage · · · · · · · · · · · · · · · · · · 467
gUntransformHomogPoint3D Usage · · · · · · · · · 368
gUntransformPoint2D Usage · · · · · · · · · · · · 238
gUntransformPoint3D Usage · · · · · · · · · · · · 368
gUpdateView Usage · · · · · · · · · · · · · · · · 388
gViewRotate Usage · · · · · · · · · · · · · · · · 411
gViewShift Usage · · · · · · · · · · · · · · · · · 410
gViewTurn Usage · · · · · · · · · · · · · · · · · 410
gWaitForEvent Usage · · · · · · · · · · · · · · · 451

H

Hardware
  arcs · 89
  transformations · · · · · · · · · · · · · · · · 371
Hatch styles - defining · · · · · · · · · · · · 172 - 182
Hewlett-Packard plotters
  HPGL · 802
  HPGL/2 · 806
Hewlett-Packard printers
  laserjet · · · · · · · · · · · · · · · · · · · 809
  paintjet and deskjet · · · · · · · · · · · · · 810
Hidden Surface Removal · · · · · · · · · · · · · 327
Hierarchical Segments · · · · · · · · · · · · · · 434
Highlighting segments · · · · · · · · · · · · · · 431
HLS colour system · · · · · · · · · · · · · 215 - 216
Homogeneous coordinates · · · · · · · · · · · · 369
HSV colour system · · · · · · · · · · · · · 212 - 214

I

ICO files · 60
Image handling · · · · · · · · · · · · · · · · · 189
  display · · · · · · · · · · · · · · · · · 191 - 197
  hiding · · · · · · · · · · · · · · · · · · · 197
  reading · · · · · · · · · · · · · · · · · · · 203
  replication · · · · · · · · · · · · · · · · · 201
Importing · 57
Initializing GINO · · · · · · · · · · · · · · · · 25
Input
  device · 454
Interaction
  advanced · · · · · · · · · · · · · · · · · · · 447
  basic · 241
Interpolation
  2D · 107
  3D · 294
Italic Characters · · · · · · · · · · · · · · · · 150

J

Joining of lines · · · · · · · · · · · · · · · · · 120
JPEG · · · · · · · · · · · · · · · · · · · 827 - 828
JPEG files · · · · · · · · · · · · · · · · · · · 60
Justification of character strings · · · · · · · · · 153

K

Key values from keyboard and mouse · · · · 452 - 453
Keyboard state · · · · · · · · · · · · · · · 457 - 458

L

Light pen simulation · · · · · · · · · · · · · · · 439
Light switch · · · · · · · · · · · · · · · · · · 332
Lighting · · · · · · · · · · · · · · · · · · 329 - 336
Line
  attributes · · · · · · · · · · · · · · · · · · 111
  attributes affecting characters · · · · · · · · 152
  colour · 117
  current · · · · · · · · · · · · · 112,114,130 - 131
  style table · · · · · · · · · · · · · · · · · 866
  styles · · · · · · · · · · · · · · · · · 124 - 133
Line of sight · · · · · · · · · · · · · · · 386,410 - 414
Lines
  2D · 80 - 81
  3D · 280
Lotus 1-2-3 - importing GINO files into · · · · · · 62
Lotus Freelance - importing GINO files into · · · · 62
Lotus WordPro - importing GINO files into · · · · 62

M

Machine independence of GINO · · · · · · · · · · 23
Mapping - viewport · · · · · · · · · · · · · 219 - 221
Markers · · · · · · · · · · · · · · · · · · · 161 - 163
Marking segments · · · · · · · · · · · · · · · · 431
Masking
  enquiry · · · · · · · · · · · · · · · · · · · 225
  polygonal · · · · · · · · · · · · · · · · · · 265
  rectangular · · · · · · · · · · · · · · · 224 - 225
Material properties · · · · · · · · · · · · · · · 339
Messages - errors and warnings · · · · · · · · · · 29
Metafile
  drivers · · · · · · · · · · · · · · · · · · · 818

formats · 58 - 61
Microsoft Excel - importing GINO files into · · · · 62
Microsoft IE5 - importing GINO files into · · · · · 62
Microsoft PowerPoint - importing GINO files into · 62
Microsoft Word - importing GINO files into · · · · 62
Mipmapped textures · · · · · · · · · · · · · · · · 348
Modelling · 359
Modify transformations · · · · · · · · · · · · · · 377
Mouse
pointer shapes with GLX driver · · · · · · · 750
pointer shapes with X driver · · · · · · · · · 795
Mouse position · · · · · · · · · · · · · · · · · · 456
Multiple Devices · · · · · · · · · · · · · · · · 52 - 56

N

Netscape - importing GINO files into · · · · · · · · 62
Normals · 297
Notional device · · · · · · · · · · · · · · · · · · 40
Numerical output · · · · · · · · · · · · · · · 138 - 140

O

Object
axes system · · · · · · · · · · · · · · · · · · 306
complexity · · · · · · · · · · · · · · · · · · 306
shading · 306
Oblique projection · · · · · · · · · · · · · · · · 415
OpenGL
features · · · · · · · · · · · · · · · · · 269 - 271
performance · · · · · · · · · · · · · · · · · · 271
X-Windows driver · · · · · · · · · · · · · · · 743
OpenVMS specifics · · · · · · · · · · · · · · · · 737

P

Paintshop Pro - importing GINO files into · · · · · 62
Paper size and type · · · · · · · · · · · · · · · · 45
Parallel projection · · · · · · · · · · · · · 396 - 397
Pen · 77
position · · · · · · · · · · · · · · · · · 80,279
Permutating axes · · · · · · · · · · · · · · · · · 362
Perspective view · · · · · · · · · · · · · · 388 - 392
Phong shading · · · · · · · · · · · · · · · · · · 326
Picture coordinates
2D · 238
3D · 368
Picture mode · · · · · · · · · · · · · · · · · · · 382
Picture segments · · · · · · · · · · · · · · · · · 423
Pixel · 189
coordinates · · · · · · · · · · · · · · · · · · 190
data definition · · · · · · · · · · · · · · · · 194
enquiry / resolution · · · · · · · · · · · · · 202
single pixel reading / writing · · · · · · · · · 191
Planar normal · · · · · · · · · · · · · · · · · · · 297
Plotter devices · · · · · · · · · · · · · · · 41,797 - 798
PNG · 829
PNG files · 60
Point light source · · · · · · · · · · · · · · · · · 330
Point storage
2D · 103
3D · 293
Polygon
3D · 290 - 292
area filling · · · · · · · · · · · · · · · · 169 - 170
complex area filling · · · · · · · · · · · · · 257
drawing · · · · · · · · · · · · · · · 247 - 249,251
enquiry · 250
identification · · · · · · · · · · · · · · · · · 249
interaction · · · · · · · · · · · · · · · · · · 261
selection · · · · · · · · · · · · · · · · 254 - 256
simple area filling · · · · · · · · · · · · · · 165
vertices · · · · · · · · · · · · · · · · · 247 - 249
workspace · · · · · · · · · · · · · 250,252 - 253
Polyline set
2D · 84 - 85
3D · 282 - 283
Polylines
2D · 82 - 83
3D · 280 - 281
shaded · 307
Position
2D · 80
3D · 279
segment · · · · · · · · · · · · · · · · · · 430 - 431
POSTSCRIPT · · · · · · · · · · · · · · · · 813 - 817
POSTSCRIPT files · · · · · · · · · · · · · · · · · 59
Printer devices · · · · · · · · · · · · · · 41,797 - 798
Priority of task · · · · · · · · · · · · · · · · · · 466

Q

Quark Xpress - importing GINO files into · · · · · 62
Queues · 456

R

Random number generation · · · · · · · · · · · · · 466
Rectangular area fill · · · · · · · · · · · · · 165 - 166
Reflection · 230
Refresh displays · · · · · · · · · · · · · · · 423 - 425
REGIS · 752 - 754
RGB colour system · · · · · · · · · · · · · · · · 211
Rotation
2D · 228
3D · 360

Ruled surface · 321

S

SAVDRA · 830 - 832
SAVDRA files · 60
Scaling
2D · 229
3D · 363
Screen devices · · · · · · · · · · · · · · · · · · · 41,742
SDF · 424
Segment
anchor · 427
body · 430
building · · · · · · · · · · · · · · · · · · · 426 - 429
copying · 434
enquiry · 432
groups · 437
hierarchies · · · · · · · · · · · · · · · · · · · 434
modelling transformations · · · · · · · · · · · 435
redrawing · 433
structures · · · · · · · · · · · · · · · · · · 434 - 438
transformation · · · · · · · · · · · · · · · · · 431
Sensitivity of segments · · · · · · · · · · · · · · · 431
Shading · 270,325
Shadows · 342 - 343
Shearing
2D · 231
3D · 364
Shifting
2D · 227
3D · 360
Smooth shading · · · · · · · · · · · · · · · · · · · 326
Smoothness of arcs · · · · · · · · · · · · · · · · · · 89
Software Characters · · · · · · · · · · · · · · · · · 136
Software Display File
archive · 442
hard copy · 440
Sound · 466
Space
axes · 227
mode · 382
Space coordinates
2D · 238
3D · 368
Specular light · 332
Sphere
Bezier · 322
faceted · 310
Spline Curves · · · · · · · · · · · · · · · · · · 98 - 100
3D · 288 - 289
end conditions · · · · · · · · · · · · · · · · · · 99
Spline surface · 313
Spot light · 331
Storage tube · 243
Straight lines
2D · 80 - 81
3D · 280
SUN raster files · · · · · · · · · · · · · · · · · · · 825
Surface primitive · · · · · · · · · · · · · · · · 312 - 324
Swept surface · 320
System errors · 888
System Utilities · · · · · · · · · · · · · · · · · · · 459

T

Tables
colour · 113
hatch style · 172
line definition · · · · · · · · · · · · · · · · · · 129
Tabulated surface · · · · · · · · · · · · · · · · · · 320
Texel · 355
Text output · 135
Texture coordinates · · · · · · · · · · · · · · · · · 349
Texture mapping · · · · · · · · · · · · · · · · · · · 345
Textured facet · 299
Tiling images · 349
Time Delay · 463
Time enquiry · 463
Titling · 52
Tolerance
arc · 91
Tolerance of arcs · · · · · · · · · · · · · · · · · · · 90
Trace facilities · 31
Transformation
enquiry · 381
initializing · 372
matrix · 375
mode · 239,382
state · 372 - 374
Transformation control · · · · · · · · · · · · · · · · 371
Transformations
2D · 227
3D · 359
characters · 239
pixel · 198 - 201
Translucence · 342
Transparent · 326
Trapping error messages · · · · · · · · · · · · · · · 30

U

Underlining characters · · · · · · · · · · · · · · · · 152
UNIX specifics · 736
Untransforming
2D · 238

3D · 368
User name · 464

V

View plane · 386
View transform mode · · · · · · · · · · · · · · · · · 371
Viewing · 385
modifications · · · · · · · · · · · · · · · · 400 - 403
modify matrix · · · · · · · · · · · · · · · · · · · 417
state · 416
transformations · · · · · · · · · · · · · · · 398 - 399
Viewport
2D · 219 - 221
3D · 273 - 274
Visibility
line · 116
segments · 430
Visual Basic
calling GINO from · · · · · · · · · · · · · · · · 776
importing GINO files into · · · · · · · · · · · · 62
Volume
faceted · 311

W

Warning messages · · · · · · · · · · · · · · · · · · 29
WEB Browsers · 57
Wedges · 309
Width of lines · 119
Winding rule · 327
Window visibility · · · · · · · · · · · · · · · · · · 51
Windowing
2D polygonal · · · · · · · · · · · · · · · 261 - 267
3D polygonal · · · · · · · · · · · · · · · · · · 275
enquiry · · · · · · · · · · · · · · · · · · 224,275
mode · 222
rectangular · · · · · · · · · · · · · · · · · · · 223
switching · 222
Windows driver (MWIN) · · · · · · · · · · · 759 - 777
Windows OpenGL Driver · · · · · · · · · · 778 - 789
Windows Programming · · · · · · · · · · · · · · · 769
Windows specifics · · · · · · · · · · · · · · · · · 738
WMF · 833 - 834
WMF files · 61
Workspace
clearing · 250
enquiry · 36
management · · · · · · · · · · · · · · · · · 33 - 35
polygon · · · · · · · · · · · · · · · 245,252 - 253
Workstation devices · · · · · · · · · · · · · · · · 41
Workstations · 742
WP packages · 57

X

X Windows · · · · · · · · · · · · · · · · · · 790 - 796
XWD Driver · 825
XWD files · 61

Z

Z Buffering · 327
Zooming · 404