GDDM

Base Application Programming Guide

Version 3 Release 2

SC33-0867-01

GDDM

Base Application Programming Guide

Version 3 Release 2

SC33-0867-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xxiii.

Second Edition (September 1996)

This edition applies to Version 3 Release 2 of the IBM GDDM series of licensed programs. The programs and their numbers are:

GDDM/MVS	5695-167
GDDM/VM	5684-168
GDDM/VSE	5686-057

It also applies to GDDM/MVS as an element of OS/390 (program number 5645-001).

Publications are not stocked at the addresses given below. Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers’ comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM United Kingdom Laboratories,
Information Development, Mail Point 095,
Hursley Park, Winchester, Hampshire, England SO21 2JN

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

No part of this book may be reproduced in any form or by any means, including storing in a data processing machine, without permission in writing from IBM.

© Copyright International Business Machines Corporation 1982, 1996. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . xxiii
Trademarks and service marks . . . xxiv

Preface . . . xxv
What this book is about . . . xxv
Who this book is for . . . xxv
How to use this book . . . xxv
Latest GDDM information . . . xxvi
GDDM publications . . . xxvii
Books from related libraries . . . xxviii

Summary of changes . . . xxix
Changes to this book for Version 3 Release 2 . . . xxix
Changes to this book for Version 3 Release 1 . . . xxix

Part 1 GDDM basics . . . 1

Chapter 1. An introduction to programming with GDDM . . . 3
Supported languages . . . 3
The GDDM external application programming interfaces . . . 4
The nonreentrant programming interface . . . 4
The reentrant programming interface . . . 4
The system programmer interface . . . 5
Example programs . . . 5
Example: The HOUSE program . . . 6
Concepts introduced by the HOUSE program . . . 7
Initialization of GDDM . . . 7
Termination of GDDM . . . 8
Graphics primitives . . . 8
Graphics attributes . . . 8
The graphics segment . . . 8
The current position . . . 8
Graphics text . . . 8
The GDDM page . . . 9
Sending output from a GDDM program to a device . . . 9
Learning interactively to program with GDDM . . . 10
Error handling . . . 10
Entry points to GDDM . . . 11
Data types of GDDM call parameters . . . 11
How to compile and run a PL/I GDDM program . . . 14
How to compile and run the HOUSE program under VM/CMS . . . 14
How to compile, link-edit, and run the HOUSE program under TSO . . . 14
GDDM-REXX—the fast path to programming with GDDM . . . 15
Prototyping your solutions quickly . . . 15
An example of a REXX program using GDDM's functions . . . 16
Points to remember when using GDDM-REXX . . . 16
Converting PL/I examples to GDDM-REXX . . . 18
Specifying call parameters made easy by GDDM-REXX . . . 18
More complex programming with GDDM-REXX . . . 20

© Copyright IBM Corp. 1982, 1996

Multiple instances of GDDM and GDDM-REXX . . . 21
Invoking GDDM-REXX programs from other programs or from CMS subset . 23
Coding styles—strict or loose syntax . . . 23

Chapter 2. Drawing graphics pictures . . . 25
Example: Program that draws a street map . . . 25
Drawing graphics primitives . . . 27
Setting up a coordinate system for drawing graphics . . . 28
Moving the current position, using GSMOVE . . . 28
Drawing a line, using GSLINE . . . 28
Drawing a series of straight lines, using GSPLNE . . . 29
Drawing a circular arc, using GSARC . . . 29
Drawing an elliptic arc, using GSELPS . . . 29
Drawing a series of curved lines, using GSPFLT . . . 29
Drawing a graphics marker symbol, using GSMARK . . . 30
Drawing a shaded graphics area, using GSAREA and GSENDA . . . 30
Querying the current position, using GSQPOS . . . 32
Drawing graphics image pictures, using GSIMG . . . 32
Drawing a scaled image picture, using GSIMGS . . . 34
Specifying graphics attributes for primitives . . . 35
Setting the current color, using GSCOL . . . 35
Setting a new current line type, using GSLT . . . 36
Setting a new current line width, using GSLW or GSFLW . . . 36
Setting the current marker symbol, using GSMS . . . 36
Changing the scale of a graphics marker symbol, using GSMB . . . 37
Setting the current shading pattern, using GSPAT . . . 37
Setting the foreground color-mixing attribute, using GSMIX . . . 40
Special treatment of the background color, using call GSMIX . . . 42
Setting the background-mix attribute, using GSBMIX . . . 44
Specifying a transform for graphics primitives, using GSSCT . . . 45
Changing attributes inside an area . . . 45
Querying the attributes of graphics in a segment . . . 46
Storing and restoring graphics-attribute values, using GSAM and GSPOP . . . 46
Changing default attribute values . . . 47
Device variations with graphics pictures . . . 48
IBM 3279 terminals . . . 48
Workstations supported by GDDM-OS/2 Link or GDDM-PCLK . . . 49
IBM 3270-PC/G and /GX workstations . . . 49
IBM 5080 and 6090 Graphics Systems . . . 50
5550-family Multistations . . . 50
Plotters . . . 50
Printers . . . 51

Chapter 3. Including text functions in your programs . . . 53
Graphics text . . . 53
Alphanumeric text . . . 54
Procedural alphanumerics . . . 54
Mapped alphanumerics . . . 54
High-performance alphanumerics (HPA) . . . 55
Comparison of the three methods of implementing alphanumeric functions . . . 56

Chapter 4. Creating graphics-text output in your application 57
Drawing graphics text 57
Drawing a line of graphics text at a specified position, using GSCHAR . . . 57
Drawing a line of graphics text at the current position, using GSCHAP . . . 57
Affecting the appearance of graphics text 58
Choosing a suitable mode of graphics text 58
Advantages and disadvantages of each character mode 61
Example: Subroutine to label the streets of the TOWN program 62
Tasks illustrated by the LABELS subroutine 64
Selecting the mode of graphics text to be used 64
Ensuring that graphics text is readable 64
Breaking lines of graphics text 64
Changing the size and proportions of text characters 64
Changing the space between characters of graphics text 65
Concatenating graphics text 65
Changing the slope of a graphics-text string 66
Changing the direction of a graphics-text string 66
Making graphics-text characters appear italic 66
Outlining the text box around a graphics-text string 66
Aligning text within the text box 67
Using proportionally-spaced characters 68
Device variations with graphics text 69
On IBM 3279 color displays 69
On the IBM 3270-PC/G and /GX workstations 69
On the IBM 5080 and 6090 Graphics Systems 69
On IBM 5550-family Multistations 69
On advanced function printers and the IBM 4250 70
On plotters 70

Chapter 5. Basic procedural alphanumerics 71
Defining an alphanumeric field, using call ASDFLD 71
Sending and receiving alphanumeric data, using ASCPUT and ASCGET 72
Breaking lines of alphanumeric text 72
Clearing an alphanumeric field, using call ASFCLR 73
Deleting an alphanumeric field, using call ASDFLD 73
Positioning the alphanumeric cursor, using ASFCUR 73
Querying the position of the alphanumeric cursor, using ASQCUR 73
Attribute bytes on 3270 terminals 74
Alphanumeric attributes 74
Setting the attributes of alphanumeric fields 74
Setting the attributes of alphanumeric characters 76
Example: Program using procedural alphanumerics to display a bank balance . 77
Points illustrated by the Bank Account program 79
Mixing alphanumeric and graphic functions 80
Device variations with procedural alphanumerics 82
3179-G, 3192-G, 3472-G, 3270-PC/G and /GX, and IPDS printers 82
IBM 5080 and 6090 graphics systems 82
5550-family multistations 83

Chapter 6. Image basics 85
Hardware required for image processing with GDDM 85
How images are defined for processing by GDDM 85
Transferring image data from one type of image to another 87
How to scan, display, and save an image 87

Scanner echoing . . . 88
Creating an image . . . 89
Loading the document into the scanner using call ISLDE . . . 89
Transferring images using call IMXFER . . . 90
Deleting images using call IMADEL . . . 91
Synchronizing output and input . . . 91
Saving images using call IMASAV . . . 91
Loading an image, using call IMARST . . . 92
Obtaining a new image identifier, using call IMAGID . . . 92
Querying image attributes . . . 93
Projections . . . 93
Example code to define and save a projection . . . 95
Creating a projection using call IMPCRT . . . 95
Extracting a rectangular sub-image using call IMREXR . . . 96
Changing the size of an extracted image using call IMRSCL . . . 97
Completing the image transform and positioning it in the target image . . . 97
Saving a projection using call IMPSAV . . . 97
Deleting a projection, using call IMPDEL . . . 98
How to apply a projection during a transfer operation . . . 98
The remaining transform elements . . . 101
Turning (reorienting) the image through multiples of 90 degrees . . . 101
Reflecting the image about a chosen axis, using call IMRREF . . . 102
Getting the negative of an image, using call IMRNEG . . . 102
Defining the resolution conversion algorithm, using call IMRRAL . . . 102
Putting transform calls in the right sequence . . . 103
Order of evaluation in projections . . . 104
Some other facilities . . . 104
Gray-scale image manipulation . . . 104
Applying a projection during image save and restore . . . 104
Getting a new projection identifier, using call IMPGID . . . 105
Changing the image resolution type, using call IMARF . . . 105
Editing images without a transfer operation . . . 105
Clearing a rectangle in an image, using call IMACLR . . . 105
Trimming an image, using call IMATRM . . . 105
Converting the resolution of an image, using call IMARES . . . 106
Using IMXFER with target image the same as source image . . . 106

Chapter 7. Hierarchy of GDDM concepts . . . 107
The device . . . 108
Virtual devices . . . 108
The partition set . . . 109
The partition . . . 109
The page and page window . . . 111
The graphics field and the image field . . . 112
The picture space . . . 113
The viewport . . . 114
The graphics window . . . 117
Uniform world coordinates . . . 118
How to avoid inverting the graphics window . . . 119
The graphics segment . . . 120
Redefining objects in the hierarchy . . . 121
Viewports and graphics windows . . . 121
Picture space and graphics field . . . 121
Other objects . . . 121

Example: Program using the GDDM hierarchy . . . 121
Concepts introduced by the TWOPAGE program . . . 123
A graphics hierarchy with two devices . . . 124
Graphics clipping . . . 125
Precise clipping at the data boundary . . . 126
Rough clipping at the data boundary . . . 127
Drawing graphics outside the segment viewing limits . . . 128

Chapter 8. Error handling and debugging . . . 131
The causes of errors in GDDM application programs . . . 131
GDDM error messages . . . 132
Identifying bugs in your program . . . 133
Querying the GDDM error record, using FSQERR . . . 133
Using GDDM trace to debug application programs . . . 134
Format of the trace output file . . . 136
Error information returned in a control block . . . 136
Information returned in register 15 . . . 136
Error information for the reentrant and system programmer interfaces . . . 136
Writing programs that can cope with error conditions . . . 136
Specifying an error exit and threshold, using call FSEXIT . . . 137
Using the default error-exit routine . . . 137
Bypassing GDDM's parameter checking to improve the speed of applications . 140

Part 2 Advanced GDDM functions . . . 143

Chapter 9. Manipulating graphics segments . . . 145
Creating segments, using GSSEG . . . 145
Deleting segments, using GSSDEL . . . 147
Segment attributes . . . 147
Unnamed segments . . . 149
Transforming segments, using GSSAGA or GSSTFM . . . 149
How and when transformations take effect . . . 152
Transforming text, markers, and graphics images . . . 153
Moving a segment and its origin using call GSSPOS . . . 153
Transforming segments using call GSSTFM . . . 154
Querying transforms . . . 156
Examples of transformations . . . 156
Moving the origin of a segment, using GSSORG . . . 158
Transforming primitives within a segment, using GSSCT . . . 159
Copying segments, using GSSCPY . . . 160
Including segments, using GSSINC . . . 162
Combining segments, using GSSINC and GSSDEL . . . 163
Drawing chain and segment priority . . . 164
Querying the order of all segments, using GSQPRI . . . 165
Calling segments from other segments, using GSCALL . . . 165
Graphics attribute handling with called segments . . . 169
Graphics not in named segments . . . 170
Primitives outside segments . . . 170

Chapter 10. Storing and retrieving graphics pictures . . . 173
The stored-graphics formats that GDDM supports . . . 173
Saving pictures in Graphics Data Format, using call GSSAVE . . . 175
Saving all graphics on the current page . . . 176

Selecting individual segments to be saved 176
Naming the file or data set in which the GDF data is to be saved 177
Specifying whether GDF files of the same name should be overwritten . . . 177
Choosing the type of GDF data for the graphics you want to save 177
Inter-Release compatibility 177
Saving pictures in Computer Graphics Metafile, using call CGSAVE 178
Naming the file or data set in which the CGM data is to be saved 178
Using a conversion profile to store CGM orders that suit another application 179
Specifying a code page for saved CGM data 179
Including descriptive text in the CGM data saved 179
Retrieving graphics pictures from external storage 179
Retrieving pictures stored in Graphics Data Format, using call GSLOAD . . 180
The three types of load 183
Maintaining a library of segments 188
Panning and zooming 189
Retrieving pictures stored in Computer Graphics Metafiles, using call CGLOAD 192
Retrieving pictures stored in Picture Interchange Format, using call GSLOAD 192
Modifying graphics pictures that have been loaded into your program 193
Placing graphics data from the GDDM page in a program variable 193
Device variations with GDF 195

Chapter 11. Writing interactive graphics applications 197
Overview of graphics input functions 197
Simple interactive graphics program 198
Locator input 201
Choice input 202
Enabling data keys for choice input in applications 202
Effects of stroke and string devices 203
Choice devices as triggers 203
Processing choice input from the data keys 203
String input 204
Enabling end users to draw graphics with the puck, mouse, or stylus 204
Querying stroke input 205
Simple polyline program 205
Enabling or disabling a logical input device, using call GSENAB 206
Passing input to your program, using call GSREAD 208
Checking for further graphics input records using call GSQSIM 209
Handling the input queue 209
Using ASREAD instead of GSREAD 210
Initializing logical input devices 211
Initializing a locator device, using call GSILOC 211
Specifying locator-echo type and initial position, using call GSILOC 211
Initializing a pick device, using calls GSIPIK and GSIDVF 213
Initializing a string device, using calls GSISTR and GSIDVI 213
Initializing a stroke device, using call GSISTK 213
Using a locator, pick, and stroke device together 214
When to issue GSENAB calls 214
Querying a logical input device 215
Segment-picking example 215
Simple free-hand drawing program 217
Dragging segments 219
How the 3270-PC/G and GX draw echoes 220

Local origin when dragging a segment . . . 221
Local origin when transforming a segment . . . 223
Panning and zooming . . . 224
Retained and nonretained modes on the 3270-PC/G and GX . . . 224
Query primitives and segments in specified area using call GSCORR . . . 225
Querying segment structure in specified area using call GSCORS . . . 228
Interactive graphics with multiple partitions . . . 229
Device variations with interactive graphics . . . 229

Chapter 12. Using symbol sets . . . 233
General information about symbol sets . . . 233
Loading symbol sets for graphics text . . . 235
Specifying a symbol set for graphics text . . . 236
Loading symbol sets for alphanumeric text . . . 238
Specifying a symbol set for use in an alphanumeric field . . . 239
Specifying a symbol set for individual characters in a field . . . 239
Multicolored image symbols . . . 242
Symbols for pounds, dollars, and cents . . . 242
Device-dependent symbol-set suffixes . . . 242
Manipulating symbol sets in a program . . . 243
Symbol sets and program variables . . . 243
Loading a symbol set from an application program . . . 243
Using double-byte characters for graphics text . . . 244
GDDM default required for Kanji and Simplified Chinese . . . 246
Using GDDM to convert character code pages for international applications . . . 247
General information on code pages and national characters . . . 247
Country-extended code pages . . . 248
Code-page conversion . . . 248
Converting code pages using API calls in the program . . . 250
Compatibility with releases of GDDM before Version 2 Release 2 . . . 253
Code-page conversion for 4250 printers . . . 253
APL characters . . . 253
Device variations with symbol sets . . . 254
Transferring programs between different types of device . . . 254
Displays that use programmed symbols for graphics . . . 254
IBM 3270-PC/G and GX workstations . . . 255
Printers managed by PSF and CDPF . . . 256
Plotters . . . 256

Chapter 13. Advanced procedural alphanumerics . . . 257
Example: Alphanumeric menu program . . . 257
Concepts introduced by the MENU program . . . 260
Defining multiple alphanumeric fields . . . 260
Setting the field attributes as you define the field . . . 260
Discovering how many fields on the current page were modified . . . 261
Identifying which fields have been modified . . . 261
Choosing advantageous field identifiers . . . 261
Redefining the attributes of existing fields . . . 262
Resetting the default value of an alphanumeric field attribute . . . 262
Processing an alphanumeric field with changed status . . . 263
Processing light-pen fields . . . 264
Using procedural alphanumerics for double-byte characters . . . 265
Example: Routine to fill an alphanumeric field with Kanji data . . . 266
Points illustrated by the example . . . 266

Performing output of strings mixing single- and double-byte characters 267
Example: Routine to mix SBCS and DBCS data in an alphanumeric field . . 268
Points illustrated by the example 269
Returning the mixed-string contents of a user-input field to the application . 269
Field outlining on the IBM 5550 Multistation 270
Improving the performance of procedural alphanumerics applications 270

Chapter 14. GDDM high-performance alphanumerics 273
How to use high-performance alphanumerics 273
Declaring and initializing the field list 273
Declaring and initializing the bundle list 274
Declaring and initializing the data buffer 274
Example: Program displaying high-performance alphanumeric output 274
Points illustrated by the EXHPA program 276
Returning HPA user input to the application 279
Displaying alphanumeric fields again 279
Field-list update rules 279
Data buffer update rule 280
Bundle list update rule 280
Dynamic fields 280
Programming HPA with interpreted languages 280
Read-only storage 281
Shared storage 281
Choosing between validation and improved performance 281

Chapter 15. Mapped alphanumerics 283
Using predefined screen formats for alphanumeric applications 283
A simple mapping application 284
Tasks illustrated by the MAPEX01 program 285
Compilation and execution of a mapping application program 288
ADS conversion for mapping applications written in C/370 288
A mapping application that sets up a dialog with the end user 289
Why you do not always need to call MSPUT 291
A typical mapping cycle 291
Steps in creating a mapping application 291
Changing existing maps 296
Using more than one map to present and process alphanumeric information . 296
Using maps with positions fixed by GDDM-IMD 296
Using several maps that position themselves relative to each other 297
Example of a program that uses fixed and floating maps 299
Querying changed maps 302
Input from multiple copies of a map 303
Device-independence for mapped-alphanumeric applications 304
Attribute handling when mapgroup does not match device 305
Output-only displays 305
Mapping queries 306

Chapter 16. Variations on a map 307
Selecting fields from a map for use in complex dialogs 307
Programming example using a selector adjunct to display a message 308
Write, rewrite, and reject 310
Selector adjuncts on input 310
Effect of reject operation 311
Uses of selector adjuncts 311

Alarm and keyboard locking . . . 315
Effects of maps . . . 315
Other considerations . . . 315
Protecting fields from the end user . . . 315
The base attribute adjunct . . . 316
Defining the base attributes that are to apply to mapped fields . . . 316
The position of the cursor . . . 318
Positioning the cursor when your program sends output to the display . . . 318
Determining the cursor position following input by the end user . . . 320
Padding mapped fields with null characters . . . 321
Light pen and CURSR SEL key . . . 321
Example of selection with cursor, light pen, and PF key . . . 322
Specifying a PF key for alphanumeric input . . . 326
Changing the highlighting, color, and symbol sets of mapped fields . . . 327
Changing the attributes of individual characters in a mapped field . . . 330
Discovering which character attributes have been changed by the user . . . 332
Folding and justification of input . . . 332
Mapping and graphics . . . 332
Example of graphics in a mapped display . . . 333

Chapter 17. Using GDDM's advanced image functions . . . 339
Querying image devices . . . 339
Scanning gray-scale images . . . 340
Defining brightness conversion definition, using call IMRBRI . . . 340
Defining contrast conversion, using call IMRCON . . . 341
Defining the conversion algorithm, using call IMRCVB . . . 342
Ordering of brightness, contrast, and image type conversion calls . . . 343
Querying image-related device characteristics . . . 343
Querying formats supported by a device, using call ISQFOR . . . 343
Querying compressions supported by a device, using call ISQCOM . . . 344
Querying resolutions supported by a device, using call ISQRES . . . 345
Scaling an image to fit the display screen . . . 346
Interactive image manipulation, using image cursors . . . 348
Enabling or disabling device input, using call FSENAB . . . 348
Enabling or disabling an image cursor, using call ISENAB . . . 348
Querying the image locator cursor, using call ISQLOC . . . 349
Querying the image box cursor, using call ISQBOX . . . 349
Initializing the image cursors, using calls ISILOC and ISIBOX . . . 350
Local operations on the 3193 display station . . . 350
Interactive image manipulation example . . . 351
Transferring images into and out of your program . . . 355
Starting a PUT operation, using call IMAPTS . . . 356
PUTTING data into an image, using call IMAPT . . . 357
Ending a PUT operation, using call IMAPTE . . . 357
Starting a GET operation, using call IMAGTS . . . 358
GETTING data from an image, using call IMAGT . . . 359
Ending a GET operation, using call IMAGTE . . . 359
Controlling host offload by specifying image quality . . . 359
Image size rounding . . . 360
Scaling and resolution conversion . . . 361
Scaling algorithm (also used in resolution conversion) . . . 361
Multiple extraction and placing of rectangles . . . 361
Controlling image quality, using call ISCTL or ISXCTL . . . 362
Direct transmission . . . 364

Direct transmission from a scanner . . . 364
Direct echoing when scanning . . . 365
Combining an image with text or graphics . . . 365
Defining an image field, using call ISFLD . . . 365
Querying the attributes of an image field, using call ISQFLD . . . 366
Printing images . . . 366
Printing an image on an IPDS printer . . . 366
Improving the performance of image programs . . . 368
Image processing on image devices . . . 368
Image processing on graphics devices . . . 369
Device variations for image . . . 370
Displays that support graphics . . . 370

Chapter 18. Device support in application programs . . . 371
Using DSOPEN to tell GDDM about a device you intend to use . . . 371
Coding a complete device definition on the DSOPEN call . . . 372
Coding a partial device definition for end users to change with nicknames . 374
How GDDM compounds device-definition information for a conceptual device . . . 375
Offering end users a menu of devices available for output . . . 376
How a nickname can cause a device definition to be revised completely . . 380
Coding nickname statements within application programs . . . 381
Specifying device usage using the DSUSE call . . . 382
Discontinuing use of a device, using DSDROP . . . 383
Using the default primary device . . . 383
Sending output to a device other than the invoking device . . . 383
Using more than one primary device . . . 384
Opening and using a dummy device . . . 387
Example: Program using a dummy device to create a stored picture . . . 388
Closing a device using the DSCLS call . . . 389
Reinitializing a device, using the DSRINIT call . . . 390

Chapter 19. Designing device-independent programs . . . 391
Device dependence in GDDM application programs . . . 391
Coping with device variation and dependence in your programs . . . 391
Avoiding dependencies when opening and using devices . . . 392

Chapter 20. Sending output from an application to a printer . . . 399
Overview of printing with GDDM . . . 399
Family-1 output: GDDM directly attached printers . . . 401
Family-2 output: Print files for GDDM queued printers . . . 402
Family-3 output: Print files for system printers . . . 404
Family-4 output: Print files for PostScript and PSF- and CDPF-attached printers . . . 404
Defining the area of the paper you want the printer to use . . . 405
Positioning graphics, image, and alphanumeric fields in the usable area . . 406
Directing the program's output . . . 406
Specifying the format to be used for family-4 output . . . 407
Specifying a data stream to suit the purposes of your family-4 output . . . 408
Retrieving family-4 output for the application . . . 409
Using a printer as an alternate device . . . 412
Copying a transformed picture to a printer, using call DSCOPY . . . 412
Copying a page to a printer using call FSCOPY . . . 413
Copying graphics to a printer using call GSCOPY . . . 414

Sending a character string to a printer using call FSLOG 414
Sending a character string with control character to printer using call FSLOGC 415
Example: Copying screen output to a printer 416
Printing GDDM family-2 print files 417
Printing composite documents 419
Example: Program to print a composite document 419
Printing non-GDDM sequential files 421
Re-rastering when copying 421
Mixed graphics and alphanumerics 422
Using loadable symbol sets on family-3 3800 printer 423
Using typographic fonts on a family-4 4250 printer 424
| Code-page support for 4250 output 425
Example: Program using 4250 fonts 426
Color masters for publications 427
DSOPEN statement for color masters 430

Chapter 21. Sending output from an application to a plotter 433
DSOPEN for plotters 433
Processing options for plotters 434
Setting up the plotter 437
Terminating a plot 438
Cells, pixels, and plotter units 438
A simple plotting program 439
| Copying screen output to a plotter 441
Plotting to scale 443
Using nicknames to direct and control plotted output 445
Diverting a program's output from a printer to a plotter 445
Diverting a program's output from a plotter to a printer 446
| Diverting a program's output from a plotter to an IBM-GL file 446
| Supplying processing options 446
Special considerations for graphics on plotters 446

Chapter 22. Designing end-user interfaces for your applications 453
Using partitions to divide up the screen 453
A simple partitioning example 454
Some other things you can do with partitions 460
Large and small pages 473
Partitioning with scrolling and variable cell size 476
Using operator windows to write task-manager programs 479
Example: Program using one operator window 481
Example: Program using two operator windows 485
Modifying the attributes of an operator window, using call WSMOD 490
Prioritizing operator windows 491
Querying the priority of overlapping operator windows 492
Querying operator window attributes, using WSQRY 494
Task management 494
How FSSAVE and FSSHOW perform with operator windows 497
Allocation of resources to operator windows 497
How to free resources when a task terminates 498

Part 3 Examples of GDDM programs 499

Chapter 23. Programming examples 501
A System/370 Assembler programming example 501
An APL2 programming example 502
A BASIC programming example 503
A C/370 programming example 505
A REXX programming example 508
A CICS pseudoconversational programming example 510

Appendixes 517

Appendix A. GDDM sample programs 519
Sample program 1 520
IMS version of sample program 1 520
Sample program 2 520
IMS version of sample program 2 520
Sample program 3 520
Sample program 4 521
What sample program 4 does 521
Invoking ADMUSP4 521
Sample program 8 521
Compiling and link-editing sample program 8 under TSO 522
Running sample program 8 under TSO 522
Compiling sample program 8 under VM/CMS 522
Running sample program 8 under VM/CMS 523
Using the sample task manager 523
Compiling, link-editing, and running the sample programs 524
Compiling the programs 524
Link-editing the programs 525
Running the sample programs 527
REXX sample programs 528

Appendix B. Programming with GDDM under VM/CMS 529
How to compile, load, and run a PL/I GDDM application program 529
Running a GDDM utility program 531
Considerations for running multiple instances of GDDM 531
Native CMS files 531
Native CMS spool files 533
Display terminal conventions 533
Asynchronous interrupts on VM/CMS 534
Using APL terminals 536
Using nonqueriable displays with the APL feature 537
Using nonqueriable printers with the APL feature 537
Batch processing 538
GDDM application programs under VM/XA 538

Appendix C. Programming with GDDM under TSO 539
Link-editing a GDDM application program 539
Using the system programmer interface by means of dynamic load 539
Data sets 540
Partitioned data sets 540

Sequential data sets . . . 541
Direct access data sets . . . 541
File-name usage . . . 542
Display terminal processing . . . 544
Using the CLEAR key in full-screen mode . . . 545
Entering attention interrupts in full-screen mode . . . 545
Reshow key processing in full-screen mode . . . 546
Device errors in full-screen mode . . . 546
Line-by-line input in full-screen mode . . . 547
NOEDIT mode under TSO . . . 547
Using APL terminals . . . 548
Using GDDM under TSO batch . . . 549
Using GDDM under MVS batch . . . 550
Programming under TSO on extensions of MVS . . . 550
GDDM code above 16 megabytes . . . 550
Application code above 16 megabytes . . . 551
AMODE(31) applications and application parameters above 16 megabytes . 551
Application programming considerations . . . 551
User exits . . . 551
Example: JCL for link-editing GDDM applications under TSO . . . 552

Appendix D. Programming with GDDM under IMS . . . 553
Restrictions on the use of GDDM under IMS . . . 553
The structure of GDDM application programs for use on IMS . . . 554
Programming under IMS on extensions of MVS . . . 556
GDDM code above 16 megabytes . . . 556
Application code above 16 megabytes . . . 556
AMODE(31) applications and application parameters above 16 megabytes . 556
Application programming considerations . . . 556
User exits . . . 556
Link-editing a GDDM application program . . . 557
Using the system programmer interface with dynamic load . . . 557
Program specification blocks for GDDM applications . . . 558
Data sets and file processing . . . 559
Specifying the default error exit under IMS . . . 559
GDDM and the Message Format Service . . . 560
GDDM DL/I interface . . . 560
Use of message queues . . . 561
Use of databases . . . 562
IMS considerations for GDDM utilities . . . 563
GDDM object import/export utility . . . 564
Example: JCL to compile and link PL/I GDDM applications under IMS . . . 565
Example: JCL to compile and link COBOL GDDM applications under IMS . . . 566

Appendix E. Programming GDDM applications for use with CICS . . . 567
Programming languages and restrictions . . . 567
CICS conversational applications . . . 567
CICS pseudoconversational applications . . . 568
Transaction-dependent pseudoconversations . . . 569
Transaction-independent pseudoconversations . . . 572
Requesting transaction-independent services . . . 573
Using the resource audit trails . . . 573
Using GDDM with Basic Mapping Support . . . 574
Using GDDM and Basic Mapping Support consecutively . . . 574

Using GDDM and BMS concurrently without coordination mode 575
Using GDDM and BMS concurrently with coordination mode 575
CICS GDDM default error exit 576
Display terminal conventions 577
Using the GDDM nonreentrant interface 577
Using the GDDM system programmer interface with dynamic load 578
Data sets and file processing 578
VSAM key-sequenced data sets 578
Transient data queues 581
Temporary storage data sets 582
Programming under CICS on extensions of MVS 583
GDDM code above 16 megabytes 583
Application code above 16 megabytes 583
AMODE(31) applications and application parameters above 16 megabytes . 583
Application programming considerations 583
User exits 584
Compiling and link-editing GDDM application programs 584
Compiling a PL/I program 584
Link-editing a GDDM application program under CICS 584
Example of JCL for compiling and linking PL/I GDDM/CICS applications on MVS 586
Example: JCL to compile and link COBOL GDDM/CICS applications on MVS . 587
Example of JCL for compiling and linking C/370 GDDM/CICS applications on MVS 588
Example: JCL to assemble and link-edit Assembler GDDM/CICS applications on MVS 589
Example: JCL to compile and link PL/I GDDM/CICS applications on VSE . . . 590
Example: JCL to compile and link COBOL GDDM/CICS applications on VSE . 591
Example: JCL to compile and link C/370 GDDM/CICS applications on VSE . . 592
Example: JCL for GDDM under CICS/VSE using Assembler 593

Appendix F. Programming with GDDM using VSE batch mode 595
Link-editing 595
Using the system programmer interface with dynamic load 596
Large 4250 page segments 596
Spill files 597

Glossary 599

Index 615

Figures

1. "HOUSE" example graphics program . . . 6
2. Output from the HOUSE example graphics program . . . 9
3. CLIST to compile, link-edit, and run a PL/I GDDM program under TSO . 15
4. A simple GDDM-REXX program . . . 16
5. Routine to terminate all instances of GDDM-REXX when run ends abnormally . . . 22
6. Program using GDDM graphics API calls to draw a map of a town . . . 25
7. Output from the TOWN program . . . 30
8. How graphics areas are shaded . . . 31
9. Output from GSIMG calls . . . 34
10. The 16 GDDM system shading patterns . . . 38
11. GDDM geometric pattern set - ADMPATTC . . . 39
12. GDDM 64-color pattern set - ADMCOLSD . . . 40
13. Color-mixing table . . . 42
14. Mode-1 and mode-2 graphics text . . . 59
15. Subroutine to name streets on the town plan . . . 62
16. Output from the subroutine to annotate the town plan . . . 63
17. Effects of proportional spacing . . . 68
18. Using alphanumeric field attributes and character attributes . . . 76
19. "Bank Account" program using alphanumeric functions . . . 78
20. Output from "Bank Account" example alphanumerics program . . . 80
21. Parts catalogue alphanumerics program . . . 81
22. Image processing . . . 86
23. Simple image program – scan, display, and save an image . . . 88
24. Projection containing a transform . . . 94
25. Projection containing two transforms . . . 95
26. The "IMGPROG2" program . . . 99
27. Resolution conversion . . . 103
28. Creating partitions . . . 110
29. Defining pages within partitions (each partition has its own pages) . . 110
30. GSFLD – defining a graphics field . . . 112
31. Defining a picture space . . . 114
32. Defining a viewport . . . 114
33. Defining a graphics hierarchy (with default partitioning) . . . 117
34. Program showing how the definition of the graphics window affects the position of graphics . . . 119
35. Program creating two GDDM pages . . . 122
36. Example of a graphics hierarchy with two devices . . . 124
37. Graphics on both sides of the data boundary with clipping switched off 126
38. Graphics crossing the data boundary with precise clipping . . . 126
39. Graphics crossing the data boundary with rough clipping . . . 127
40. The effect of segment viewing limits on a primitive exceeding them . . 128
41. Moving the primitives in a segment within the viewing limits . . . 129
42. Primitives in the same segments share the same attributes. . . . 147
43. The four segment transformations . . . 150
44. Shearing . . . 151
45. Rotation . . . 152
46. Effects of GSSPOS calls . . . 154
47. Results of example transformations . . . 157
48. The GSSORG call . . . 158

© Copyright IBM Corp. 1982, 1996

49. Copying segments, using GSSCPY 161
50. Example program using called segments 166
51. Building plan produced by called segments 168
52. Table and chair segments with origin 169
53. How application programs move saved graphics between external storage and GDDM. 175
54. Segments as saved 183
55. Segments as loaded 183
56. Type 2 load 186
57. Type 3 load 188
58. Program using type-1 load to let users to pan and zoom a saved graphics picture 189
59. Handling GDF data with GSGET and GSPUT 194
60. Graphics menu routine 199
61. Program using polylocator stroke device 206
62. Emptying the input queue 209
63. Segment-picking example 216
64. Program for freehand drawing on the screen 218
65. Program for dragging segments 219
66. Defining a local origin for dragging 221
67. Local origin of echo segment 222
68. Defining a reference point for segment dragging 223
69. Correlation with rubber box 227
70. Comparison of image and vector symbols 234
71. Program using symbol sets for graphics text 237
72. Routine to specify symbol sets for alphanumeric fields 241
73. Routine to add graphics text to the page using double-byte characters. 245
74. Routine to add graphics text to the page mixing single- and double-byte characters. 246
75. Code-page conversion 249
76. MENU programming example 257
77. Output from “Menu” program 262
78. Routine to place double-byte characters in an alphanumeric field. . . . 266
79. Routine mixing single- and double-byte characters in an alphanumeric field. 268
80. Program to display high-performance alphanumeric output 275
81. Source code of MAPEX01 286
82. Field definitions for map used by MAPEX01 285
83. Initial display of MAPEX01 288
84. Source code of MAPEX02 290
85. Typical cycle of mapping operations 292
86. Positioning of fully floating maps 298
87. Source code of MAPEX04 300
88. Field definitions for map used by MAPEX04 301
89. Typical display by MAPEX04 302
90. Listing of MAPEX05 source code 308
91. Listing of MAPEX08 source code 324
92. Field definitions of map used by MAPEX08 326
93. Listing of MAPEX09 source code 328
94. Listing of MAPEX11 source code 334
95. Typical display by MAPEX11 337
96. Field definitions of map used by MAPEX11 337
97. Program that scales an image to fit the display screen 346

98. Interactive program that enables end users to trim the edges of an image . . . 352
99. Program manipulating an image that is larger than the screen . . . 354
100. Vertical overlap . . . 361
101. Horizontal overlap . . . 362
102. Program to display a menu of output devices for end users. . . . 376
103. Program using two displays, each as the primary device . . . 385
104. Program using a dummy device to create a stored picture . . . 388
105. Overview of GDDM's support for printers . . . 399
106. Example using an IBM 4224 printer for family-1 output. . . . 401
107. Opening a device for family-4 output . . . 405
108. Copying to printers . . . 416
109. Example of using 4250 fonts . . . 426
110. Output of 4250 font example . . . 427
111. How a picture is changed into a number of color masters . . . 428
112. ADMDHIPK, the GDDM sample symbol set for color masters . . . 429
113. Creating color-separation masters . . . 431
114. Plotting area . . . 435
115. Program using plotter as primary device . . . 440
116. Program using plotter as alternate device . . . 442
117. Scale plotting program . . . 444
118. The eight GDDM line types for plotters . . . 449
119. The 16 GDDM shading patterns for plotters . . . 450
120. Example of a program using partitions to control data entry . . . 454
121. Screen formatted by the PARTEX1 partitioning program . . . 460
122. Skeleton of program changing visibility of partitions to control data entry 461
123. First panel using visible and invisible partitions . . . 465
124. Second panel using visible and invisible partitions . . . 465
125. Output of program with overlapping partitions . . . 467
126. Example of program with controlled viewing order of partitions . . . 469
127. Output from program with prioritized partition viewing . . . 471
128. Program using scrollable partitions and two cell sizes . . . 477
129. Screen with two cell sizes . . . 479
130. Hierarchy of devices and windows in a single application . . . 481
131. The “OPWIN1” program . . . 482
132. The “OPWIN2” program . . . 486
133. Task manager with several applications . . . 495
134. The coordination exit routine . . . 496
135. APL2 programming example . . . 503

Tables

1. The seven displayable colors 41
2. GDDM-supplied conversion profiles for conversion of data between ADMGDF and CGM formats. 174
3. Choice data returned by nonPC 3270 terminals 230
4. Examples using symbol sets for alphanumerics and graphics text . . . 235
5. Acceptable combinations of format and compression 356
6. Carriage-control codes for FSLOGC 415
7. Plotter-page sizes available to plotters with roll-feed media. 437
8. Suggested color scheme for plotter pens 447
9. Color and pen numbers on plotters 448
10. GDDM load library for link-edit SYSLIBs 525
11. GDDM automatic library calls 526
12. GDDM interface modules 526
13. GDDM global TXTLIBs 526
14. GDDM data-set characteristics for VM/CMS 531
15. GDDM data-set characteristics for TSO 543
16. GDDM data-set characteristics for IMS 559
17. GDDM data-set characteristics for CICS 580

© Copyright IBM Corp. 1982, 1996

Notices

The following paragraph does not apply to any country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service in this publication is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact Laboratory Counsel, Mail Point 151, IBM United Kingdom Laboratories, Hursley Park, Winchester, Hampshire SO21 2JN, England. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, 500 Columbus Avenue, Thornwood, New York, 10594, U.S.A.

General-use programming interfaces allow the customer to write programs that obtain the services of GDDM.

© Copyright IBM Corp. 1982, 1996

Trademarks and service marks

The following terms, used in this book, are trademarks or registered trademarks of IBM Corporation in the United States or other countries or both:

APL2	GDDM	PS/2
CICS	graPHIGS	System/370
CICS/ESA	IBM	VM/XA
CICS/VSE	MVS/ESA	VTAM
DisplayWrite	MVS/XA	OS/390

The following terms, used in this book, are trademarks of other companies:

CorelDRAW	Corel Systems Corporation
Freelance Plus	Lotus Corporation
Harvard Graphics	Software Publishing Corporation
Micrografx Designer	Micrografx Inc.
Monotype Times New Roman	Monotype Corporation, Limited.
Univers	Allied Corporation
PostScript	Adobe Systems Corporation

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Microsoft, Windows and the Windows 95 Logo are trademarks or registered trademarks of Microsoft Corporation.

Preface

What this book is about

This book is intended to help you understand the background to programming with the IBM GDDM Version 3 Release 2 series of products, and with GDDM/MVS as an element of OS/390.

The book contains guidance about programming aspects of GDDM, and shows you how to write GDDM programs to the best effect. It also documents general-use programming interfaces and associated guidance information. General-use programming interfaces enable the customer to write programs that request or receive the services of GDDM. If you need to know more about where programming interface information is described, or about the definitions of the different types of information in the GDDM library, you should read the *GDDM General Information* book.

Who this book is for

This book is for application designers and programmers who are familiar with:

- Application programming in the language in which the GDDM programs are to be written. For example:
  - C/370
  - COBOL
  - FORTRAN
  - PL/I
  - REXX
  - System/370 Assembler
- The subsystem under which the GDDM programs are to run. For example:
  - CICS
  - CMS
  - IMS
  - TSO
  - VSE
- The information contained in the *GDDM General Information* book.

How to use this book

This Application Programming Guide is in three parts.

Use the first part for learning the basics of programming with GDDM and the second part for guidance on more advanced programming tasks. The last part of the book is devoted to complete programming examples.

You can read the sections sequentially, or just read those sections that relate to the immediate task you want to perform. The structure of this book is detailed in the table of contents. There is an index at the back of the book, that you can use for reference.

© Copyright IBM Corp. 1982, 1996

Latest GDDM information

For up-to-date information on GDDM products, check our Home Page on the Internet at the following URL:

`http://www.hursley.ibm.com/gddm/`

You might also like to look at the IBM Software Home Page at:

`http://www.software.ibm.com`

GDDM publications

GDDM Base	*GDDM Base Application Programming Guide*, SC33-0867
	GDDM Base Application Programming Reference, SC33-0868
	GDDM Diagnosis, SC33-0870
	GDDM General Information, GC33-0866
	GDDM/MVS Program Directory, GC33-1801
	GDDM/VM Program Directory, GC33-1802
	GDDM/VSE Program Directory, GC33-1803
	GDDM Messages, SC33-0869
	GDDM Series Licensed Program Specifications, GC33-0876
	GDDM System Customization and Administration, SC33-0871
	GDDM User's Guide, SC33-0875
	GDDM Using the Image Symbol Editor, SC33-0920
GDDM-GKS	*GDDM-GKS Programming Guide and Reference*, SC33-0334
GDDM-IMD	*GDDM Interactive Map Definition*, SC33-0338
GDDM-IVU	*GDDM Image View Utility*, SC33-0479
GDDM-PGF	*GDDM-PGF Application Programming Guide*, SC33-0913
	GDDM-PGF Programming Reference, SC33-0333
	GDDM-PGF Interactive Chart Utility, SC33-0328
	GDDM-PGF Vector Symbol Editor, SC33-0330
	GDDM-PGF OPS User's Guide, SC33-1776

GDDM/MVS is an element of OS/390. GDDM-REXX/MVS and GDDM-PGF are optional features of OS/390. For a complete list of the publications associated with OS/390, see the *OS/390 Information Roadmap*, GC28-1727.

Books from related libraries

Advanced function printing: Printer information, G544-3290.
APL2 Programming: Guide, SH21-1072.
C Language Manual, SC09-1128.
C User's Guide, SC09-1263.
CICS/ESA 3.3 Application Programming Reference Manual, SC33-0676.
CICS/ESA 4.1 Application Programming Reference Manual, SC33-1170.
CICS/VSE Application Programming Reference Manual, SC33-0713.
Composed Document Printing Facility General Information, GC33-6133.
Document Composition Facility and Document Library Facility General Information Manual, GH20-9158.
Document Composition Facility Script/VS Language Reference, SH35-0070.
GWSP User's Guide, SC33-0574.
GWSP Migration from Graphics Control Program, SC33-0563.
IBM BASIC Application Programmer's Guide, GC26-4026.
IBM-GL Programming Manual (Graphics Language), SH23-0092.
OS PL/I Programming Guide, SC26-4307.
3270 PC/G or PC/GX Reference, SC33-0181.
PSF/MVS: Application Programming Guide, S544-3673.
PSF/VM: Application Programming Guide, S544-3677.
PSF/VSE: Application Programming Guide, S544-3666.
VS COBOL II Application Programming Guide, SC26-4045.
VS Fortran Programmer's Guide, SC26-4118.

Summary of changes

Changes to this book for Version 3 Release 2

Minor technical updates and corrections have been made throughout the book.

Changes to this book for Version 3 Release 1

The following changes have been made to this book for Version 3 Release1:

- Only guidance information for programming to the GDDM **Base** API is contained in this book. Guidance information for programming to the GDDM-PGF API is now contained in a separate book, the *GDDM-PGF Application Programming Guide*.
- Guidance information for GDDM programming under the various supported subsystems is now included in the appendixes of this book.
- Task-oriented headings have been used to aid navigation through the book.
- Chapter 1, “An introduction to programming with GDDM” on page 3 has been changed to reflect the support of the C/370 and REXX programming languages by the GDDM Base API.
- Chapter 19, “Designing device-independent programs” on page 391 has been added to help programmers design programs that work equally well on many devices.
- The information in Chapter 20, “Sending output from an application to a printer” on page 399 relating to the use of printers for family-4 output has been changed to reflect the enhanced support for advanced-function printing.
- The information in Chapter 21, “Sending output from an application to a plotter” on page 433 has been changed to reflect changes in the API that allow programs to produce long plots on certain devices.

© Copyright IBM Corp. 1982, 1996

Part 1 GDDM basics

© Copyright IBM Corp. 1982, 1996

Chapter 1. An introduction to programming with GDDM

GDDM is a family of IBM licensed programs that make it possible for application programs to produce graphics, alphanumerics, and images on display devices, printers, and plotters, and to read input from display devices. These graphics, alphanumerics, and image facilities, known as **base** facilities, are accessed by means of a call-type application programming interface (API).

This book is a guide to programming with GDDM, rather than a comprehensive reference document. The *GDDM Base Application Programming Reference* book contains complete descriptions of all the calls and their parameters.

All the illustrations in this book are produced by GDDM programs.

Supported languages

The GDDM Application Programming Interface supports several popular programming languages. Here is a list of supported languages with some examples of calls to the GDDM API for each one.

PL/I

```
CALL FSINIT;
CALL ASREAD(TYPE,VALUE,COUNT);
```

FORTRAN

```
CALL FSINIT
CALL ASREAD(TYPE,VALUE,COUNT)
```

COBOL

```
CALL 'FSINIT'.
CALL 'ASREAD' USING TYPE, VALUE, COUNT.
```

System/370 Assembler

```
CALL FSINIT,(0),VL
CALL ASREAD,(TYPE,VALUE,COUNT),VL
```

REXX

```
Address command 'GDDMREXX INIT' /* On CMS */
Address link 'GDDMREXX INIT'    /* On TSO */
Address gddm
'ASREAD .TYPE .VALUE .COUNT'
```

C/370

```
fsinit();
asread(&type,&value,&count);
```

Sample programs in each of these supported languages, except assembler, are supplied with GDDM. For information about these, see Appendix A, “GDDM sample programs” on page 519.

APL2 and BASIC can also call GDDM functions. This support, however, is provided by software associated with the languages rather than by GDDM. For further information you will need to refer to the manuals describing this language-related software. You can find examples of GDDM programs written in APL2 and BASIC in Chapter 23, “Programming examples” on page 501.

© Copyright IBM Corp. 1982, 1996

The GDDM external application programming interfaces

There are three different programming interfaces that you can use to invoke the functions of GDDM from an application program.

Whichever one you use, you must link-edit or load the appropriate interface module with your program to convert call statements into the standard internal interface of GDDM.

For more detailed information on each of these interfaces, see the *GDDM Base Application Programming Reference* book.

The nonreentrant programming interface

This is the simplest programming interface for writing application programs that use GDDM.

All the examples in this book use the nonreentrant interface.

Special actions are required to use the nonreentrant interface under the CICS subsystem (see Appendix E, “Programming GDDM applications for use with CICS” on page 567).

The reentrant programming interface

This allows programs using GDDM to be made reentrant; that is, capable of use by more than one user simultaneously or by the same user in different windows simultaneously.

For an application to use this interface of GDDM, every call statement to GDDM must include a special control block called the application anchor block (AAB). The application must provide at least 8 bytes of storage for the AAB. The application anchor block identifies the particular instance of GDDM being addressed. If the AAB is in reentrant storage, any program or module addressing that instance of GDDM runs reentrantly.

If your application program is modular and several modules need to use GDDM in a reentrant way, the program can pass the AAB as a parameter across its module calls. Alternatively, under the CICS subsystem, quasi-reentrancy can be achieved by locating the AAB in the program’s Transaction Work Area (TWA).

Note: The program is free to extend the application anchor block for other uses, such as passing information to an error exit routine.

Multiple instances of GDDM and GDDM-REXX can be controlled separately. A REXX program can have multiple instances of GDDM-REXX, each of which can have multiple instances of GDDM. Each instance of GDDM-REXX is initialized by the “GDDMREXX INIT” command and ended by “GDDMREXX TERM.” Each instance of GDDM is initialized by the “FSINIT” call and ended by the “FSTERM” call.

The system programmer interface

This interface is provided for programmers who intend to use GDDM to create graphics software products of their own. This interface enables you to invoke each GDDM function by means of a function code (request control parameter, RCP) rather than by issuing an API call.

The system-programmer interface is available only in reentrant form and shares many features of the reentrant interface. Calls to the system-programmer interface can be mixed with calls to the reentrant interface in the same program.

The system-programmer interface provides an initialization function of its own (SPINIT). This gives the program greater control over the subsystem environment and allows more programming flexibility. All other functions are invoked by calls to the ADMASP entry point to GDDM. Each ADMASP call passes an application anchor block, the RCP code of an API call, and the parameters of that API call to the GDDM interface modules link-edited or loaded with the program. System/370 Assembler language definitions of the RCP codes are supplied as part of GDDM. For more information on using the system-programmer interface, see the *GDDM Base Application Programming Reference* book.

Example programs

Throughout this book, PL/I example programs and code fragments are used to illustrate specific points about GDDM. In each section, new concepts are introduced in example programs such as the HOUSE program in Figure 1 on page 6. The text that follows these programs explains these new concepts by referring to specific statements in the programs using reference keys, which look like this **A** .

These example programs are not necessarily intended to demonstrate good programming practice. A well-written application program might test the return codes from every GDDM call and take special action to handle any errors. The example programs in this book do not, in general, do this because it would obscure the main points.

Chapter 23, “Programming examples” on page 501 contains programs that use GDDM's functions written in each of these languages:

APL2
Basic
C/370
REXX
System/370 Assembler

In most sections of this book, it is assumed that the device invoking the programs is an IBM 3472-G terminal. The descriptions of GDDM calls reflect their functions on this terminal. If functions differ when invoked on other devices, these variations are described at the end of each section.

The programming examples in Chapter 11, “Writing interactive graphics applications” on page 197 assume the 3270-PC/G or GX to be the invoking device.

Example: The HOUSE program

Figure 1 shows a simple PL/I graphics program to draw a picture of a house, with its dimensions marked. The output of the program is shown in Figure 2 on page 9. If you like, when you have read the explanation of the calls in the program, you can copy it, and try putting in the calls to draw some windows.

```
HOUSE: PROC OPTIONS(MAIN);

DCL (TYPE,VAL,COUNT) FIXED BIN(31); /* Parameters for ASREAD   */

CALL FSINIT;           /* Initialize GDDM                     */   A

CALL GSSEG(0);         /* Create a graphics segment to        */   B
                       /* contain the lines and text that     */
                       /* make up the picture                 */
CALL GSCOL(7);         /* Set color to neutral (white)        */   C
CALL GSLW(2);          /* Set line width to thick             */   C

/**************************/
/* DRAW OUTLINE OF HOUSE  */
/**************************/

CALL GSMOVE(20.0,70.0); /* Move current position to (X=20,Y=70)*/   D
CALL GSLINE(20.0,20.0); /* Draw line from current position to  */   E
                        /* (X=20,Y=20)                         */
CALL GSLINE(80.0,20.0);
CALL GSLINE(80.0,70.0);
CALL GSLINE(20.0,70.0);
CALL GSMOVE(45.0,20.0); /* Move to begin drawing doorway       */
CALL GSLINE(45.0,40.0);
CALL GSLINE(55.0,40.0);
CALL GSLINE(55.0,20.0);

/**************************/
/* NOW DRAW THE ROOF      */
/**************************/

CALL GSCOL(2);          /* Set color to red                    */
CALL GSAREA(1);         /* Start an area - a shaded shape      */   F
CALL GSMOVE(15.0,70.0); /* Move to begin drawing roof          */
CALL GSLINE(35.0,95.0); /* Draw first edge of roof             */
CALL GSLINE(65.0,95.0); /* and so on...                        */
CALL GSLINE(85.0,70.0);
CALL GSLINE(15.0,70.0);
CALL GSENDA;            /* Area now complete and is shaded     */
```

Figure 1 (Part 1 of 2). "HOUSE" example graphics program

```
/**************************/
/* ADD DIMENSIONS         */
/**************************/

CALL GSCOL(5);          /* Set color to turquoise             */
CALL GSLW(1);           /* Set line width to normal           */
CALL GSMOVE(20.0,15.0); /* Move to begin dimensioning         */
CALL GSLINE(47.0,15.0); /* Draw first stroke of first arrow   */
CALL GSMOVE(22.0,13.0); /* and so on...                       */
CALL GSLINE(20.0,15.0);
CALL GSLINE(22.0,17.0);
CALL GSCHAR(49.0,14.0,2,'50'); /* 2 characters at (x=49,y=14)  */   G
CALL GSMOVE(53.0,15.0);        /* Begin second arrow           */
CALL GSLINE(80.0,15.0);        /* and so on...                 */
CALL GSLINE(78.0,13.0);
CALL GSMOVE(78.0,17.0);
CALL GSLINE(80.0,15.0);

CALL GSCHAR(33.0,2.0,28,'All dimensions are in feet  ');          H
                               /* 28 characters at (x=33,y=2)   */

CALL GSSCLS;                   /* Close the graphics segment    */

/**************************/
/* SEND PICTURE TO SCREEN */
/**************************/

CALL ASREAD(TYPE,VAL,COUNT); /* Send the picture to the screen */   I
                             /* and await a response           */
CALL FSTERM;                 /* Terminate GDDM                 */   J

                             /* GDDM Entry point declarations  */
%INCLUDE ADMUPINA;           /* for calls beginning  A.....    */   K
%INCLUDE ADMUPINF;           /* for calls beginning  F.....    */   K
%INCLUDE ADMUPING;           /* for calls beginning  G.....    */   K

END HOUSE;
```

Figure 1 (Part 2 of 2). "HOUSE" example graphics program

Concepts introduced by the HOUSE program

This program introduces the following important GDDM calls and concepts:

Initialization of GDDM

Before an application program can use any GDDM functions, it must initialize GDDM by issuing the FSINIT call as at **A** in the HOUSE program.

Alternatively, if you wish to use the system-programmer interface, (on its own or mixed with calls from the reentrant interface) you must use the SPINIT call to initialize GDDM.

Termination of GDDM

At **J** in the program, the FSTERM call terminates the instance of GDDM, freeing up storage and other resources that GDDM has been using.

If you don't do this, other programs (or reruns of the same program) may fail through lack of storage.

Graphics primitives

These are the graphics objects (lines, arcs, areas) that make up the picture.

Examples of calls requesting the addition of a **graphics primitive** to the picture can be found at points in the program marked **E** and **F**.

Graphics attributes

These govern the visual characteristics of graphics primitives (color, line type, line width).

All **graphics attributes** have default values initially. You only need to set a particular attribute when you require a different value. Calls that change the values of graphics attributes can be found at the points marked **C** in the HOUSE program.

The graphics segment

The GSSEG call at **B** creates a **graphics segment**, which is a logical grouping of primitives and the attributes that determine their appearance.

Note: It is recommended that you always create segments in which to enclose any graphics drawn by your program. By doing this, you ensure that you can save your picture, print it, and manipulate the different segments in your program.

The current position

An important notion when drawing graphics is the **current position**. When you draw a line, for example, you do not specify its start point. The line is drawn from the current position to the specified end point. The current position is normally the end point of the previous primitive, but it can be set explicitly by calling GSMOVE, as at **D** in the program.

Graphics text

The GSCHAR call at **H** in the example program produces **graphics text**. It is composed of lines, arcs, areas, and dots like other graphics primitives.

Graphics text should not be confused with alphanumerics (which is described in Chapter 5, "Basic procedural alphanumerics" on page 71).

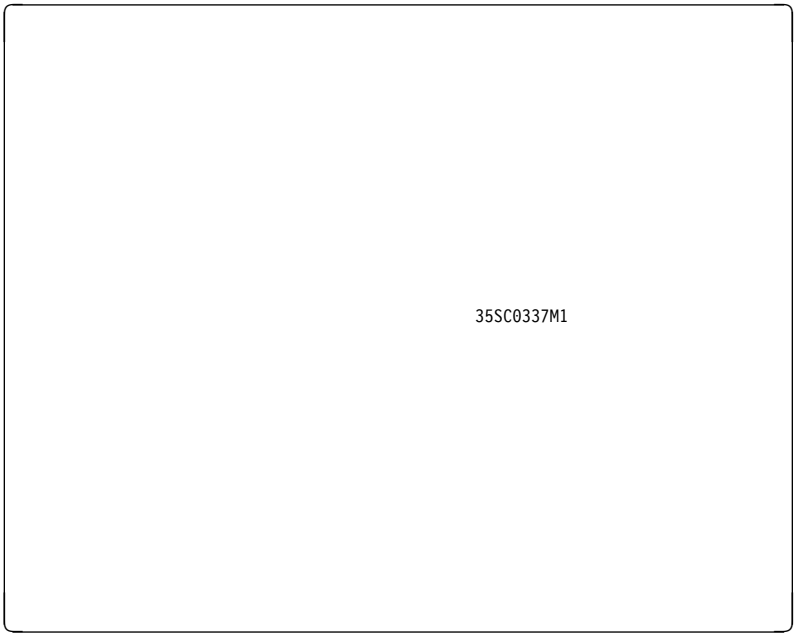


Figure 2. Output from the HOUSE example graphics program

The GDDM page

The picture built up as a logical entity by GDDM is called **the page**. The HOUSE program has only one page but a program can explicitly create and use multiple pages. Only one page can be in use, or current at any one time. All graphics calls refer to **the current page**.

Sending output from a GDDM program to a device

The picture gradually being built by GDDM is held on the current page and does not appear on the screen unless a call specifically sends it to the display. The call most commonly used to do this is ASREAD, which is issued at **I** in the example program.

ASREAD requests a write-and-read operation; it sends the picture to the screen and then performs a screen “read” awaiting a response or “interrupt” from the end user of the program.

While GDDM waits for the end user's response, it positions the cursor at the top left-hand corner of the screen, unless an ASFCUR call or the end user specifies otherwise.

The end user can reply to the read by pressing a key, such as the ENTER key or a PF key. The three parameters of ASREAD are then set by GDDM to indicate the type of response. Control returns to the program at the statement following the ASREAD. In the example, the type of response is not relevant; the program terminates.

If subsequent calls in a program or actions by the end user change the graphics or alphanumeric contents of the picture, an ASREAD call “updates” the screen. GDDM sends only those parts of the picture that have been changed.

Another call for sending output to a device is FSFRCE. Like ASREAD, FSFRCE causes the picture on the GDDM page to be sent to the device or updated but without waiting for a response from the end user. On display devices, the picture appears for an instant and disappears as control returns immediately to the application. The FSFRCE call takes no parameters:

```
CALL FSFRCE; /* Send data stream to device and return to program */
```

The primary use of FSFRCE is to send output to a device that processes only output (such as a printer or plotter).

Another use is to send a sequence of pictures to a device (rather like a slide show) where the timing of the displays is handled by the program in some way.

As with ASREAD, the cursor is positioned in the top left-hand corner of the screen, unless otherwise specified by an ASFCUR call.

For more information about using ASREAD and FSFRCE to send output to devices, see Chapter 18, “Device support in application programs” on page 371, Chapter 20, “Sending output from an application to a printer” on page 399, and Chapter 21, “Sending output from an application to a plotter” on page 433.

Learning interactively to program with GDDM

If you write programs on a subsystem that supports the REXX programming language, such as CMS or TSO, you can use the ERXTRY exec, which is supplied with GDDM, to try out GDDM calls as you learn them.

ERXTRY executes each call the moment you issue it, so you can build up a picture on your display call by call.

In this way you can write simple GDDM routines in REXX, without much knowledge of the language at all.

GDDM call statements in REXX can be easily converted into other programming languages. More information about ERXTRY can be found under “ERXTRY—the easiest way to write GDDM-REXX programs” on page 17.

Error handling

For reasons of clarity, the HOUSE example program does not test for errors in the GDDM calls. If there is an error, GDDM issues two messages. The first names the call and gives its location in main storage. The second describes the error. Execution then continues with the next statement in the program.

Eventually execution reaches the output statement (ASREAD in the example). This may or may not produce recognizable graphics, depending on the errors. The end user needs to both clear the error messages from the screen and satisfy the outstanding read. This may involve two interactions.

More information about error handling is given in Chapter 8, “Error handling and debugging” on page 131.

Entry points to GDDM

For application programs written in either PL/I or C/370, a GDDM entry point (call name) needs to be declared for each call used by the program. For these two programming languages, a set of files containing the entry declarations for API calls is supplied with GDDM. If you include these files with your programs, **you don't have to worry about coding the declarations explicitly**. These declarations specify the data types of the parameters for any GDDM calls used in the program.

At the points marked **K** in the HOUSE program, GDDM entry-point-declaration files, for the nonreentrant interface, are included for API calls that begin with the letters A, F, and G.

For detailed information on entry point declaration files for PL/I and C/370, see the *GDDM Base Application Programming Reference* book.

Data types of GDDM call parameters

Because the parameters used with the GDDM calls in the HOUSE program were all constants, the data type of some parameters is not apparent. In most programs, however, it is advantageous to use variables as the parameters of GDDM calls. To do this, you need to declare these variables to be of a particular data type.

Notes:

1. If you address the GDDM API in a **REXX** exec, you do not need to declare the data type of parameter variables before passing them on API calls. With REXX, GDDM expects values of specific type for each parameter of each call. As long as the value specified for a parameter, either as a constant or a variable, matches the expected type, it is valid for the call.
2. In C/370 application programs, parameters returned by GDDM must be passed by address rather than name.

Floating point, integer, and character are the three basic data types used with GDDM call statements but you can also declare variables of type array or structure, which are composite types of the basic ones.

Floating point

Parameter variables of this type are specified for calls that specify positioning of any kind.

In PL/I, variables for use with such calls as GSMOVE at **D**, GSLINE at **E**, and GSCHAR at **H** in the program, must be declared as type FLOAT DECIMAL(6).

Equivalent data types in other programming languages are:

Assembler	E-constant
COBOL	COMPUTATIONAL-1
FORTRAN	REAL*4
C/370	float

Integers

In some programming languages, you can declare two different kinds of integer variable; fullword integers and halfword integers. If you are certain that the value in an integer variable will never exceed 15 bits of storage, you can declare that variable as a halfword integer and thereby save storage.

With most GDDM calls, however, halfword-integer variables are assigned the same amount of storage as fullword integers.

In PL/I, variables for use with such calls as GSCOL and GSLW at points marked **C**, and ASREAD at **I** in the program, must be declared as type FIXED BIN(31).

Some GDDM high-performance alphanumeric calls require arrays of halfword integers and so the variables are **not** converted to fullwords in storage. In PL/I, variables for use in the third parameter of the APDEF call must be declared as type FIXED BIN(15).

Equivalent data types in other programming languages are:

	fullword	**halfword**
Assembler	F-constant	H-constant
COBOL	PICTURE S9(8)	PICTURE S9(2)
FORTRAN	INTEGER*4	INTEGER*2
C/370	int	short

Character

Variables of this type can be specified for:

The names of files created by a GDDM application
Graphics text output
Alphanumeric output
Alphanumeric input

In PL/I, variables to be used as the character string parameters for such calls as GSCHAR at **G** in the example program, and GSCHAP must be declared as type CHARACTER.

Equivalent data types in other programming languages are:

Assembler	C-constant
COBOL	PICTURE X(n)
FORTRAN	string literals (or numeric data array initialized with string literals)
C/370	array of type char

Note: An array of n-byte character tokens must be defined as a two-dimensional array of type char.

Arrays

Some GDDM API calls require an array of values for one or more parameters.

Because the HOUSE program doesn't contain an example of such a call, this example of a GSPLNE call is taken from "Example: Program that draws a street map" on page 25.

```
DCL S1XVALS(4) FLOAT DEC(6)  INIT( 0.0,  0.0, 50.0, 50.0);
DCL S1YVALS(4) FLOAT DEC(6)  INIT(95.0, 90.0, 90.0, 95.0);

CALL GSPLNE(4, S1XVALS, S1YVALS); /* Polyline joining 4 points*/
```

The second and third parameters of the GSPLNE call are each the name of an array of four elements.

The type of an array is the type of its individual elements; in this case, short floating point. For most calls, the required arrays are one-dimensional. However, if you are declaring a multidimensional array, make sure that it uses the correct storage mapping. In PL/I arrays are usually stored in row-major order. If you use FORTRAN or another language that specifies arrays a column at a time, you need to switch the row and column specifications described for the call in the *GDDM Base Application Programming Reference* book.

The recommended use for the array data type in each of the supported programming languages is:

Assembler	Use contiguous fullwords or halfwords
COBOL	Use the OCCURS clause.
FORTRAN	Use a one-dimensional array. (Multidimensional arrays are column major)
C/370	Use a one-dimensional array. (Multidimensional arrays are row major)

Structures

By using structures in your program, you can make some of the more complex GDDM calls easier to manage. With the reentrant and system-programmer interfaces, a structure can be used to interpret the application anchor block parameter passed to calls. The data returned by complex query calls such as FSQERR can be assigned into a structure and the program can then refer to specific locations for queried information.

A number of C/370 structure templates are supplied with GDDM in the ADMTSTRC.H header file.

You can use a structure to include a map, created by GDDM Interactive Map Definition (GDDM-IMD), in an application program. You can code the inclusion of a map, for each of the supported languages like this:

Assembler	Use the appropriate storage mapping
COBOL	`01 structure-name` `COPY mapname`
FORTRAN	GDDM-IMD doesn't generate an application data structure for FORTRAN.
PL/I	`DECLARE 1 structure-name` `%INCLUDE mapname`
C/370	GDDM-IMD doesn't generate an application data structure for C/370. See "ADS conversion for mapping applications written in C/370" on page 288 for advice on the use of mapping.

The description of the GDDM base calls in the *GDDM Base Application Programming Reference* book specifies what data type is required for each parameter.

How to compile and run a PL/I GDDM program

The commands for doing this differ depending on the subsystem on which the application program is to run. For example purposes, the typical commands used on the CMS and TSO subsystems are provided here. You can find the commands for compiling and link-editing programs on any subsystem that supports GDDM in the relevant appendix at the back of this book.

How to compile and run the HOUSE program under VM/CMS

1. Link and access the disks that hold GDDM and the PL/I compiler at your installation.
2. Link the macro library containing the GDDM entry point declarations, using this command:

   ```
   GLOBAL MACLIB ADMLIB
   ```

3. Invoke the PL/I Optimizing Compiler to compile the program, using this command:

   ```
   PLIOPT HOUSE (INCLUDE FLAG(I)
   ```

   The INCLUDE option is required to pick up ADMUPINA, ADMUPINF, and ADMUPING, the GDDM entry points for calls used in the program.

   The FLAG(I) option is not essential, but it ensures that useful messages about dummy variables are not suppressed. These are issued when parameters do not match GDDM's requirements exactly.

4. Specify the run-time libraries to be used by the program.

   ```
   GLOBAL TXTLIB ADMNLIB ADMHLIB ADMGLIB IBMLIB PLILIB
   ```

   The GLOBAL TXTLIB command tells CMS to use the text libraries containing GDDM and PL/I. ADMGLIB must be the last GDDM text library listed.

5. Load the program into storage and start it running.

   ```
   LOAD HOUSE (START
   ```

The picture of the house appears on the screen of the device you use to invoke the program.

How to compile, link-edit, and run the HOUSE program under TSO

This is an example of a CLIST that you can use to compile the HOUSE program under the TSO subsystem. Before you run it, you need to perform the following tasks:

1. Create these partitioned data sets:

	LRECL	RECFM	BLKSIZE
YOUR.DATASET.PLI	80	FB	8800
YOUR.DATASET.OBJ	80	FB	3120
YOUR.DATASET.LOAD	0	U	32760

2. Create a member, `YOUR.DATASET.OBJ(INCLCARD)`, to tell the linkage editor which external interface to GDDM your application program uses. The HOUSE program uses the nonreentrant interface, so you need to use this statement:

   ```
   INCLUDE INCLIB(ADMASNT)
   ```

For a program using the reentrant interface, you would use a statement such as this:

```
INCLUDE INCLIB(ADMASRT)
```

For a program using the system-programmer interface, you would use a statement such as this:

```
INCLUDE INCLIB(ADMASPT)
```

3. Customize the data-set names in the CLIST for GDDM and PL/I if your installation uses different names from those supplied.

```
PROC 1 NAME
/*****************************/
/*  CALL THE PL/I COMPILER   */
/*****************************/
PLI 'YOUR.DATASET.PLI(&NAME)' +
    OBJECT('YOUR.DATASET.OBJ(&NAME)') +
       LIB('GDDM.SADMSAM') FLAG(I)
/*****************************/
/*  CALL THE LINKAGE EDITOR  */
/*****************************/
ALLOC F(INCLIB) DA('GDDM.SADMMOD') REUSE SHR
LINK ('YOUR.DATASET.OBJ(&NAME)'  +
      'YOUR.DATASET.OBJ(INCLCARD)') +
      LIB('SYS1.SIBMBASE') +
      LOAD('YOUR.DATASET.LOAD(&NAME)') +
      LIST PLIBASE PRINT(*)
/*****************************/
/*  RUN YOUR PROGRAM         */
/*****************************/
ALLOC F(ADMSYMBL) DA('GDDM.SADMSYM') REUSE SHR
CALL 'YOUR.DATASET.LOAD(&NAME)'
```

Figure 3. CLIST to compile, link-edit, and run a PL/I GDDM program under TSO

GDDM-REXX—the fast path to programming with GDDM

If you are new to programming with GDDM and are still learning the basic concepts, it is recommended that you write your first GDDM programs in the REXX language. GDDM-REXX allows you to write programs in the REXX language that use GDDM's facilities. It contains all the base API calls and some extra subcommands and utilities.

Prototyping your solutions quickly

It is easier to write programs using GDDM-REXX because the syntax of GDDM calls in REXX is simpler than in other languages.

GDDM-REXX is interpreted, and this means that performance is not as good as that of efficient compiled or assembled code. It also means that GDDM-REXX programs are easier to debug. The speed of producing a solution is often as important as fast execution, and in this respect GDDM-REXX is often better than other methods of writing applications that use GDDM. Its neat syntax and the power of the REXX language make it perfectly suitable for many applications.

If you are using GDDM-REXX for prototyping, you should bear in mind the information under “Points to remember when using GDDM-REXX” on page 16 and “Coding styles—strict or loose syntax” on page 23.

An example of a REXX program using GDDM's functions

Here is an example of a simple GDDM-REXX program.

```
/* REXX                                                          */
/* program to draw a triangle and write some alphanumeric text   */
/* set up values in REXX                                         */
id=1                              /* identifier for field 1      */
row=3                             /* starts on row 3...          */
col=1                             /* ...in column 1              */
depth=1                           /* 1 row deep                  */
width=80                          /* 80 columns wide             */
type=0                            /* takes input and output      */
Address command 'GDDMREXX INIT'   /* initialize under CMS        */
/* Address link 'GDDMREXX INIT'      initialize under MVS or TSO  */
Address gddm                      /* pass commands to GDDM       */
'GSMOVE 50 50'                    /* move to point at X=50 Y=50  */
'GSLINE 70 50'                    /* draw line to point X=70 Y=50 */
'GSLINE 70 70'                    /* draw line to point X=70 Y=70 */
'GSLINE 50 50'                    /* draw line to point X=50 Y=50 */
'ASDFLD .id .row .col .depth .width .type'    /* create a field    */
'ASCPUT .id . "Right-angled triangle" '       /* put words in field */
'ASCGET .id 80 .inputval'         /* put input in inputval       */
'ASREAD .typ .val .count'         /* send graphics & text to screen*/
Address command 'GDDMREXX TERM'   /* terminate under CMS         */
/* Address link 'GDDMREXX TERM'      terminate under MVS or TSO   */
Exit                              /* stop the program            */
```

Figure 4. A simple GDDM-REXX program

Points to remember when using GDDM-REXX

Note the following points in Figure 4 and remember them when writing your own GDDM-REXX programs.

1. Begin every REXX program you write with a comment. For TSO, the first comment line must contain the keyword REXX.
2. Issue the `GDDMREXX INIT` command in the program before addressing GDDM or issuing any GDDM calls. Before you exit from the program, you must issue the `GDDMREXX TERM` command to tell GDDMREXX that it is no longer needed. Because `GDDMREXX` is a subsystem command, it must be preceded by an `Address command` instruction under CMS or by an `Address link` instruction under MVS or TSO unless the interpreter is already set up to send commands to the subsystem.
3. An `Address gddm` instruction must precede the first GDDM call. As can be seen in Figure 4, this command is issued after GDDM-REXX is initialized.
4. GDDM calls should always be in quotes to avoid REXX making changes before the calls are passed to GDDM. The `GDDMREXX` command must be in quotes if you use any of the options with parentheses.
5. Variable names in GDDM calls and GDDM-REXX subcommands must be preceded by dots.

6. Array parameters can be passed in the following forms:
   - As REXX stemmed variables, which are followed by a dot in the parameter string in the normal REXX manner. For example:

     ```
     'GSPLNE 3 .xarray. .yarray.'
     ```

   - Element by element in parentheses. The elements may be either variable names or values. For example:

     ```
     'GSPLNE 3 (.left .xcenter .right) (10 .ycenter 20)'
     ```

   - As names that GDDM-REXX will suffix with 1, 2, 3 and so on; the variables with these new names are formed into a list and passed as an array to GDDM. For example:

     ```
     varx1=10; varx2=8; varx3=5
     ```

     (and so on)

     ```
     'GSPLNE 3 .varx .vary'
     ```

     is passed to GDDM as

     ```
     'GSPLNE 3 (.varx1 .varx2 .varx3) (.vary1 .vary2 .vary3)'
     ```

   Array parameters are more strictly interpreted in GDDM-REXX than they are in other programming languages. For example, arrays consisting of four sets of two values are treated as two-dimensional in GDDM-REXX, although they can be treated as one-dimensional in other languages. If an array is passed by name, it will be indexed from element 1 irrespective of whether that is the first element, or whether it exists.

7. Some parameters can be replaced by dots, for example, lengths and counts that GDDM-REXX can calculate for itself, and returned values that a program does not refer to. At the ASCPUT call in Figure 4 on page 16, the second parameter is substituted by a dot, so the programmer doesn't need to count up all the characters in the string supplied as the third parameter.

ERXTRY—the easiest way to write GDDM-REXX programs

The best way to learn about GDDM-REXX is by using it at a terminal.

Begin by running the REXX sample programs that are provided with the product. You can alter them to explore GDDM, REXX, and some of the special features of GDDM-REXX.

One of these sample programs, ERXTRY takes most of the hard work out of GDDM-REXX programming. It takes care of the first four tasks mentioned in "Points to remember when using GDDM-REXX" on page 16 so all you need worry about are the calls themselves and their parameters. Each call you issue using ERXTRY takes immediate effect on the screen and, if you pass it a filename when you invoke it, it saves all your calls (and any error messages) in a file of that name. You can reexecute or edit your exec after you have exited from ERXTRY.

Unsure of the syntax of a GDDM call? — ERXPROTO

If you know the name and function of a call you need to use but cannot remember the exact way to code its parameters, use the ERXPROTO exec, which is also supplied with the product. Suppose you can't remember how to code the call to save a graphics segment. Type the following:

```
ERXPROTO GSSAVE
```

This syntax description appears on your screen:

```
'GSSAVE cnt1 intg.cnt1 char.8 cnt4 intg.cnt4 len6 char.len6'
```

If you use ERXPROTO while writing a program, it inserts the call description at the current line of the file, so all you need to do is overtype the description with the parameter values you require.

Converting PL/I examples to GDDM-REXX

The majority of programming examples used in this book to illustrate particular aspects of GDDM are written in PL/I. To transform these into GDDM-REXX, you should:

1. Remove the parentheses and commas from the GDDM calls, and the semicolons if there is only one statement per line.
2. Remove the DCL statements, substituting assignment statements where the DCL contains INIT.
3. Remove the `FSINIT` and `FSTERM` calls. You will need to supply `GDDMREXX INIT` and `TERM` commands.
4. Remove the %INCLUDE statements and other statements that are PL/I-only, taking care with PL/I labels and array handling.

PL/I	GDDM-REXX equivalent
`A: PROC OPTIONS (MAIN);`	`/* REXX comment */`
`CALL FSINIT;`	`Address command 'GDDMREXX INIT'`
	`Address gddm`
`DCL x, y;`	`/* remove simple DCL */`
`DCL Z INIT (3);`	`z=3`
`x=10; y=20;`	`x=10; y=20`
`GSLINE(x,y);`	`'GSLINE .x .y'`
`GSLINE(30,z);`	`'GSLINE 30 .z'`
`%INCLUDE ADMUPINA`	`/* remove */`
`CALL FSTERM;`	`Address command 'GDDMREXX TERM'`

Specifying call parameters made easy by GDDM-REXX

There are three main reason why coding GDDM call parameters is easier for GDDM-REXX than for other languages. In GDDM-REXX programs you can:

1. Omit the lengths and counts for strings and arrays of values passed to GDDM – you can use dots instead.
2. Use dots for returned values you do not care about.
3. Specify array parameters by listing them element by element in the call.

Omitting parameters

Use the interactive ERXTRY EXEC to experiment with omitting parameters. These examples will work:

```
'GSCHAR 50 50 . "Hello"'

'ASDFLD 1 10 17 1 20 2'
'ASCPUT 1 . "Hello"'
```

However, if you omit the length from ASCGET, you will get an error message. GDDM requires that lengths of returned values must be specified, so they cannot be omitted for returned values. So this is wrong:

```
'ASCGET 1 . .var'
```

You can experiment with omitting returned values in the same way. From ERXTRY, try the call

```
'ASREAD .type . .'
```

and then the REXX statement

```
Say 'The key you pressed was PF'type
```

Coding calls that take arrays as parameters

The important points about specifying array parameters are:

Using stemmed variables: Try experimenting with any call that takes a one-dimensional array.

```
procopts.1=1000
procopts.2=2
'DSOPEN 9 1 * 2 .procopts. . ()'
```

Then try further calls with two-dimensional arrays (see also “Defining stemmed variables for columns of two-dimensional arrays”).

```
'ASQFLD 2 3 4 .array.'             /* sets values in array.1.1       */
                                   /* through array.3.4              */
```

Enumerating array values within the call: If you use the GSPLNE call to draw a series of straight lines, the benefit of coding directly into the parameter string becomes apparent. Try:

```
x1=10
/*     cnt1  float.cnt1    float.cnt1     - syntax from ERXPROTO         */
'GSPLNE 3    (.x1 20 30)   (40 50 60)'
```

If you specify an explicit count value for an array, the array is truncated or expanded with zeroes to match. For example:

```
'GSPLNE 3    (5 6 7 8)     (6)'
```

is passed to GDDM as

```
'GSPLNE 3    (5 6 7)       (6 0 0)'
```

Replacing dimension information with dots: Because GDDM-REXX can count the numbers in the parentheses, you can replace the count with a dot:

```
'GSPLNE .    (.x1 20 30)   (40 50 60)'
```

Defining stemmed variables for columns of two-dimensional arrays: Some GDDM calls require parameters in the form of a two-dimensional array. For example, ASDFMT allows a number of alphanumeric fields to be defined; the parameters are a count of the number of fields, a count of the number of elements being defined for each field, and an array that contains the element values. This call defines two fields with five elements provided for each field:

```
'ASDFMT 2 5 ((1 1 1 80 1) (2 2 1 80 1))'
```

Many programs need to create a number of similar alphanumeric fields, one below another. You can use single values for columns in multi-dimensional arrays, as in this example of `ASDFMT`, to set up ten fields numbered 1 to 10 at the top of the screen:

```
Do i= 1 to 10                      /* set up array with values 1-10  */
   nums.i=i
End i
/*    fields  count    ids    row   col depth width  type           */
'ASDFMT 10      6     (.nums. .nums.  1  1      80     1  )'
/* Fields 1 to 10 are set up in rows 1 to 10 respectively.  Each     */
/* field starts in column 1, is one row deep, eighty characters wide */
/* and has the field type of 1 (alphanumeric output, numeric input)  */
```

Avoiding problems with GDDM-REXX parameters

If you code parameters incorrectly, particularly array parameters, GDDM-REXX will normally give you an error message. GDDM-REXX error messages all take the form ERXnnnn. If you code GDDM call parameters in your REXX programs according to these hints, you won't get many error messages referring to parameters.

- Ensure that you pass arrays with the correct number of dimensions to GDDM calls that require them. For such calls, ERXPROTO indicates whether the arrays are to be of one or two dimensions.
- Always put GDDM calls in single quotes, and put double quotes around character strings within the calls. Some GDDM calls work without quotes, but it is safer to use them on every call.
- When specifying negative values for parameters, enclose the minus sign within quotes or express the value –n in the form 0–n. Otherwise, the sign could be interpreted as an infix minus.
- Pass character strings in uppercase, where they refer to the names of symbol sets, mapgroups, or GDDM objects.

Finding errors in GDDM-REXX programs

Sometimes, GDDM may interpret incorrectly specified parameters in unexpected ways, without giving an error message. If you get unexpected results, try tracing. Put `'GXSET TRACE ON'` before the call that is giving trouble, and `'GXSET TRACE OFF'` after it.

The trace will show the values GDDM-REXX got, and the values it sent on to GDDM.

More complex programming with GDDM-REXX

This section describes special actions that you may need to take when writing more complex GDDM function in REXX programs.

Multiple instances of GDDM and GDDM-REXX

You can have multiple instances of GDDM and of GDDM-REXX. This facility can be used to create programs that run a number of independent applications, each with its own environment.

Multiple instances of GDDM and GDDM-REXX can be controlled separately. You can have multiple instances of GDDM within one instance of GDDM-REXX. You can also have multiple instances of GDDM-REXX.

Instances of GDDM: Instances of GDDM are controlled by GDDM-REXX, using the reentrant interface to GDDM. GDDM-REXX allows simplified access to this by the `GXGET AAB` and `GXSET AAB` subcommands.

An application can jump between instances by using a `GXSET AAB` subcommand with the application anchor block (AAB) of the instance. Instances are chained together such that the default instance is always the first in the chain (and cannot be terminated by FSTERM). When a new instance is created, it is added to the end of the chain and becomes the current instance. If an instance is terminated by an FSTERM call, the chain is remade, and control returns to the previous instance in the chain.

Here is a simple EXEC that demonstrates GDDM instances and lets you move between them by using the PF keys. The variable `v` in the ASREAD call is returned with the number of a PF key. If you press anything other than PF1, 2 or 3, you will leave the EXEC.

```
/* REXX EXEC to demonstrate multiple instances of GDDM              */
Address command 'GDDMREXX INIT'    /* Initialize GDDM-REXX          */
Address gddm                       /*  under CMS                    */
Do i = 1 to 3                      /* set up 3 instances of GDDM    */
   'GSCOL .i'                      /* draw a colored line           */
   'GSLINE 50 50'
   'GXGET AAB .name.i'             /* extract the AAB               */
   'FSINIT'                        /* start another instance        */
End i
'GSCHAR 15 50 . "PF1 2 or 3 to select GDDM instance 1, 2, or 3"'
'GSCHAR 15 40 . "Any other PF key to end"'
'ASREAD .a .v .'                   /* V gets number of PF key       */
Do forever
   If a¬=1|v>3 then                /* leave if not PF 1, 2, or 3    */
      Leave
   'GXSET AAB .name.v'             /* select an instance            */
   string= 'this one is' v
   'GSCHAR 40 50 . .string'
   'ASREAD . .v .'
End                                /* do forever                    */
Address command 'GDDMREXX TERM'    /* Terminate under CMS           */
Exit
```

Instances of GDDM-REXX: Instances of GDDM-REXX are controlled by the GDDMREXX command. `GDDMREXX INIT` starts a new instance; `GDDMREXX TERM` terminates the latest instance. It is not possible to jump between instances. Only the latest instance can be used, and when that is terminated the previous one will be activated – they are on a push-down stack.

You can experiment with instances of GDDM-REXX using the EXEC shown above. Simply place an outer loop around the whole EXEC. Notice, however, that you must use an `Address command` or an `Address link` statement before you use a `GDDMREXX` command.

Termination: Proper termination of instances of both GDDM and GDDM-REXX is important. All instances of GDDM that are active are properly terminated by a `GDDMREXX TERM` command. However, it is important that each instance of GDDM-REXX is terminated at any possible exit including error and abnormal exits.

Failure to terminate GDDM-REXX can result in storage being used up progressively. A programming technique for avoiding this is illustrated in Figure 5.

On VM systems, GDDM-REXX is automatically terminated on return to CMS when you get the ready message; however, if you are working from FILELIST or a similar program that does not return to CMS command ready, it is not terminated. If you suspect that one or more instances of GDDM-REXX are still active, you can terminate them all with the `GDDMREXX TERM (ALL)` command, or check using the NUCXMAP command for entries starting with `ERX` (possibly with a preceding blank).

Terminating GDDM-REXX when your program is ended abnormally

By including a special error-exit routine in your GDDM-REXX exec, such as the one in Figure 5, you can ensure that your program terminates GDDM-REXX properly every time.

```
/* REXX comment                                                  */
 Signal on Syntax                      /* Set up error handling     */
 Signal on Halt                        /*                           */
 Address link 'GDDMREXX INIT'          /* Initialize GDDM-REXX      */
                                       /*  under MVS or TSO         */
/*  GDDMREXX processing here                                      */

 Signal Endit

/*  More GDDMREXX processing here                                 */

 Error:
 Syntax:
 Halt:
 Endit:
  Address link 'GDDMREXX TERM (ALL'    /* Terminate GDDM-REXX & GDDM */
                                       /*  under MVS or TSO         */

/* REXX processing only here - no GDDM                            */

  Exit
```

Figure 5. Routine to terminate all instances of GDDM-REXX when run ends abnormally

Invoking GDDM-REXX programs from other programs or from CMS subset

If you intend to use GDDM-REXX EXECs in subset mode, or by calling them from other programs, you should first load the GDDMREXX command module in the nucleus. This can be done with the command `NUCXLOAD GDDMREXX`. The module is not large and the command can be included in a PROFILE EXEC. The reason you may need to do this is that the first time you use GDDM-REXX, it loads the module at address `X'20000'` unless that module is already in the nucleus. (GDDMREXX then loads itself into the nucleus for subsequent use.) This may interfere with the use of that part of virtual storage by other programs. When you have finished using your GDDM-REXX EXECs, you can issue the `NUCXDROP` command to free nucleus storage.

Coding styles—strict or loose syntax

With GDDM-REXX, you have a choice of coding styles. If you are coding your own private EXECs, you can minimize the time spent in typing by replacing parameters with dots and putting array and other values directly into the GDDM calls. In fact you can go further, and provided you understand REXX's rules, you can sometimes leave out the quotes around GDDM calls. However, be aware of the potential difficulties when doing this; REXX may attempt substitution of values in place of names, and negative values may cause REXX to attempt subtraction.

If you are producing prototype code that will later be recoded in another language, all parameters should be passed by name:

```
count=3
xarray.1=20; xarray.2=30; xarray.3=40;
yarray.1=40; yarray.2=50; xarray.3=60;
'GSPLNE .count .xarray. .yarray.'
```

and not:

```
'GSPLNE . (20 30 40) (40 50 60)'
```

This will minimize the difficulties of conversion. The main remaining problem will be producing the data declarations for the target language. See "Converting PL/I examples to GDDM-REXX" on page 18 for guidance on PL/I equivalents.

Chapter 2. Drawing graphics pictures

In “Concepts introduced by the HOUSE program” on page 7 the basic concepts of the graphics primitive and the graphic attribute were introduced. With the aid of another programming example, this section demonstrates how you can use GDDM’s graphics calls to draw graphics primitives and to specify attributes affecting their appearance.

Note: You can find detailed information about the syntax and parameter values of each call described in this section in the *GDDM Base Application Programming Reference* book.

Example: Program that draws a street map

The program in Figure 6 draws a picture showing the center of a town. Features such as a river, the streets, and a fountain in the square are drawn by combining various graphics primitives.

```
TOWN:   PROC OPTIONS(MAIN);

DCL(TYPE, MOD, COUNT) FIXED BIN(31);  /*Parameters for ASREAD */

DCL MXVALS(16) FLOAT DEC(6)                                      A
  INIT(70,70,80,100,100, 80, 70, 70, 50,50,40, 0, 0,40,50,50);

DCL MYVALS(16) FLOAT DEC(6)                                      A
  INIT( 0,40,50, 50, 70, 70, 80,100,100,80,70,70,50,50,40, 0);

DCL S1XVALS(4) FLOAT DEC(6)  INIT(0,   0, 50, 50);
DCL S1YVALS(4) FLOAT DEC(6)  INIT(95, 90, 90, 95);

DCL S2XVALS(5) FLOAT DEC(6)  INIT(20, 20, 25, 25, 20);
DCL S2YVALS(5) FLOAT DEC(6)  INIT(75, 70, 70, 75, 90);

DCL(X1, X2, Y1) FLOAT DEC(6);

CALL FSINIT;                    /* Initialize GDDM           */
CALL GSUWIN(0,100,0,100);       /* Open a uniform window     */   B
```

Figure 6 (Part 1 of 3). Program using GDDM graphics API calls to draw a map of a town

© Copyright IBM Corp. 1982, 1996

```
/*&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.¯*/
/* DRAW RIVER                                                        */
/*_________________________________________________________________*/
CALL GSSEG(1);                  /* Open a named segment.      */
CALL GSLW(2);                   /* Use lines of double width  */  C
CALL GSLT(7);                   /* Use unbroken lines         */  D
CALL GSCOL(1);                  /* Use blue as current color  */  E
CALL GSMOVE(0, 72);             /* Move current position      */

CALL GSAREA(1);                 /* Start a graphics area      */  F
   CALL GSCOL(7);               /* Outline area in white      */
   CALL GSELPS(-30, 12, -12,  20, 75);  /* Draw an ellipse    */
   CALL GSARC(17, 100, 85);     /* Arc through 85°            */  G
   CALL GSQCP(X1, Y1);          /* Query the current position */  H
     X1 = X1 - 12;              /* Specify a point 12 x-units */
     CALL GSLINE(X1, Y1);       /* away and draw a line to it */
     CALL GSARC(17, 100, -100); /* Clockwise arc through 100° */
     CALL GSELPS(30, 12, -12,  0, 85);                            I
CALL GSENDA;                    /* End drawing and shade area */  J
CALL GSSCLS;                    /* Close the current segment  */

/*&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.¯*/
/* DRAW STREETS                                                      */
/*_________________________________________________________________*/
CALL GSSEG(3);                  /* Open a new named segment   */
CALL GSCOL(7);                  /* Use white as current color */
CALL GSMIX(2);                  /* Streets will overlap river */  K
CALL GSLW(2);                   /* Use lines of double width  */
/* DRAW MAIN STREETS */
CALL GSAREA(0);                 /* Begin drawing an area      */
CALL GSMOVE(50, 0);
CALL GSPLNE(16, MXVALS, MYVALS); /* Draw the main streets as a*/  L
                                  /* polyline through 16 points*/
```

Figure 6 (Part 2 of 3). Program using GDDM graphics API calls to draw a map of a town

```
/*ADD SIDE STREETS  AS PART OF THE SAME AREA */
CALL GSMOVE(50, 95);
CALL GSPLNE(4, S1XVALS, S1YVALS); /* Polyline joining 4 points*/
/*next street */
CALL GSMOVE(15, 90);
CALL GSPLNE(5, S2XVALS, S2YVALS); /* Polyline joining 5 points*/
/*next street */
CALL GSMOVE(15, 50);            /* New current position 15,50 */   M
   CALL GSLINE(15, 40);         /* Draw line to point 15,40   */   N
   CALL GSARC(40, 40, 75);      /* Draw counterclockwise arc  */
   CALL GSLINE(50, 10);
   CALL GSLINE(50, 16);
   CALL GSLINE(36, 21);
   CALL GSARC(40, 40, -75);     /* Draw clockwise arc         */
   CALL GSLINE(20.75, 50);
CALL GSENDA;                    /* Finish drawing streets and */   O
                                /* shade them in white        */

/*&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.&v.¯*/
/* DRAW FOUNTAIN                                                  */
/*_______________________________________________________________*/
CALL GSFLW(1.8);                /* Use lines 1.8x normal width */  P
CALL GSBMIX(2);                 /* Background takes precedence */  Q
CALL GSPAT(2);                  /* Use shading pattern 2 to    */  R
CALL GSAREA(1);                 /* shade the completed area    */
  CALL GSMOVE(52, 60);
  CALL GSCOL(8);                /* Use black to outline area   */
  CALL GSARC(60, 60, 360);      /* Circle, with center 60,60   */  S
CALL GSENDA;
CALL GSPAT(3);                  /* Use shading pattern 3 and   */
CALL GSCOL(1);                  /* color blue to shade area    */
CALL GSMOVE(55, 60);            /* when it is complete         */
CALL GSAREA(1);
  CALL GSCOL(7);
  CALL GSARC(60, 60, 360);                                         S
  CALL GSENDA;
CALL GSMS(6);               /* Select marker for fountainhead */   T
CALL GSMARK(60, 60);        /* Mark the center of fountain    */   U
CALL GSSCLS;                /* Close graphics segment         */

CALL ASREAD(TYPE,MOD,COUNT); /* Send out all the graphics      */
CALL FSTERM;                /* Terminate GDDM                 */
%INCLUDE ADMUPINA;          /* GDDM Entry declarations        */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
```

Figure 6 (Part 3 of 3). Program using GDDM graphics API calls to draw a map of a town

Drawing graphics primitives

The TOWN program uses many of the API calls that GDDM provides for drawing graphics primitives. The program is used to illustrate some important points about the calls and the primitives to which they relate in the sections that follow.

Setting up a coordinate system for drawing graphics

Using GSUWIN, shown at **B** in the program, you can set up a coordinate system for drawing graphics primitives, in which 1 x-unit is equal to 1 y-unit. When you add a graphics primitive to your picture, you specify locations, such as the end of a line or the center of an arc, in terms of graphics window coordinates. These are also known as **world coordinates**.

It is good practice to issue a GSUWIN call before opening any graphics segments or drawing any graphics. This ensures that squares appear square and circles appear round and that any primitives drawn by your program will have the same proportions and shape on different devices.

Although the GSUWIN call at **B** defines a uniform graphics window of 0 to 100 on both axes, points outside this range can also be addressed. The values specified on the GSUWIN call are a minimum range for the coordinate system.

There is more information on the concept of the uniform graphics window in “Uniform world coordinates” on page 118.

If you do not define a graphics window in your program, the default **graphics window** of 100 units by 100 units is used. An x-unit of the default graphics window is equal to the width of the device's hardware cells, and a y-unit is equal to the cell height. Very few devices have cells of equal height and width, so the x-units and y-units of the default graphics window are rarely the same size.

Points outside the range of the default graphics window cannot be addressed by your program. For more information about the graphics window, see “The graphics window” on page 117.

Moving the current position, using GSMOVE

At **M** in the program, (and at several other places too), the current position is moved to a specified position. The current position is taken as the starting point for any primitive that you draw. When your application opens a graphics window, the current position is the origin by default and, when you draw a primitive, the current position becomes the last point drawn in that primitive.

You will need to use the GSMOVE call in programs whenever you do not want your first primitive to be drawn from the origin, or whenever you don’t want the end point of the previous primitive to be the starting point of the next.

Drawing a line, using GSLINE

By specifying an end point for a line on the GSLINE call, you can draw a line from the current position to that point. At **N** in the example, the point (15,40) is specified as the end point of a line. The line produced by this call extends from the current position, (15,50) to (15,40).

The visual characteristics of this line are determined by attributes, which are set by calls discussed in “Specifying graphics attributes for primitives” on page 35.

Drawing a series of straight lines, using GSPLNE

The main streets drawn in the TOWN program consist of a series of straight lines joining 16 points. Rather than issue a long series of GSLINE calls, the program uses the GSPLNE call at **L** to draw a **polyline** through all 16 points at once. The points are passed to the GSPLNE call in two arrays, which are declared at the places marked **A**. One array holds all the x-coordinates of the points, and the other holds the corresponding y-coordinates.

Drawing a circular arc, using GSARC

At **G** in the example, the GSARC call is used to draw a curve in the bank of the river. The current position after the previous call is (20,75), and this is taken as the starting point of the arc. The GSARC call specifies a center point for the arc of (17,100) and an angle of sweep of 85 degrees. GDDM determines the radius of the arc by taking the distance from the specified center point to the current position. It then draws an arc in a counterclockwise direction from the current position through 85 degrees. (To draw a clockwise arc, use a negative angle of sweep.)

One difficulty with using this call is determining the current position after it is issued. If you draw an arc through 90, 180, or 360 degrees, the new current position is easily determined but, for other angles, this becomes more difficult. In such situations, you can use a special call provided by GDDM for determining the current position (see “Querying the current position, using GSQPOS” on page 32).

You can draw an entire circle with the GSARC call by specifying 360 degrees as the angle of sweep. This is used at the places marked **S** in the program to draw the fountain.

Two other GDDM calls you can use to draw curved lines are discussed below.

Drawing an elliptic arc, using GSELPS

Unlike curved lines drawn using GSARC, the elliptic arc drawn at **I** in the program has its end point specified explicitly by the last two parameters of the GSELPS call. From the previous current position, it draws a portion of an ellipse to the point (0,85). This ellipse has a major axis of 30 units, tilted at an angle of 12 degrees to the horizontal and a minor axis of 12 units.

The values specified for the two axes both carry the same sign. This causes the elliptic arc to be drawn in a counterclockwise direction. If a negative value is specified for one axis and a positive value for the other, the resulting elliptic arc is drawn in a clockwise direction.

The longest elliptic arc you can draw is half an ellipse.

Drawing a series of curved lines, using GSPFLT

The third way of using GDDM to draw curved lines in a graphics program is to use the GSPFLT call, which draws a **polyfillet**. Just as a polyline draws straight lines between a number of specified points, a polyfillet draws curved lines between specified points. There isn’t an example of a GSPFLT call in the TOWN program, but it has a format like that of the GSPLNE call.

If you specify the same two-dimensional array for a GSPFLT call as for a GSPLNE call, such as the one at **L**, a series of curved lines is drawn tangential to the midpoints of each line of the polyline and ending at the last point specified.

The algorithm used by the GSPLFT call is explained fully in the *GDDM Base Application Programming Reference* book.

Drawing a graphics marker symbol, using GSMARK

At **U** in the example, a graphics marker is drawn at the center of the circular arc representing the fountain. This marker represents the fountain head from which the water comes. The appearance of the marker is determined by attributes, which are dealt with in “Specifying graphics attributes for primitives” on page 35.

When a point is marked by a graphics marker, it becomes the current position.

If you want to draw a whole series of markers in your picture, you can use the GSMRKS call.

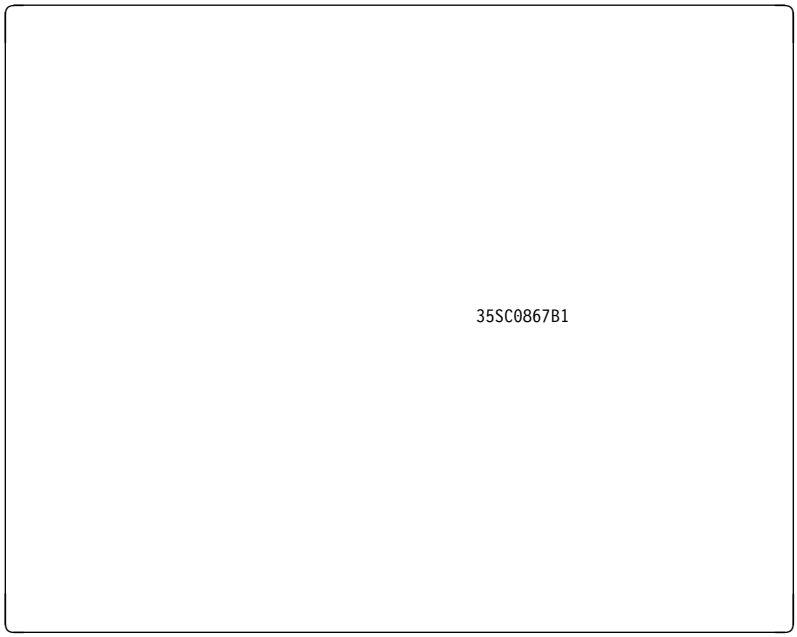


Figure 7. Output from the TOWN program

Drawing a shaded graphics area, using GSAREA and GSENDA

If you want a shape drawn by your program to be shaded, you need to specify this before issuing any of the calls to draw it. At **F**, the program issues a GSAREA call before issuing any of the calls to draw the primitives that make up the shape of the river. A GSENDA call is issued when the outline of the river is finished at the point marked **J**. This automatically instructs GDDM to shade the area subject to the graphics attributes that are current at that point in the program.

Outline of a graphics area

The parameter specified on the GSAREA call determines whether the outline of a graphics area is visible or not. The GSAREA call at **F** has a parameter value of 1, which causes the river to be outlined in the picture. The GSAREA call for the multipart area representing the streets has a parameter value of 0, which specifies that the outline is not to be shown.

Closure of a graphics area

A graphics area must be closed before it is shaded. If the current position after the last primitive is drawn is not the same as the current position when the area was opened, GDDM closes the area by drawing a line to the original current position. This happens at **J** in the program. The current position after the last primitive drawn is 0,85, and a line is drawn to the initial current position (0,72) before the area is shaded.

The graphics area shading algorithm

Whether a portion of an area is shaded or not depends on the number of lines you need to cross to move from that portion to outside the area. If the number of lines crossed is odd, the portion is shaded. If the number of lines crossed is even, the portion is left unshaded. Figure 8 shows the algorithm that is used to shade graphic areas.

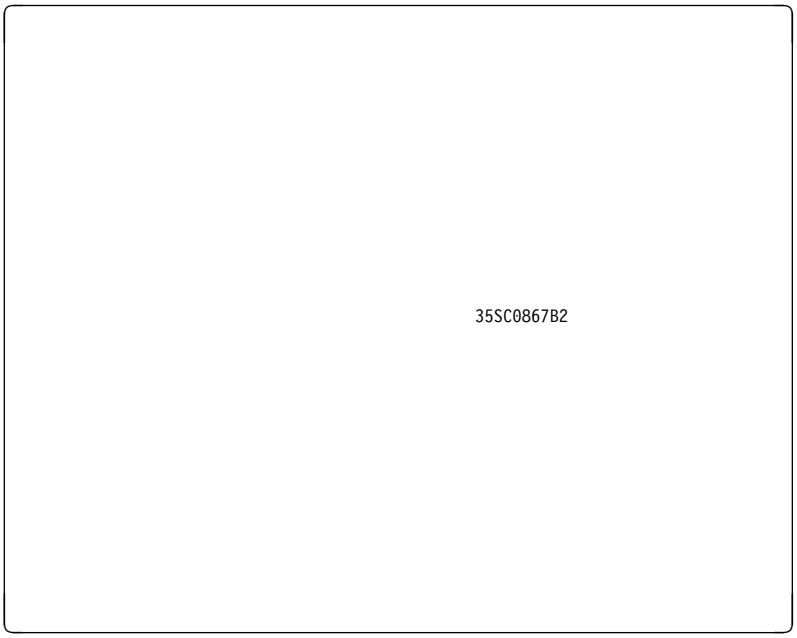


Figure 8. How graphics areas are shaded

Drawing a multipart graphics area, using GSMOVE

In the TOWN program all of the streets are drawn as one graphics area. When the primitives for drawing each street are complete, a GSMOVE call moves the current position to the point where the drawing of the next street is to begin. Because they represent streets, these parts of the area need to be touching, but another split area could be used to draw detached buildings in the picture.

Note: It is good practice, when drawing a split area, to close each part of it explicitly, unless you want GDDM to join the last part drawn to the first.

Querying the current position, using GSQPOS

After the GSARC call at the place marked **G**, the current position is at the end point of the arc. Because the arc is not drawn to a specified end point, the current position is not known. This would not be a problem, if the program needed to draw the next primitive at some other point. (A GSMOVE call could take the current position to that point and drawing could recommence.) However, the next primitive needs to be a horizontal line following on from the end point of the arc, so the current position must be determined.

At **H** in Figure 6 on page 25, the GSQCP call queries the x and y coordinates of the current position and assigns them into the variables `X1` and `Y1`. Only the x coordinate need then be changed to be used as the end point of a horizontal line.

After this call, the current position is still unknown, but it is not needed for the GSARC call drawing an arc concentric with the first one.

GSQCP, which you can use to determine the current position, is one of the many very useful query calls provided by the GDDM API. They are denoted by the letter “Q” in their call names. Some of the others are described in later sections of this book. All query calls are described in detail in the *GDDM Base Application Programming Reference* book.

Drawing graphics image pictures, using GSIMG

The GSIMG call enables you to declare a pattern of dots within the program and then add the pattern to the current page's graphics, as you would any other primitive. Each dot of the pattern is represented by a **pixel** (also referred to as a pel) which can be either switched on or switched off.

Because the size and aspect ratio of a pixel varies from one type of device to another, the size and aspect ratio of the image may vary if the program is transferred between devices.

This is an example of an image pattern declaration and a GSIMG call that adds it to the GDDM graphics field:

```
DCL SPIDER CHAR(198);
UNSPEC(SPIDER)=
        '000000000000000000000000000000000000000000000000'B||
        '000000000000000000000000010000000000000000000000'B||
        '000000000000100000000000010000000000000000000000'B||
        '000000000000010000000000010000000000000000000000'B||
        '000000000000010000000000100000000000000000000000'B||
        '000000000000010000000010000000000000000000000000'B||
        '000000000000010000000100000000000000000000000000'B||
        '000000000000010000001000000000000000000000000000'B||
        '000000000000010000010000000000000000000000000000'B||
        '000000000000010000010000000000000000000000000000'B||
        '000000000000010000010000001100000000000000000000'B||
        '000000000000010000010000111110000000000001100000'B||
        '000000000000000100001000011111000000000010000000'B||
        '000000000000000100001000011111000000000001100000'B||
        '100000000000000010001111111110000000010000000000'B||
        '011111111100000010111111110000000001100000000000'B||
        '000000000011000011111111110000000010000000000000'B||
        '000000000000110111111111111111111100000000000000'B||
        '000000000000001111111111110000000000000000000000'B||
        '000000000000001111111111110000000000000000000000'B||
        '000000000000001111111111110000000000000000000000'B||
        '000000001111111111111111101110000000000000000000'B||
        '000000110000000111111110000001110000000000000000'B||
        '000011000000000011111010000000001111111111000000'B||
        '011100000000000000100001100000000000000000100000'B||
        '000000000000000000100000010000000000000000000000'B||
        '000000000000000000100000001000000000000000000000'B||
        '000000000000000000100000001000000000000000000000'B||
        '000000000000000000100000001000000000000000000000'B||
        '000000000000000000100000001000000000000000000000'B||
        '000000000000000000100000001000000000000000000000'B||
        '000000000000000000100000000100000000000000000000'B||
        '000000000000000000100000000010000000000000000000'B||
        '000000000000000000100000000001000000000000000000'B||
        '000000000000000000100000000000100000000000000000'B;

/*       TYPE WIDTH DEPTH  LENGTH   NAME                              */
CALL GSIMG(0,  43,    33,   198,   SPIDER); /* Send image of spider    */
                                            /* to the current position */
```

Although the width of the image in the example is only 43 pixels, it must be padded out to 48 pixels per row in the declaration. Images created in this way can be up to 2040 pixels wide but the image data must have each row padded to a multiple of 8 bits.

The top left-hand corner of the image is placed at the current position. Any bits set to 1 cause the corresponding dots on the screen to be set on. The image is shown in the current color, but it is always monochrome. To obtain multicolor images you must overlay images of different colors.

Figure 9 on page 34 shows output from several GSIMG calls, similar to the one above. Note that images appear black, when placed on a shaded background.

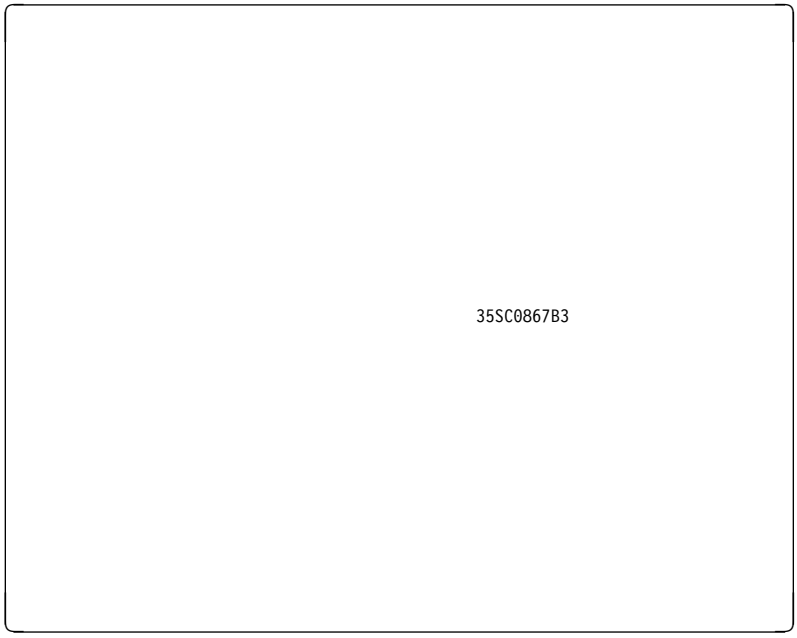


Figure 9. Output from GSIMG calls

Note: GDDM's image-processing functions are dealt with in Chapter 6, “Image basics” on page 85 and Chapter 17, “Using GDDM’s advanced image functions” on page 339 of this book.

Drawing a scaled image picture, using GSIMGS

The GSIMGS call also enables you to draw an image as a graphics primitive and to increase the scale of it when you add it to the graphics field. This is an example of the call:

```
/*        TYPE  WIDTH DEPTH  LENGTH   NAME   X-SIZE  Y-SIZE     */
CALL GSIMGS(0,   43,   33,     98,   SPIDER,  30.0,   20.0);
                                     /* Fit spider image into a */
                                     /* box 30 world-coordinate */
                                     /* units by 20             */
```

The first five parameters have the same meaning as in the GSIMG call. The last two parameters define a box, called an **image window**, in world-coordinate units. GDDM fits the image into the image window by displaying each bit in the character variable as a rectangular array of dots, rather than as a single dot. The number of dots in the array is such that the image is the largest possible one that does not overflow the image window. The top left-hand corner of the image window is placed at the current position.

Because the array need not be a square, the horizontal and vertical dimensions are scaled separately. The mechanism allows images to be increased in scale by a whole number but never to be scaled down. If an image requires a scaling factor of less than one to fit in the image window, it is displayed using a factor of one, and is allowed to overflow the image window.

Another method of presenting image data (using an image symbol set) is described in Chapter 4, “Creating graphics-text output in your application” on page 57.

Specifying graphics attributes for primitives

The TOWN program in Figure 6 on page 25 uses a number of API calls that specify attributes for graphics primitives. These attributes determine how primitives appear and how they behave in relation to other primitives. The program is used to illustrate some important points about the calls and the attributes to which they relate in the sections that follow.

Note: You can find detailed information about the syntax and parameter values of each of the calls described in this section in the *GDDM Base Application Programming Reference* book.

You can change a particular attribute at any stage of your program. All primitives drawn subsequently assume the new attribute value.

Attributes can be grouped together in graphics segments. If your program uses segments, a call to change an attribute only takes effect in the segment in which that call is issued. When a new segment is opened, the attributes return to their default settings. In the following sections, the defaults quoted are those initially supplied by GDDM at the start of a program.

Note: You can change the default attribute settings in your program to defaults of your own choosing. See “Overriding the standard default of a graphics attribute” on page 47 for details.

Setting the current color, using GSCOL

The current color affects the appearance of all graphics output – lines, arcs, areas, markers, graphics text, and graphics images. The GSCOL call at **E** in the TOWN example has a parameter setting of 1. This sets the current color to blue. The area drawn after that, representing the river, is shaded in blue. (The area's outline, however is, white.)

Each numerical value specified on the GSCOL call represents a different color. The range of numbers you can specify depends on the color support of the device on which you expect the program to run. Most display devices support these ten basic values:♦

-2	White	4	Green
-1	Black	5	Turquoise (cyan)
0	Default (green)	6	Yellow
1	Blue	7	Neutral
2	Red	8	Background
3	Pink (magenta)		

The codes 1 through 8 are used to specify the same colors in other calls too.

Color 8, background, is a special color. You can find out more about its uses in “Special treatment of the background color, using call GSMIX” on page 42 and “Setting the background-mix attribute, using GSBMIX” on page 44.

♦ A suggested mnemonic for remembering the codes for the colors blue through neutral is: Boys Reading Politics Go To Yale Now

Information about what happens if the device does not support the chosen color is given in the *GDDM Base Application Programming Reference* book.

Setting a new current line type, using GSLT

At **D** in the TOWN program, the GSLT call is used to specify the type of line that is to be used to draw primitives in the first segment. The value 7 specifies a solid line. For details of the other eight values that you can pass with this call and the line types they produce, see the *GDDM Base Application Programming Reference* book.

Setting a new current line width, using GSLW or GSFLW

Another call affecting the appearance of lines is issued at **C** in the example. This GSLW call specifies that lines twice the thickness of normal lines are to be used for drawing primitives in that segment. The parameter of GSLW specifies a factor by which the standard width for the current device is multiplied.

If you don't specify the line width explicitly, lines of the standard width are used.

On display devices, lines of standard width are 1 pixel wide and the only other available thickness, two pixels, is supported by only a few displays. You can specify wider lines but they will not look wide unless printed on a suitable printer.

High-resolution devices, such as page printers, have much smaller pixels. The standard line on such devices is typically 6 pixels wide and they can print lines of up to 600 pixels wide.

At the place marked **P**, the GSFLW call is used to specify that a multiplier of 1.8 is to be applied to the standard width of lines in the current segment. The resultant line width is rounded down to the nearest whole number of pixels. On a device with a standard line width of 6 pixels, this multiplier would cause lines with a width of 10 pixels to be drawn.

Setting the current marker symbol, using GSMS

The center of the fountain drawn in the TOWN program is marked by a graphics marker in the form of an 8-point star. Before the GSMARK call is issued, the GSMS call at **T** specifies the value 6, which selects the 8-point star as the current graphics marker symbol.

There are 10 graphics marker symbols provided by GDDM, each denoted by a different parameter value in the range 1 through 10. These symbols, known as system markers, are illustrated in the description of the GSMS call in the *GDDM Base Application Programming Reference* book.

You can also specify symbols that you have created yourself, (using the Image Symbol Editor or Vector Symbol Editor), for use as the current graphics marker. To do this, you must first load your set of symbols using the GSLSS call, and then specify a value in the range 65 through 254 on the GSMS call. Here is an example:

```
CALL GSLSS(4,'NEWMARKS',0); /* Load user marker set called NEWMARKS  */

CALL GSMS(72);              /* Set type of current marker symbol to  */
                            /* to symbol 72 (X'48') in the currently */
                            /* loaded symbol set                     */

CALL GSMARK(50,50);         /* Draw user marker 72 centered at 50,50 */
```

You can find more information on the GSLSS call in “Loading symbol sets for graphics text” on page 235 and in the *GDDM Base Application Programming Reference* book.

Note: If symbols in your marker set are multicolored, you must set the current color to neutral, GSCOL(7), before drawing any of the multicolored markers in the picture.

Changing the scale of a graphics marker symbol, using GSMB

If the marker symbol you are using comes from a vector symbol set, you can enlarge or reduce its size using the GSMB call.

Vector markers expand to fill the marker boxes that contain them, so by increasing the size of the marker box, you can increase the size of the markers. You can specify the dimensions of the marker box in world coordinates using the GSMB call.

This call has no effect on markers from image symbol sets. They are always displayed at a size defined by the symbol itself.

Setting the current shading pattern, using GSPAT

The scheme for shading patterns is similar to that for markers. There are 16 system patterns, and users may also create their own patterns using the Image Symbol Editor (but not the Vector Symbol Editor), and then specify them for use. At **R** in the TOWN program, the GSPAT call is used to select a shading pattern for the path around the fountain.

All subsequently drawn areas are shaded in this pattern until a new shading pattern is specified. Values in the following ranges can be specified on the call parameter:

0	Default (initially solid on displays, half-tone on printers)
1 through 16	GDDM system-defined patterns
65 through 254	User-defined patterns

The available system patterns for displays are shown in Figure 10 on page 38.

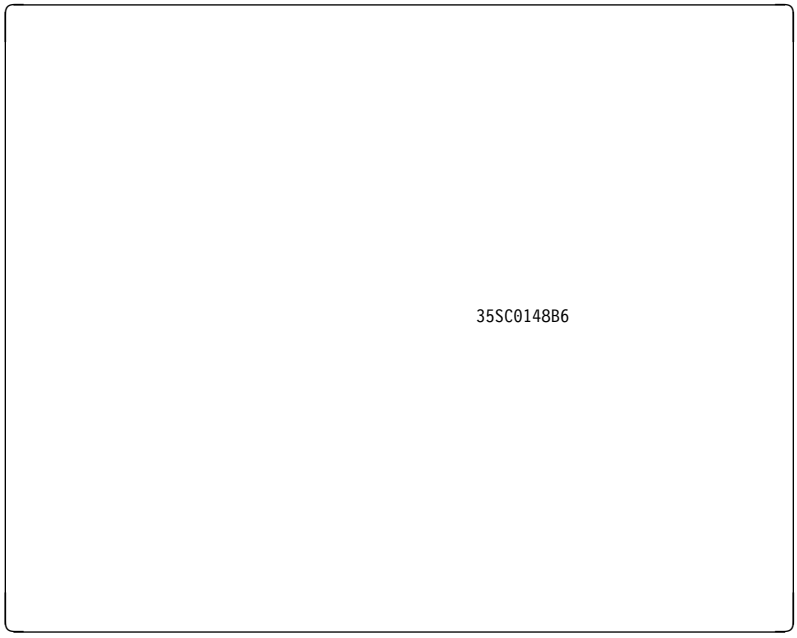


Figure 10. The 16 GDDM system shading patterns

Using shading patterns other than the GDDM system patterns

A user pattern set can be either a GDDM-supplied one or one that you have created yourself using the GDDM Image Symbol Editor. Such pattern sets should be designed to match the width and depth in pixels required by the device.

Some GDDM-supplied pattern sets are shown in Figure 11 on page 39 and in Figure 12 on page 40.

Just as you load a symbol set containing graphics marker symbols, you can load a pattern set other than the system pattern set using the GSLSS call. You then specify a pattern in the range 65 through 254 on the GSPAT call. If you specify a position in the symbol set at which no pattern has been created, subsequent areas are left unshaded.

You are allowed to load only one user pattern set at a time. You can then use either a pattern from the loaded set or one of the 16 system patterns.

Several sample user pattern sets are supplied with the GDDM package. One of them, the geometric pattern set, is shown in Figure 11 on page 39.

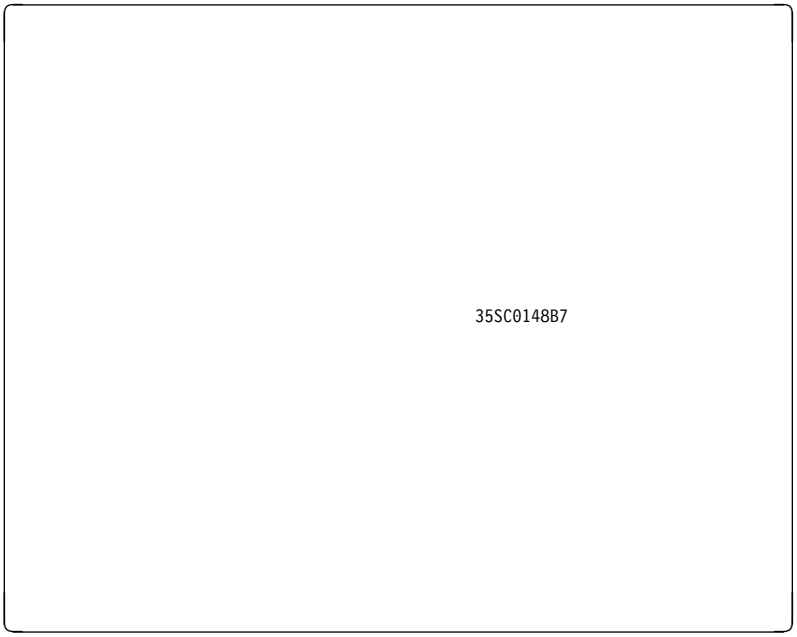


Figure 11. GDDM geometric pattern set - ADMPATTC

All the GDDM sample pattern sets are listed in the *GDDM Base Application Programming Reference* book.

Selecting from a wider range of colors, using GSPAT

The GDDM-supplied symbol sets ADMCOLSD, ADMCOLSN, and ADMCOLSR enable you to shade your areas in any of 64 different colors. These colors are shown in Figure 12 on page 40. The three sets differ only in the size of the symbols.

The chosen color is specified with a GSPAT call:

```
CALL GSLSS(3,'ADMCOLSD',0); /* Load GDDM-supplied              */
                            /* 64-color pattern set.           */
CALL GSCOL(7);              /* Set current color to neutral to  */
                            /* permit use of multicolored      */
                            /* pattern set.                    */
CALL GSPAT(93);             /* Set pattern to orange, pattern 93*/
                            /* in the GDDM 64-color pattern set.*/
```

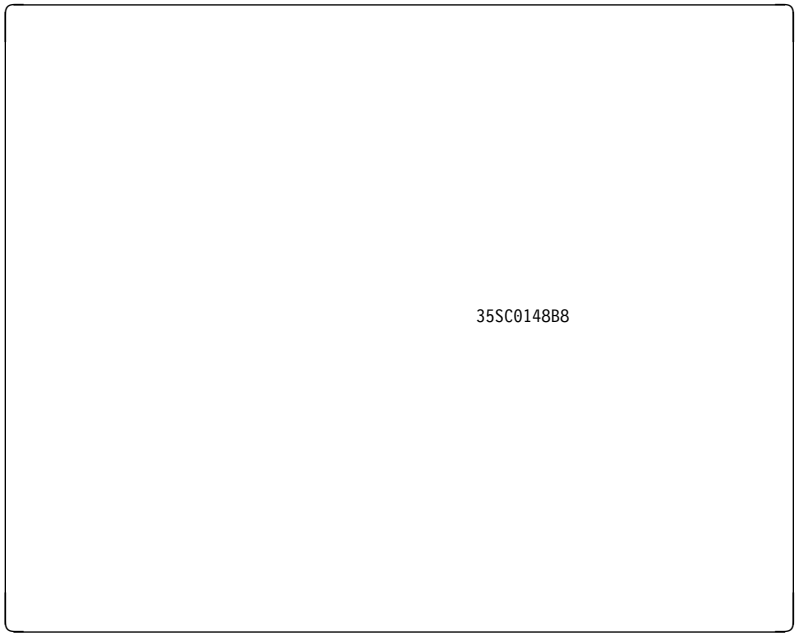


Figure 12. GDDM 64-color pattern set - ADMCOLSD

Pattern 93 in the image symbol set ADMCOLSD is a mixture of red and yellow points. When every cell (and part cell) inside a graphics area is loaded with this pattern, the area appears in orange.

When you use a multicolored shading pattern in this way, the boundary line is white (or black on a printer) unless you reset the color after opening the area.

Note: If you intend using a colored shading pattern to shade an area, you must set the current color to neutral before opening the area. Otherwise, the current color may conflict with the color of the pattern.

Setting the foreground color-mixing attribute, using GSMIX

When you begin to draw a graphics primitive, those parts of the graphics field that already contain visible graphics, are known as the **foreground**.

By default, if the most recent graphics primitive drawn coincides with any part of the foreground, it is added to your picture over all other primitives.

This is known as **overpaint mode**, and is specified explicitly for the segment to which the streets belong by the GSMIX call at **K** in the example. The white area representing the streets is drawn after the blue area representing the river. Where the two areas cross, the streets obscure the river under the overpaint color mixing mode.

You can also use GSMIX to specify that subsequently drawn primitives are to be treated in other ways where they coincide with those already drawn. The other modes that can be set for primitives subsequently added to the foreground are **mix mode**, **underpaint mode**, **exclusive-OR mode**, and **transparent mode**

The seven basic colors that GDDM displays are made up of one or more of the three primary colors, blue, red, and green. If you specify mix mode, and then draw a blue line crossed by a green one, the point where they cross is the mixture of

blue and green, that is, turquoise. Using all combinations of the three primary colors, seven colors can be created, as shown in Table 1.

Table 1. The seven displayable colors

Color displayed	No.	Primaries used		
Blue	1	Blue		
Red	2		Red	
Pink	3	Blue	Red	
Green	4			Green
Turquoise	5	Blue		Green
Yellow	6		Red	Green
White	7	Blue	Red	Green

When you mix two colors, the result is the same as combining all their constituent primaries. For example, red mixed with pink (blue and red) gives blue and red, that is, pink. Turquoise (blue and green) mixed with yellow (red and green) gives blue, red, and green, which is white.

A color representation of the possible mixes is given in Figure 13 on page 42.

The third form of color mixing is underpaint mode. Wherever two primitives cross, the displayed color is that of the first-drawn primitive. If you draw a blue line, then a green line crossing it, the crossing point is shown in blue. Not all devices support underpaint mode (see “Device variations with graphics pictures” on page 48).

You can also use GSMIX to set the color mixing attribute to exclusive-OR (XOR) mode. If the most recent primitive drawn when this mode is set coincides with another, the color that results is a mix of their constituent primary colors less any primary color that is common to both. This means that primitives of the same color cancel each other out, if mixed under this mode. A special use of this quality is described in “Erasing graphics from part of the screen” on page 42.

You should always draw primitives in this mode within a graphics segment. However, you should close the segment or reset the color-mixing attribute before issuing any call to update the output on the GDDM page. If you issue an ASREAD, FSFRCE, or GSREAD call while such a segment is current, the resulting picture may be incorrect.

There are restrictions on the use of this mode with certain devices. These are listed in “Device variations with graphics pictures” on page 48.

If you use GSMIX to select transparent mode for color mixing, the primitives you draw afterward are transparent and therefore do not appear. Not all devices support transparent mode (see “Device variations with graphics pictures” on page 48).

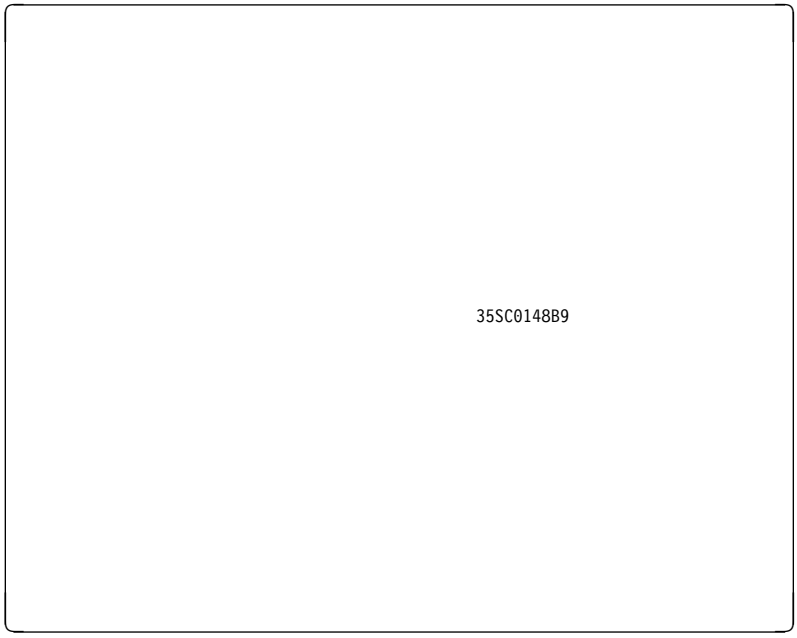


Figure 13. Color-mixing table

The possible parameter values of the GSMIX call and the effects they produce are described in the *GDDM Base Application Programming Reference* book.

As for other graphics attributes, this setting affects only primitives drawn subsequently.

Special treatment of the background color, using call GSMIX

One of the colors you can specify using the GSCOL call is color 8, the **background**. The term background refers to all parts of the graphics field where no pixels have been activated, so this color shows as black on a display and white on a printer or plotter. When it is specified for a graphics primitive, it has a number of special uses.

Erasing graphics from part of the screen

You can do this by using overpaint mode (specified explicitly or by default) and by specifying background as the current color before drawing a graphics area over the primitives you want erased. This technique can be used on a display device, to produce the effect of animation. To show an owl blinking its eye, for example, you would use this sequence of calls:

```
CALL GSSEG(0);   /* Open a graphics segment.               */

Draw owl...

CALL FSFRCE;     /* Send picture of owl with two open eyes. */

CALL BLACK_EYE;  /* Call subroutine to black out one eye.   */

Draw closed eye in blacked-out area...

CALL FSFRCE;     /* Send picture of owl with one eye closed.*/

CALL BLACK_EYE;  /* Call subroutine to black out closed eye.*/

Redraw open eye...

CALL FSFRCE;     /* Send picture of owl with two open eyes. */

BLACK_EYE: PROC;
CALL GSPAT(16);          /* Solid shading pattern.          */
CALL GSCOL(8);           /* Set current color to background.*/
CALL GSMIX(2);           /* Set mixing mode to overpaint.   */

CALL GSMOVE(53.4,70.0);      /* Move to bottom of eye.      */
CALL GSAREA(0);              /* Open area.                  */
CALL GSARC(53.4,70.6,360.0); /* Overpaint eye in background.*/
CALL GSENDA;
END BLACK_EYE;
```

Because “background” means having no primary colors switched on, neither underpaint mode nor mix mode has any effect when a previously drawn background primitive coincides with the most recent one in a visible color.

You cannot draw a new primitive with a visible current color under one already in the graphics field that is colored with background. The background primitive has no color to take precedence.

Remember that the effect of mix mode is to add the primary components of the two colors together. If one of these colors is background, there is nothing to be added – the visible color is unchanged.

Another way of erasing graphics from a picture is to draw it with the color-mixing attribute set to exclusive-OR (XOR) mode. If you draw the same primitive again in the same position with exclusive-OR mode set on, it is deleted. The effect is as if it was never drawn and the underlying primitives or background are shown complete. This provides applications with a way of removing graphics primitives from the picture without causing it to be completely redrawn.

Producing a reverse-video effect

You can achieve the effect of reverse-video by setting the current color to background and writing background graphics text on a colored area. The text may be mode-2 (image) or mode-3 (vector). (Text modes are explained in Chapter 4, “Creating graphics-text output in your application” on page 57.) On most devices, this technique does not work with mode-1 graphics text, because each mode-1 character has a background of its own. If mode-1 graphics text is colored with background, it becomes invisible. The effects of color mix modes on graphics text are described more fully in Chapter 4, “Creating graphics-text output in your application” on page 57. The effects of background-mix modes on graphics text are described in the next section.

Setting the background-mix attribute, using GSBMIX

As you have seen, GDDM gives you control over the mixing of the foreground color of overlapping primitives. For some primitives, you can also control how the **background** of the current primitive combines with any previously drawn primitives with which it overlaps. By default, previously drawn primitives can be seen through the background of the current primitive. This form of background mixing is called **transparent mode**.

The other background mix mode that you can set is **opaque mode**. In this mode, the background of the current primitive completely obscures any previously drawn primitives that it overlaps. The background is black for a display, and white for a printer or plotter.

At **Q** in the example, the GSMIX call sets the background color mixing attribute to opaque mode. This ensures that the underlying primitive, the white area representing the streets, isn’t visible through the background of the shading patterns used for the fountain.

The graphics primitives (and their backgrounds) for which you can set the background-mix attribute are:

Graphics images
The background is every pixel that is not set within an image.

Graphics markers
The background is different if the current marker is an image symbol or a vector symbol.

For image symbols, the background is every pixel that is not set within the marker definition.

For vector symbols, the background is the complete marker box.

Areas
The background is every pixel within the area that is not activated by the shading pattern.

Because the default shading pattern for a graphics area is initially a solid pattern, all points in a shaded area are in the foreground. However, if you select a shading pattern with dots or hatching, the inactive pixels between the dots or lines form the area's background. With the background-mix attribute set to transparent mode, any primitives underlying such a shaded area are visible through the area's background. If the background mix mode is set to opaque, the underlying primitives are covered up by the inactive pixels, which contain background color only.

Graphics text The effect of background mix on graphics text characters is similar to that on graphics marker symbols. It depends on the mode of the text. For mode-1 and mode-2 text, the background of a character is every pixel that is not set within the character definition. The effect of background mix on mode-1 text is also device-dependent. For more information, see “Device variations with graphics pictures” on page 48. For mode-3 text, the background is the complete character box. For more information, see Chapter 4, “Creating graphics-text output in your application” on page 57.

GSBMIX has no effect on lines. The background of a line, even a broken one, is always transparent. Background mix mode is valid for all devices when the foreground mix mode is overpaint. Some devices do not support other combinations of foreground and background mix modes. For details of which devices support which combinations, see “Device variations with graphics pictures” on page 48.

Specifying a transform for graphics primitives, using GSSCT

Using the GSSCT call, you can set a transform that applies to all the primitives that are drawn after that call. Primitives can be transformed in four ways:

Displaced Moved to another x,y location

Scaled Made larger or smaller in the x- or y-direction, or in both

Rotated Moved about a turning point in the x,y plane

Sheared Sloped to one side

Here is a typical call:

```
       /* Scaling  Shearing  Rotation  Displacement Type */
CALL GSSCT( 1,1,     0,1,      1,0,        0,0,       0 );
```

Although the current transform is an attribute of a primitive, the call can only be issued within a currently open segment, and is processed in relation to the origin of the segment (the position x=0,y=0 in world coordinates when the primitive is drawn). GSSCT is described more fully in “Transforming primitives within a segment, using GSSCT” on page 159.

Changing attributes inside an area

If you change attribute values while defining a graphics area, the new attributes affect subsequent parts of the area boundary but not the area fill. Only four attributes may be changed between the GSAREA and GSENDA calls; the line type (CALL GSLT), the line width (CALL GSFLW or CALL GSLW), the color (CALL GSCOL), and the mixing mode (CALL GSMIX).

Note: The color used to shade the area is the color that is current when the GSAREA call is issued. After the GSENDA call is issued, the current color is that specified by the most recent GSCOL call, whether inside the area or not.

Querying the attributes of graphics in a segment

All GDDM calls that set an attribute have a matching call to query the current attribute value. For example: GSQCOL, GSQCA, and GSQFLW query the attributes that can be set by GSCOL, GSCA, and GSFLW.

One use of these calls is to enable a program to call a subroutine using the same graphics environment, again and again, if needs be. For example, a subroutine that draws a thick red circular arc at an x,y position passed to it might look like this:

```
/* Subroutine to draw red circular arc centered on passed x,y position*/

RCIRCLE: PROC(X,Y);
DCL (X,Y) FLOAT DEC(6);    /* Parameters passed to subroutine.  */
DCL COL FIXED BIN(31),     /* Temporary variables.              */
    LW FLOAT DEC(6);

/**********************/
/* Query attributes   */
/**********************/
CALL GSQCOL(COL); /* Save current value of color attribute.     */
CALL GSQFLW(LW);  /* Save current value of line width attribute.*/

CALL GSCOL(2);             /* Change current color to red.      */
CALL GSFLW(2.0);           /* Change current line width to thick*/
CALL GSMOVE(X-2.0,Y-2.0);  /* Move to start of red circle.      */
CALL GSARC(X,Y,360);       /* Draw circular arc about point x,y */

/**********************/
/* Restore attributes */
/**********************/
CALL GSCOL(COL);           /* Restore the color attribute.      */
CALL GSFLW(LW);            /* Restore the line-width attribute. */

END RCIRCLE;
```

So, this subroutine might be called from several different points in the main program. On each occasion, the attributes in the main program would be left unchanged.

Storing and restoring graphics-attribute values, using GSAM and GSPOP

Whenever you alter a primitive attribute to a new value, the old setting of the attribute is automatically saved (PUSHED) by GDDM onto a last-in/first-out stack, unless you specify otherwise. If you wish, your program can subsequently retrieve (POP) the stored attribute value from the stack and reuse the value. The following call controls the pushing:

```
CALL GSAM(0);  /* Preserve attributes */
```

The alternative parameter setting for this call is the value 1, which discards all attribute settings that aren't current. You can save all the graphics attributes introduced in this section (for example, color, line type, current transform) and many others covered elsewhere in this guide. For the full list of the attributes that

can be saved, see the description of GSAM in the *GDDM Base Application Programming Reference* book.

The following call controls the popping:

```
CALL GSPOP(5); /* Restore the last five attributes saved.*/
```

The single parameter defines the number of attribute values to be restored, starting with the last value saved.

For an example of the use of pushing and popping of attribute values, see “Graphics attribute handling with called segments” on page 169.

Changing default attribute values

Application programs can specify attributes for the graphic, alphanumeric, and image elements they process by assigning values to the parameters of API calls. In some cases, if these parameters are omitted from API calls, or if the API call for specifying the attribute is omitted altogether, default values are used for the attribute. These default values are known as **standard defaults**. Not every attribute parameter has a standard default allocated to it.

Note: Do not confuse attribute defaults with GDDM external defaults. External defaults apply to the subsystem environment in which GDDM runs and can be defined in the external defaults module, the user's external defaults file, and within application programs using the SPINIT, ESSUDS, or ESEUDS call.

Overriding the standard default of a graphics attribute

If the standard default of a graphics attribute does not suit the needs of your program and you don’t want to have to specify another value explicitly in each segment in your program, you can choose to override the standard default.

At any point in your program, you can define for each graphics attribute a **drawing default**, which overrides the standard value supplied by GDDM.

You can do this by including calls that reset the attributes between two calls, GSDEFS and GSDEFE, which respectively start and end a definition of drawing defaults.

For example, to change the default value of the current marker symbol from a cross (the GDDM-supplied default) to a square, you would use these calls:

```
CALL GSDEFS(1,1);    /* Start new drawing defaults definition.  */

CALL GSMS(4);        /* Set current marker symbol to square.    */

CALL GSDEFE;         /* End new drawing defaults definition.    */
```

For the above example, any past or subsequent occurrence in your program of GSMARK or GSMRKS for which the marker symbol has not been set (or is set to 0), results in a square marker symbol.

The square remains as the default marker symbol until the end of the program or until the drawing default is changed by another pair of GSDEFS and GSDEFE calls. The first parameter of GSDEFS is always 1. The second parameter may take these values:

1. **Merge** (the default). When merge is specified, the defaults within the new default definition are merged with those in the existing default definition. So the only existing defaults that are affected by the new definition are those specifically set within it.
2. **Override**. When override is specified, the new default definition completely overrides any existing default definition. As with merge, any attribute default specifically set within the new definition changes the existing default attribute that it relates to. Unlike merge, any default that is not specifically set within the new definition is reset to the GDDM default value.

For both merge and override, the existing defaults can be either GDDM standard defaults, or defaults set by a previous default definition.

In general, whenever you change a drawing default, any segment primitive drawn using the old default is redrawn using the new one. For example, you could draw and display a primitive using the default color, green. You could subsequently use several drawing default definitions to change the default color attribute to red, pink, yellow, or any of the colors supported by your display. Each time that you change the default color, the primitive is redrawn in the new default color. Primitives outside segments are discarded when the redraw occurs.

See Chapter 10, “Storing and retrieving graphics pictures” on page 173 for information on how default definitions can affect the storing and restoring of pictures.

For the rules that apply to the use of GSDEFS and GSDEFE, and a complete list of the calls that you can use with them, see the *GDDM Base Application Programming Reference* book.

Device variations with graphics pictures

The preceding sections of this section refer primarily to the 3472-G terminal. However, most of the function is device-independent, so most of the information applies to all graphics devices. The following sections describe functional variations on other types of device.

IBM 3279 terminals

This also applies to other members of the 3270 family that use programmed symbols for graphics.

GSBMIX call

Background mix is only supported when the foreground mix mode is overpaint.

Workstations supported by GDDM-OS/2 Link or GDDM-PCLK

GSCOL call

If the program is to be run on a workstation that is supported by GDDM-OS/2 Link or GDDM-PCLK, you can specify many more values on the GSCOL call than are valid for other devices. Most workstations can display up to 256 different colors, and some can display even more.

GSMIX call

Underpaint mode is not supported for workstations supported by GDDM-OS/2 Link and GDDM-PCLK.

GSPAT call

On workstations supported by GDDM-OS/2 Link and GDDM-PCLK, you can only use the default set of shading patterns. You cannot use patterns from a loaded symbol set for shading graphics.

IBM 3270-PC/G and /GX workstations

GSCOL call

If the workstation is a 3270-PC/GX with a 5371 Model CO1 display unit, 16 colors are supported. Their values are as follows:

-2	White	8	Background
-1	Black	9	Dark blue
0	Default (green)	10	Orange
1	Blue	11	Purple
2	Red	12	Dark green
3	Pink (magenta)	13	Turquoise
4	Green	14	Mustard
5	Turquoise (cyan)	15	Gray
6	Yellow	16	Brown.
7	Neutral		

GSMIX call

Mode 3 (underpaint) is not supported. It is treated as overpaint.

The results of mix mode with combinations of the above colors are described in the *GDDM Base Application Programming Reference* book.

Pattern sets

There must be sufficient symbol set storage available in the workstation for any specified pattern set, otherwise the default pattern is used for shading.

IBM 5080 and 6090 Graphics Systems

GSIMG call

Images created with the GSIMG call require one byte of storage per pixel in both the host computer and 5080 or 6090.

Because of this, you cannot produce multicolored images by overlaying graphics images created by GSIMG. The whole of each successive image blanks out any underlying graphics.

GSAREA call

On the IBM 5080 and 6090 graphics systems, the control parameter on the GSAREA call is ignored and boundary lines are always drawn.

GSBMIX call

This call is not supported.

GSCOL call

16 colors are supported. Their values are the same as the values for the 3270-PC/G and /GX.

GSLW call

Only single line width is supported. Any other specified line width defaults to this.

GSMIX call

Only overpaint mode is supported. A warning message is issued if any other mode is specified.

Pattern sets

Only the 16 GDDM-supplied pattern sets are available, in any of the 16 supported colors. Any other specified pattern set results in pattern 16 (solid).

5550-family Multistations

GSMIX call

Mode 3 (underpaint) is not supported. It is treated as overpaint.

GSBMIX call

Opaque mode is not supported. It is treated as transparent.

Plotters

GSMIX call

Mix mode is not supported.

GSCOL call

The parameter to this call is the number of a pen holder on the plotter, rather than a color. The color that results depends on the color of the pen that the plotter operator puts into the holder. More information is given in “Colors” on page 446.

Pattern sets

You cannot specify user pattern sets for plotters.

GSBMIX call

Background mix is only supported when the foreground mix mode is overpaint.

Printers

GSCOL call

If color separation is required on a family-4 device (see “Family-4 output: Print files for PostScript and PSF- and CDPF-attached printers” on page 404), the value of the GSCOL parameter can range from 0 to the number of entries in the selected color table.

GSMIX call

On some IPDS printers, such as the IBM 4028, 4224, and 4234, only overpaint is supported.

Pattern sets

IPDS printers support only the 16 GDDM-supplied system shading patterns. If your application specifies any other patterns, you need to translate them to supported patterns. On the IBM 4224 printer, unsupported patterns can be translated into supported patterns in different colors. You can do this by using the PATTRAN processing option when you open the device. See “Using DSOPEN to tell GDDM about a device you intend to use” on page 371.

Chapter 3. Including text functions in your programs

GDDM provides four different sets of functions for displaying characters and other symbols:

- Graphics text
- Procedural alphanumerics
- Mapped alphanumerics
- High-performance alphanumerics (HPA)

This section briefly describes each one, to help you decide which to use for a particular purpose, and tells you where to find more information.

Graphics text

This is the simplest set of functions. The caption on the house in Figure 1 on page 6 is in graphics text. It was created simply by executing a GSCHAR call for the line on which it appears.

The primary purpose of graphics text is to annotate graphics displays. It is also used when maximum control over the appearance of text is required, for instance, when preparing presentation material, such as overhead projection transparencies or slides.

The location of the text is specified in world coordinates, and it can be positioned to pixel accuracy. The application program can specify its size, angle, and direction. Characters can be proportionally spaced. Large and complex symbols can be displayed, as well as characters.

On most types of terminal, graphics text is output only. On 3270-PC/G and /GX workstations, and on 5080 and 6090 Graphics Systems, graphics text functions can be used for input, that is, for reading data from the terminal, but they are suitable for obtaining only small amounts of data. The input functions, like the output functions, are intended primarily for use in a graphics context, for instance, to enable the terminal user to enter parameters concerning a picture currently on display.

Graphics text is supported on all devices except alphanumerics-only terminals and printers.

For more information about graphics-text output, see Chapter 4, “Creating graphics-text output in your application” on page 57. For information on receiving graphics-text input, see Chapter 11, “Writing interactive graphics applications” on page 197.

© Copyright IBM Corp. 1982, 1996

Alphanumeric text

The primary purpose of alphanumeric text in GDDM applications is for interacting with the end user. Alphanumeric functions provide a good way of prompting end-users to take actions and of processing their responses.

There are three different ways of including alphanumeric functions in your programs: procedural alphanumerics, mapped alphanumerics, and high-performance alphanumerics. To help you decide which method to use, see "Comparison of the three methods of implementing alphanumeric functions" on page 56.

Procedural alphanumerics

The GDDM alphanumeric calls display one symbol per hardware cell, and exploit the 3270 family's alphanumeric field functions. Comprehensive support is provided for both output and input on 3270 devices. Alphanumeric functions are not supported on some devices, such as plotters.

The procedural functions are so named because the alphanumeric fields are defined procedurally, that is, during execution of the program. There are calls first to define the characteristics of the fields (such as size and position), and other calls to put data into them. After an ASREAD call, alphanumeric data entered by the terminal operator can be read from the fields.

In general, it is not advisable to mix alphanumeric fields with graphics. Their positions are defined in terms of rows and columns rather than by the window coordinates used for graphics. They can be positioned only to cell accuracy, and their appearance cannot be controlled to the same extent as graphics text.

Alphanumerics and graphics can be used together, but to be successful, they usually need to occupy separate areas of the display.

The procedural alphanumeric calls are described in Chapter 5, "Basic procedural alphanumerics" on page 71 and Chapter 13, "Advanced procedural alphanumerics" on page 257.

Mapped alphanumerics

Mapped alphanumerics, like procedural alphanumerics, exploit hardware cells and fields in the terminal. They are supported on a similar range of devices. Mapped alphanumerics differ from procedural alphanumerics in that the layout of a display is defined independently of the application program.

The definition is done interactively, using the GDDM Interactive Map Definition (GDDM-IMD) licensed program. This generates a record of each layout, called a map, to be stored on disk and used by GDDM when the application program is executed.

Compared with procedural alphanumerics, mapped alphanumerics are generally somewhat slower to implement because they require the initial map-definition step. But for displaying more than a small number of fields, particularly if their layout is crucial, mapping has considerable advantages:

- You can define the positions and sizes of all the fields in a display by positioning the cursor on the screen. This is generally much easier than specifying row and column numbers, and it is the major advantage of mapping.
- Execution time performance is likely to be better with mapping than with procedural alphanumerics.

  Mapped alphanumerics generally have significant performance advantages over procedural calls, particularly when 20 or more fields are displayed on the screen. Savings of 60 to 80% of the processor cost are likely if mapped displays are used in preference to procedural alphanumeric calls.
- GDDM automatically maintains recently used mapgroups in storage in case the application needs to use them again, so retrieving the map from DASD can be a one-time cost for the application. However, this increases the end-user's dynamic storage requirement.
- You can change the layout of mapped fields more easily than procedural ones. In many cases, you do not need to recompile the program.

Graphics can be added to mapped alphanumerics in a special graphics area, the size and position of which is specified during map definition.

After sending the mapped output to the terminal, either using ASREAD or the special MSREAD call, an application program can read any alphanumeric input data entered by the operator.

More information is given in Chapter 15, "Mapped alphanumerics" on page 283 and Chapter 16, "Variations on a map" on page 307.

High-performance alphanumerics (HPA)

High-performance alphanumerics (HPA) is another way of handling alphanumerics using GDDM, and is intended for complex applications that require instruction paths of minimum length within GDDM.

HPA provides the dynamic field-definition capabilities of procedural alphanumerics combined with a "buffer" style interface and an even shorter instruction-path length than that required for mapped alphanumerics.

The HPA calls build a data structure that describes all the data, and pass it to GDDM for output. The data entered by the device operator is returned to the HPA application in the same data structure. Changes to the data are shown by indicators, which are part of the structure.

HPA provides most of the advantages of mapped alphanumerics and is particularly suited for use with partitions and dynamic screen layouts.

Note: You should not mix mapped or procedural alphanumeric field definitions with HPA field definitions on the same GDDM page.

You can find more information about using HPA in Chapter 14, "GDDM high-performance alphanumerics" on page 273.

Comparison of the three methods of implementing alphanumeric functions

Procedural alphanumerics, HPA, and mapping provide an alphanumeric input/output service. When should you use each one?

Procedural alphanumerics are likely to be best suited to your needs only if your application is a simple one with a small number of fields. In such cases, the overhead of a separate map definition operation may not be justified. If you need to alter the layout of the display during execution, you should use procedural or high-performance alphanumerics. Otherwise, it is worthwhile creating a map for the following reasons:

- It is much easier to define a display format with GDDM-IMD than with procedural alphanumeric calls. Procedural alphanumeric calls require field locations to be defined in terms of rows and columns, whereas GDDM-IMD enables you to physically indicate locations on a screen.
- Mapping uses the system's resources more efficiently. Some of the processing required to create output data streams and interpret input data streams can be done when the map is generated. It is therefore done only once, instead of every time the program is executed.
- Your application is easier to change. If you need to alter the display format, to take advantage of a new device for instance, you can in many cases use GDDM-IMD to just alter the map. You would not need to alter or even recompile (or reassemble) your program.

Using mapped alphanumerics presupposes a certain knowledge of GDDM-IMD. If you are not very familiar with GDDM-IMD, you might also consider using high-performance alphanumerics for these other reasons:

- High-performance alphanumerics does not require DASD I/O to retrieve a mapgroup as mapped alphanumerics does.
- The dynamic nature of HPA field definition means that it is better suited than mapped alphanumerics for use in applications where the layout of the display is altered during execution.
- The length of the instruction path of HPA is shorter than that for mapped alphanumerics.
- With mapped alphanumerics the layout of the screen can be changed independently of the application. The same application can use different maps for different sized screens and different national languages.

Chapter 4. Creating graphics-text output in your application

This section describes how to write programs that produce graphics-text output on devices that use vector graphics, such as the IBM 3472-G. Input on 3270-PC/G and /GX workstations, and on 5080 Graphics Systems, is described in “String input” on page 204. See 69 for device variations.

Subjects described in this section are:

- Drawing graphics text
- Affecting the appearance of graphics text
- A programming example is provided to demonstrate:
  - Selecting a mode for the graphics text
  - Ensuring graphics text is readable
  - Breaking lines of graphics text
  - Changing the size and proportions of text characters
  - Changing the space between characters of graphics text
  - Concatenating strings of graphics text
  - Changing the slope of a graphics-text string
  - Changing the direction of a graphics-text string
  - Making graphics-text characters appear italic
  - Outlining strings of graphics text
  - Aligning graphics text
- Graphics-text variations on other devices

Drawing graphics text

There are two GDDM API calls that enable you to do this; GSCHAR and GSCHAP.

Drawing a line of graphics text at a specified position, using GSCHAR

In “Example: The HOUSE program” on page 6, graphics text was added to the HOUSE example program by means of this call.

```
/*          X-COORD  Y-COORD  LENGTH          STRING                */
CALL GSCHAR( 33.0,     2.0,     28,  'All dimensions are in meters');
```

GSCHAR writes a string of graphics text of a specified length at a specified position.

As with all graphics calls, the position is given in world coordinates rather than the rows and columns scheme used for alphanumerics.

Drawing a line of graphics text at the current position, using GSCHAP

The GSCHAP call is similar to GSCHAR, but it draws a graphics text string at the current position rather than at a specified one.

```
/*          length          string                                */
CALL GSCHAP(  28,  'All dimensions are in meters');
```

This makes it easy to concatenate graphics text strings.

© Copyright IBM Corp. 1982, 1996

Affecting the appearance of graphics text

There are eight different attributes that affect the appearance of graphics text:

character mode	character box
character angle	character direction
character shear	character box space
text alignment	character symbol set

Whenever some graphics text is written (with a GSCHAR or GSCHAP call), the current values of these eight attributes apply, whether they have been set explicitly or by default.

The most important attribute of graphics text is the mode. How much effect the other attributes have on a string of graphics-text characters depends on the mode of the text. In general, all the attributes apply fully to mode-3 graphics text. Some of them apply to mode-2 graphics text, but hardly any affect mode-1 graphics text.

The different settings of the character-mode attribute are described in “Choosing a suitable mode of graphics text” and the relative advantages of each mode are described in “Advantages and disadvantages of each character mode” on page 61. “Example: Subroutine to label the streets of the TOWN program” on page 62 shows several uses of the other graphics-text attributes.

It is the attribute values current at the time of the GSCHAR call that affect the appearance of the characters. The attribute values at the time of the ASREAD call have no particular significance. An exception to this is if GSCHAR uses the default value of any attribute (such as character mode). If such a default is subsequently changed (from mode-3 to mode-2, for example), the appearance at ASREAD **is** affected.

Choosing a suitable mode of graphics text

Before issuing any GSCHAR or GSCHAP calls to create graphics text, you can use the GSCM call to specify a value of 1, 2, or 3 for the **mode** of the text.

```
CALL GSCM(3);          /* Set character mode to 3 - vector text.*/
```

The mode applies to all subsequent GSCHAP and GSCHAR calls until the character mode is changed again. If the program uses segments, opening a new segment resets the mode to the default. If the character-mode attribute is not specified, the default is mode-1.

The character mode determines which type of **symbol set** is used to draw the characters. A symbol set is a collection of characters and other symbols; usually they are all a particular style, or font, such as Times Roman or Gothic.

For a fuller description of symbol sets, see Chapter 12, “Using symbol sets” on page 233. Briefly, there are two sorts:

Image symbols These are defined in terms of pixels. They can be either built into the terminal, in which case they are called **hardware symbols**, or loaded into it from the host computer.

Vector symbols These are defined in terms of straight and curved lines. They are loaded into the terminal from the host, except in the cases described in “On the IBM 3270-PC/G and /GX workstations” on page 69.

GDDM supplies a number of image and vector symbol sets. In addition, users can create their own.

Graphics text Modes-1 and -2 are highly device-dependent. This section describes their use primarily on terminals that use the 3270 data stream such as the IBM 3472-G. Differences on other types of device are described in “Device variations with graphics text” on page 69.

The relative advantages and disadvantages of the three modes on all types of terminal are discussed later, in “Advantages and disadvantages of each character mode” on page 61.

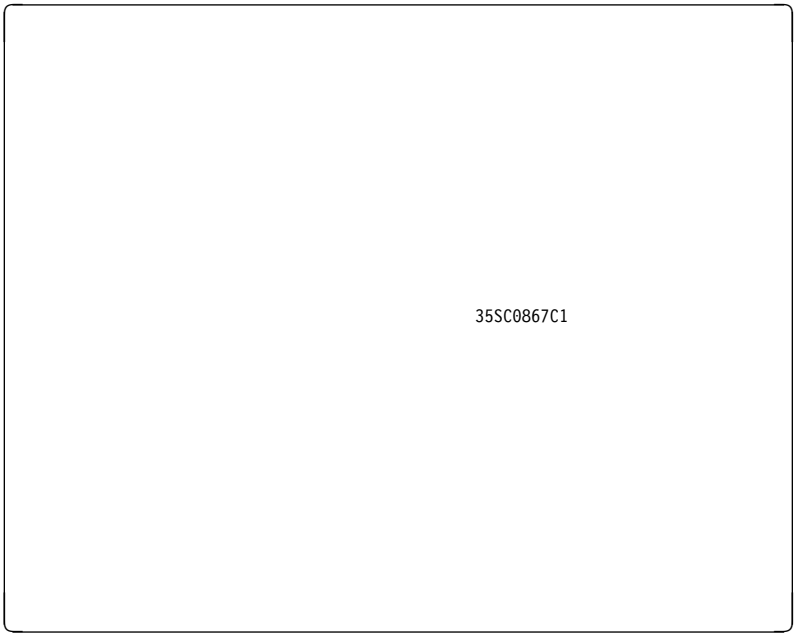


Figure 14. Mode-1 and mode-2 graphics text

Mode-1: String-positioned graphics text

Mode-1 is known as string-positioning mode because the application program can control the position of the start of the string only.

If you specify mode-1 before adding some graphics text to your program, an image symbol set is used for the characters. By default, the device’s own hardware symbol set is used, but the application program can load an alternative image symbol set (see “Loading symbol sets for graphics text” on page 235). Only image symbols that match the hardware cell size of the device can be used. Because the PS stores of most devices are monochrome, monochrome symbol sets are most suitable for mode-1 graphics text.

Like GDDM alphanumeric output (see Chapter 13, “Advanced procedural alphanumerics” on page 257), the character symbols occupy one hardware cell each. That is why only the beginning of a mode-1 graphics text string can be specified. Where a string of text ends and the amount of space placed between the characters depend on the cell size of the device.

Mode-1 symbols occupy their cells completely. On displays that use programmed-symbol stores for graphics, any graphics in the cell is obliterated. This can aid the readability of the text when it overwrites graphics.

Most display devices, however, use vector graphics and so mode-1 graphics text, like that of other modes, shares the hardware cell with underlying graphics.

Mode-2: Character-positioned graphics text

Character-positioned graphics text is so called because the position of each character (or symbol) within a string is dictated by the application rather than by the hardware.

Mode-2 text is similar to mode-1 in many respects. Like mode-1, it is composed of image symbols. GDDM loads a default image symbol set, or the application may load one explicitly (see “Loading symbol sets for graphics text” on page 235). The symbols may be of any size, and they are positioned to pixel accuracy.

Figure 14 on page 59 shows how a mode-1 character occupies a whole hardware cell, but a mode-2 character may occupy several cells. If a symbol set does match the hardware cell size, it may be used for either mode-1 or mode-2.

The pixels that make up a mode-2 text string are merged with those representing the requested graphics. They do not take precedence over the graphics. They are on an equal footing, and are subject to the same color-mixing rules (see “Setting the foreground color-mixing attribute, using GSMIX” on page 40 and “Setting the background-mix attribute, using GSBMIX” on page 44).

Mode-3: Stroke-positioned graphics text

Mode-3 is called stroke-positioned mode, because the application program can control the drawing of every stroke of every symbol.

When you specify mode-3 before adding some graphics text to your program, a vector symbol set is used for the characters. GDDM loads a default vector symbol set, or the application can load one explicitly (see “Loading symbol sets for graphics text” on page 235).

Because each symbol is created as a sequence of lines and arcs, GDDM can manipulate it into any required size, aspect ratio, angle, or shear (italicization). Each symbol is positioned in the display to the maximum accuracy allowed by the hardware (pixel accuracy on 3270 terminals).

The lines and arcs that make up a mode-3 text string are merged with those representing the requested graphics. Like mode-2 text, they do not take precedence over the graphics, and they are subject to the same color-mixing rules as graphics primitives (see “Setting the foreground color-mixing attribute, using GSMIX” on page 40 and “Setting the background-mix attribute, using GSBMIX” on page 44).

Advantages and disadvantages of each character mode

Each character mode has certain features that give it advantages over other modes in particular situations. These are the main features of each mode:

Mode-1: String-positioned graphics text

Advantages: Of the three modes of graphics text, this one requires GDDM to perform the least processing. Multicolored symbols can be used, except on the IBM 5080 and 6090 Graphics Systems. On display devices that use programmed symbols for graphics, the fact that mode-1 text is the sole occupant of the hardware cells aids its readability where text and graphics coincide. Other modes merge the text with the graphics.

Disadvantages: These are best considered individually for each type of supported device:

- IBM 3270 devices (except the 3270-PC/G and /GX): The text can be positioned only to hardware cell accuracy. Its placement relative to the graphics varies, therefore, from device to device. The size of each character in a symbol set has to match the cell size of the device. This prevents the use of large symbols and requires a separate version of the symbol set for each device of different cell size.
- Plotters, IBM 3270-PC/G and /GX, 5550: Although the text can be positioned to the maximum accuracy allowed by the hardware, the size, direction, and angle of the characters are fixed.
- Advanced-function printers: If vector symbols are used they are limited to one size–that of the default character box. The limitation can be overcome by using image symbols, which can be of any size.

Mode-2: Character-positioned graphics text

Advantages: The limitations on character size and positioning mentioned for mode-1 can be avoided. You can use image symbols. Multicolored symbols are permitted, except on the IBM 5080 and 6090 Graphics Systems. With image symbols, the dot representation of each character is always exactly the one that was defined when the symbol set was created. The characters do not therefore suffer from distortion, as vector characters may in some circumstances.

Disadvantages: The characters cannot be rotated or otherwise manipulated. You can use image symbols to achieve a particular size of character, but the size is fixed when the symbol set is created; the characters may not be expanded or contracted by the application program.

Mode-3: Stroke-positioned graphics text

Advantages: Because each character is originally created as a sequence of lines and curves, GDDM can manipulate the symbols when they are displayed. They may be shown at any size or aspect ratio (GSCB), rotated (GSCA), or sheared (GSCH).

Disadvantages: The symbols are monochrome. On 3270 devices, rastering is subject to rounding errors. The end of each line in the symbol can be resolved only to the nearest pixel (screen position). This means that mode-3 characters displayed at a small size may be difficult to read. Mode-2 may therefore be preferable when small characters are required on 3270 devices.

On the 3270-PC/G and /GX family, and the 5550 family, mode-3 text takes longer to draw than mode-1 and -2.

Example: Subroutine to label the streets of the TOWN program

If this subroutine is added to the TOWN program before the ASREAD call, it uses graphics text to attach names to the streets drawn by the main program.

```
/* ANNOTATE THE STREET MAP */
LABELS: PROC;
DCL(X1, X2, Y1) FLOAT DEC(6);
DCL XC(5) FLOAT DEC(6);
DCL YC(5) FLOAT DEC(6);

DCL NEWLINE CHAR(1);
NEWLINE ='15'X;

CALL GSSEG(7);                                                    A
CALL GSLW(2);
CALL GSMIX(2);                      /* Overpaint mode        */
CALL GSCOL(2);
CALL GSCM(3);                       /* Mode-3 graphics text  */   B
CALL GSCHAR(2.0, 91.0, 9, 'Mill St.');                            C
CALL GSCHAR(23.0, 91.0, 16, 'Old Mill Bridge');
CALL GSCA(2.0, -5.6);               /* Characters tilted at  */
                                    /* slope -5.6/2          */
CALL GSCHAR(16.0, 88.0, 10., 'New Bridge');
CALL GSCA(0.0, 0.0);                /* Reset character angle */
CALL GSCB(1.8, 6.0);                /* Use larger characters */
CALL GSCHAR(3.0, 58.0, 13, 'Cyprus Avenue');
CALL GSCHAR(75.0, 58.0, 11, 'Fountain St.');
CALL GSCA(2.0, -3.3);               /* Change character angle*/   D
CALL GSCHAR(15.0, 32.0, 3, 'Lin');
CALL GSCA(2.0, -3.0);               /* Change character angle*/
                                    /* to fit curved street  */
CALL GSCHAP(5, 'den');              /* Complete street name  */   E
CALL GSCA(2.0, -1.2);               /* Less steep angle      */
CALL GSQCP(x1, y1);                 /* Query current position*/   F
x1 = x1 + 1;
y1 = y1 + 0.5;
CALL GSCHAR(x1, y1, 4, 'Road');
CALL GSCA(0.0, 0.0);                /* Reset character angle */
CALL GSCD(2);                       /* Characters downward   */   G
CALL GSCB(1.8, 4.3);
CALL GSCHAR(60.0, 40.0, 8, 'Broad St');
CALL GSCHAR(60.0, 99.0, 7, 'Lee Way');
```

Figure 15 (Part 1 of 2). Subroutine to name streets on the town plan

```
CALL GSCD(0);
CALL GSCA(0.0, 0.0);
CALL GSCB(2.8, 8.0);                                                    H
CALL GSCBS(0.2, 0.18);                                                  I
CALL GSCOL(6);
/* Now write a string of characters and outline their text box  */
CALL GSCHAR(80.8, 91.0, 11,'TOWN'||NEWLINE||'CENTER');                  J
CALL GSQTB       (11,'TOWN'||NEWLINE||'CENTER',5,XC,YC);                K

X=80.8;
Y=91.0;

CALL GSLW(1);
CALL GSCOL(4);
CALL GSMOVE(X+XC(1),Y+YC(1));      /* Bottom left of text box */        L
CALL GSLINE(X+XC(3),Y+YC(3));        /* Draw around ...       */        L
CALL GSLINE(X+XC(4),Y+YC(4));        /*           the ...     */        L
CALL GSLINE(X+XC(2),Y+YC(2));        /*              text box */        L
CALL GSLINE(X+XC(1),Y+YC(1));        /*                       */        L
CALL GSCB(1.8, 6.0);                 /* reduce character box  */
CALL GSCBS(0.0, 0.0);                /* Default char. spacing */
CALL GSCOL(7);
CALL GSCH(1.0, 3.0,);                /* Italic characters     */        M
CALL GSCHAR(4.0, 78.0, 4, 'River');
CALL GSCA(1 1);
CALL GSCHAR(27.0, 81.0, 3, 'Usk');
CALL GSSCLS;
END LABELS;
```

Figure 15 (Part 2 of 2). Subroutine to name streets on the town plan

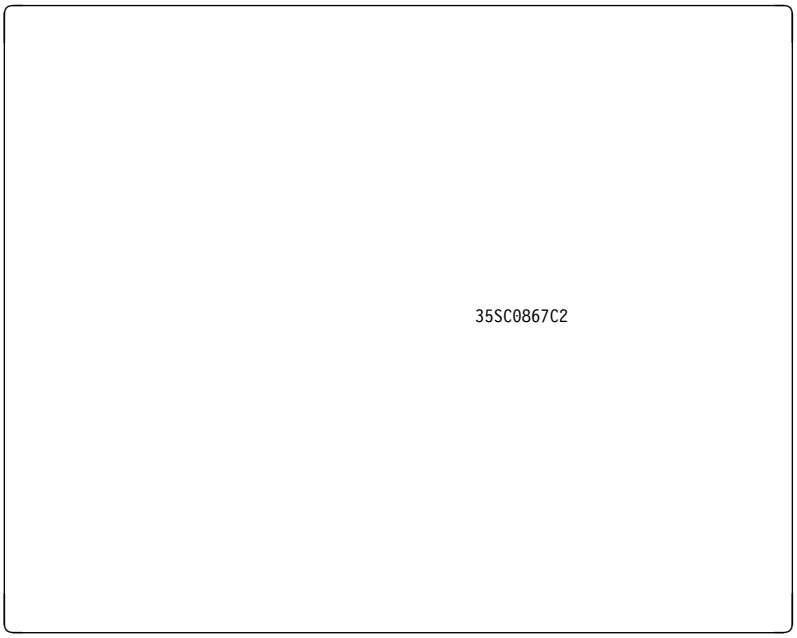


Figure 16. Output from the subroutine to annotate the town plan

Tasks illustrated by the LABELS subroutine

The subroutine demonstrates several ways in which you can affect the appearance of graphics text in an application program. These techniques are discussed in the sections that follow.

Selecting the mode of graphics text to be used

On any map, it is important that the names of the objects shown be placed as close as possible to those objects. At **B** in the example, the GSCM call is used to specify that the graphics text to name the streets be drawn using mode-3 characters. Graphics text of this mode is the most manageable and suits best the task of labeling streets that go in different directions and have different widths and lengths.

Ensuring that graphics text is readable

Graphics text is subject to the same mixing rules as other graphics primitives. If a segment that contains graphics text also contains other graphics primitives, the text mixes with the graphics where they coincide (according to the current mix-mode for that segment). Such mixing can diminish the readability of the text.

Mode-1 graphics text does not have this readability problem on devices that use programmed symbols for graphics, but this routine needs to use mode-3 characters to label the map properly. At **A**, the routine prevents the graphics text from mixing with any graphics by enclosing the text in a segment of its own.

Whether or not you keep graphics text in a segment of its own, you can make the text as readable as mode-1 text is on devices with PS stores, by setting the background mix mode to opaque.

Breaking lines of graphics text

The new-line character code, `X'15'`, contained in the string of graphics text drawn at **J**, requests that the word ‘CENTER’ begin on a new line. Instead of performing two separate GSCHAR calls to place the words ‘TOWN CENTER’ at the right-hand side, one call with this special character code is enough. The new line starts directly under the starting point of the previous one.

Note: The new-line character code is passed to the GSCHAR as the character variable, NEWLINE. GSCHAR and GSCHAP can take character constants, character variables, or mixtures of these in the string parameter.

Changing the size and proportions of text characters

Using mode-3 graphics text and manipulating the character-box attribute, you can add text characters to your program which are different, in size and shape, from those you normally get from the hardware of your device.

Each mode-2 and mode-3 character is positioned within an invisible character box. A mode-3 character expands or contracts to fill the character box when it changes size in one or both directions.

At **H** in the subroutine, the GSCB call specifies the width and height of the character box. This affects the size and spacing of the characters within the text string that follows.

The graphics text strings that are used to name streets prior to this use the default character box, the hardware character cell. At **H** a larger character box is specified because the string drawn at **J** is to be the title of the map.

Each mode-3 character is scaled to fill the current character box of 2.8 by 8.0 in world coordinates. The size of this character box can be changed to fit graphics-text strings of different lengths in streets of different length and width.

Note: If you use mode-2 graphics text and load an image symbol set, in which the characters are larger than the character box, you may find that characters overlap each other. To avoid this, you need to either enlarge the character box or introduce character-box spacing multipliers (see “Changing the space between characters of graphics text”). You can query the size of the character box using the GSQCB call.

Changing the space between characters of graphics text

The standard space between adjacent characters is the sum of the spaces between each character and the edge of its character box. With the GSCBS call, you can set the character-box spacing attribute to insert extra horizontal and vertical space between the character boxes of graphics text.

The two multipliers you supply with the GSCBS call are applied to the width and height of the character box to determine how much horizontal and vertical space to insert between character boxes in the string. (They do not affect the size of the character box itself.) Initially the value of both multipliers is 0, so standard spacing is used and the character boxes are adjacent.

At **I**, the GSCBS call specifies a positive nonzero value for character-box spacing so that subsequent GSCHAR and GSCHAP calls draw characters with more space between them. (A negative value would cause character boxes to overlap.)

A multiplier of 0.2 is applied to the width of the current character box. This specifies that 0.56 x-units of horizontal space (2.8 x 0.2) are to be placed between the character boxes in subsequent strings.

A multiplier of 0.18 is applied to the height of the current character box to specify the vertical space to be placed between the lines of subsequent strings of graphics text. When the new-line character occurs at **J**, this multiplier causes the new line to be placed 1.44 y-units (8.0 x 0.18) below the first.

Note: For graphics text characters spaced proportionally to their individual widths, see “Using proportionally-spaced characters” on page 68.

Inserting space between the character boxes of proportional characters corrupts their proportional spacing.

Concatenating graphics text

At **E** in the example, the GSCHAP call places a graphics-text string, ‘den’, at the current position. The current position prior to the GSCHAP call was at the end of the last text string drawn by the GSCHAR call. In this way, the last three letters of the word ‘Linden’ can be concatenated onto the first three. Because the GSCHAP call positions the string at the current position, there is no need to specify the point at which to position the beginning of the second string.

Changing the slope of a graphics-text string

Because the ‘New Bridge’ and ‘Linden Road’ lie at unusual angles on the street map, the graphics text that labels them must also follow these angles.

By changing the character-angle attribute before drawing any text, the routine can ensure that street names are drawn along each street. The GSCA call at **D** specifies values for dx and dy, which together determine the slope of a base line along which subsequent character boxes will be drawn. Here dx has a value of 2 and dy a value of –3.3. This causes the letters ‘Lin’ to be drawn sloping downward to the right.

Because the street is curved, the angle of slope of the letters must change to keep the label on the street. As the slope of the street becomes less steep, the character angle attribute is made less steep for the character strings ‘den’ and ‘Road’.

Changing the direction of a graphics-text string

Two streets on the map run in a vertical direction. However, the default direction for writing graphics text is from left to right. Before labeling these streets, the character-direction attribute is changed. At **G**, the parameter of the GSCD call specifies that each successive character box is to adjoin the bottom of the preceding one instead of the right-hand side. This causes the names of the streets to be written in a downward direction with each letter in an upright position.

This is the standard direction for Chinese and Japanese text. You can also use it to label the vertical axis of a chart.

Another way of labeling these streets would be to specify for the character-angle attribute a value of 0 for dx and a value of –1 for dy. With mode-3 graphics text, this would cause the names to be written downward along each street but the letters would all be rotated by 90°. With mode-2 characters, however, letters are not rotated, so the effect is similar to that of the character-direction attribute.

Making graphics-text characters appear italic

You can use the character-shearing attribute to produce an italicizing effect on mode-3 graphics text characters. The parameters of the GSCH call at **M** specify values for dx and dy that determine the slope of characters drawn by the subsequent GSCHAR and call that labels the river. The characters drawn to label the river, slope in a clockwise direction because the values for dx and dy are both positive. The slope would also be in this direction if both values were negative but, if one value were positive and the other negative, characters drawn afterward would slope in an counterclockwise direction.

Outlining the text box around a graphics-text string

The character boxes in which a graphics-text string is drawn, are conceptually enclosed within a rectangle or parallelogram called a **text box**.

If you allow the character-box space and the character direction to default, the character boxes are contiguous. This means that the width of the text box is the sum of the widths of the character boxes, and its height is the same as the current character-box height.

However, in the graphics text string drawn at **J**, a nondefault character-box spacing is used and the string is broken by a new-line character. In this case, the depth of the text box is equal to the character-box height multiplied by the number of lines plus 1.44 multiplied by the number of new-line characters. The text box is as wide as the longest line in the string. The width of the text box is not simply a multiple of the character-box width; it also includes five horizontal spaces between character boxes. (Neither are the dimensions of the text box obvious, if you use proportionally spaced vector symbols.)

At **K**, the GSQTB call is used to find out the positions of the corners of the box, and the current position after the characters have been drawn.

In the pair of arrays named as parameters of GSQTB, GDDM returns the coordinates of the corners of the text box.

A precise definition of the order in which the points are returned is given in the *GDDM Base Application Programming Reference* book.

The fifth element of each array gives the offsets of the current position after the character string has been generated. This pair of offsets identifies where the next character would be drawn.

These offsets are returned as if the starting point of the string is at 0,0. For this reason, at **L**, the coordinates of the actual starting position of the string are added to the offsets returned to the arrays, to join the points that accurately outline the text box.

Aligning text within the text box

With the GSTA call, you can use the text-alignment attribute to alter the point on the text box, toward which the text is aligned. This is a typical call:

```
GSTA(3,2); /* Align center top of text box with current position*/
```

Throughout the LABELS subroutine, the text-alignment attribute is allowed to default. The text is therefore aligned such that a point on the text box corresponds with either the point specified in the GSCHAR call, or the current position before a GSCHAP call was issued.

The character direction determines which point on the text box is used as the alignment point. The alignment point is the bottom-left corner of the leftmost character box in the first row of text, if these conditions are met:

- The graphics window is uniform and has its origin at the bottom left-hand corner of the screen.
- The character-angle, -direction, and -shear attributes use the GDDM default values.
- The width and height of the character box are both positive values.

Default alignment points for other character directions are described in the *GDDM Base Application Programming Reference* book.

Using proportionally-spaced characters

The maximum width of a mode-3 symbol is the width of the character box. But symbols can be assigned individual widths less than this when the symbol set is created.

Symbols that do have individual widths are said to be **proportionally spaced**. GDDM supplies a number of proportionally-spaced vector symbol sets that you can load into storage and use for graphics text. (See “Loading symbol sets for graphics text” on page 235 and the *GDDM Base Application Programming Reference* book for details.) Alternatively, you can create your own symbol set using the Vector Symbol Editor and assign a width to each character.

If a symbol set is not proportionally spaced, a narrow character like an “i” is allocated just as much space as a wide one like a “W”. The result is empty space around narrow characters. The advantage of proportionally spaced characters is that GDDM displays them at a spacing that is in proportion to their individual widths. This gives a more pleasing appearance and more compact character strings. The difference is illustrated in Figure 17.

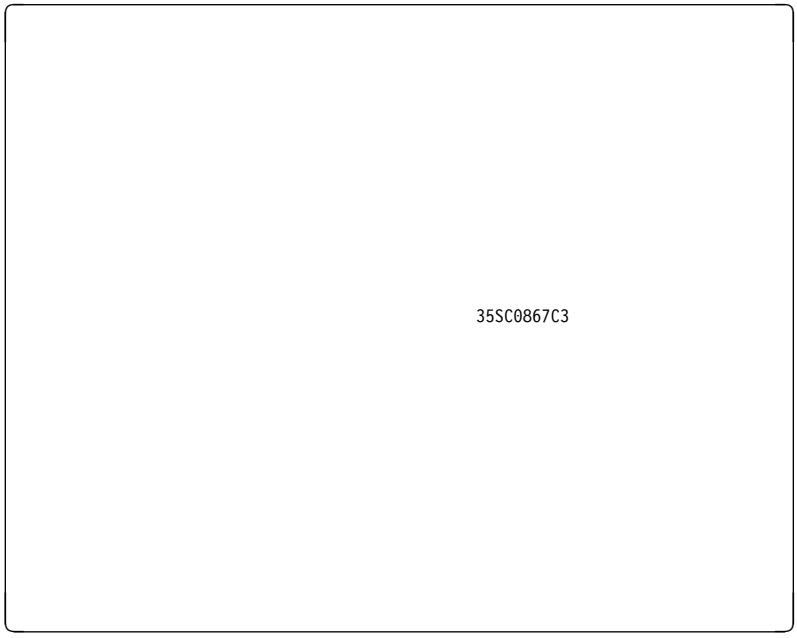


Figure 17. Effects of proportional spacing

The spacing works as follows. After GSCHAR or GSCHAP has drawn a nonproportionally spaced character, the current position is moved along by an amount equal to the width of the character box. After drawing a proportionally spaced character, the movement is a fraction of the character box width. The fraction is equal to the ratio between the character's assigned width and the maximum, as recorded in the definition of the character.

The amount of space occupied by a proportionally spaced character string can be determined by the GSQTB call (see “Outlining the text box around a graphics-text string” on page 66).

For mode-2 and mode-3 characters, you can also control the amount of space between character boxes, using the character box spacing attribute.

Device variations with graphics text

The preceding sections of this section apply primarily to members of the 3270 family that produce vector-graphics output, such as the 3472-G. The following sections describe functional variations on other types of device.

On IBM 3279 color displays

Mode-1 text The characters occupy the hardware cells exclusively. Where mode-1 graphics text coincides with other graphics, the background of each character 'blacks out' the underlying graphics.

On the IBM 3270-PC/G and /GX workstations

Mode-2 and -3 text The workstation has a hardware image symbol set and a hardware vector symbol set. These symbol sets are used as the defaults for modes 2 and 3 unless you specify that a GDDM symbol set is to be loaded and used instead.

Default character box For all modes of text, the default character box is the hardware graphics cell size, which is different from the hardware alphanumerics cell size.

Alphanumerics cells have a predefined size and predefined locations, in rows and columns, on the screen. The sizes of graphics cells are predefined, but their locations are not.

On the IBM 5080 and 6090 Graphics Systems

Mode-1 and -2 text Image symbols occupy the cell completely. Other graphics data in the cell is obscured by the text and its background.

Default character box For all modes of text, the default character box is the character size of the 5080 base-character set.

On IBM 5550-family Multistations

Mode-2 and -3 text The same as for 3270/PC/G and /GX.

Default character box The same as for 3270/PC/G and /GX.

If Japanese 3270-PC/G software before Version 6 is used, the 5550 family has no mode-3 hardware image symbol set. GDDM's default mode-3 symbol set is used if not loaded explicitly. For DBCS text, GDDM's DBCS symbol set is automatically loaded.

On advanced function printers and the IBM 4250

This section describes how text output on advanced function printers, such as the IBM 3820 and 4028, appears different from text output on displays of the IBM 3270 family.

Effect of call GSCB

Mode-1 text The GSCB call has no effect. If image symbols are used, the character box is the same size as the symbols. If vector symbols are used, the character box is the default one, and the width and depth of the symbols are scaled separately to fill the box.

Mode-2 text The symbols come from either an image symbol set specified by you, in which case the effect of the character box is the same as on ordinary 3270 devices, or from the default vector symbol set, in which case they are scaled to fill the box, as for mode-3.

Default character box
The default character box is such that letter heights approximating to 12 points (1/6 inch) are produced. The width is half the height. In terms of pixels, this means, for example, 100 pixels deep by 50 wide on a 4250, and 40 deep by 20 wide on a 3800.

On plotters

Some special considerations for plotters are described in “Symbol sets” on page 450.

Mode-1 text The start of the string is positioned to the maximum accuracy allowed by the hardware.

Mode-2 text The pixel spacing for image symbols is as described in “Cells, pixels, and plotter units” on page 438.

If no image symbol set is loaded by the program, the default vector symbol set is used. The characters are then scaled to fit the current character box as far as possible without distortion.

Default character box This is the notional cell described in “Cells, pixels, and plotter units” on page 438.

Chapter 5. Basic procedural alphanumerics

This section introduces the basic facilities that GDDM provides for programs to send alphanumeric data as output to displays and printers and to receive alphanumeric input through the end user's keyboard. Ways of providing more advanced alphanumeric functions are described in Chapter 13, "Advanced procedural alphanumerics" on page 257, Chapter 15, "Mapped alphanumerics" on page 283 and "High-performance alphanumerics (HPA)" on page 55.

Alphanumeric output cannot be sent to graphics-only devices such as plotters.

On the IBM 3270 family of devices, the display area (that is, the screen or printed page) is divided into *cells*. The cells are rectangular in shape, they are arranged in rows and columns, and each can display one character (or symbol). GDDM enables you to define contiguous blocks of cells to be *alphanumeric fields*.

You can specify where on the display area the fields are to be located. Alphanumeric data may be transmitted to them, and a terminal operator may type input data into them. All the calls that process alphanumeric fields have the format CALL ASxxxx.

The facilities provided by these calls are called *procedural alphanumerics*, to distinguish them from GDDM mapped alphanumerics and high-performance alphanumerics. A comparison of the three methods of adding alphanumeric function to your programs and guidance on which to use in different situations is given in "Comparison of the three methods of implementing alphanumeric functions" on page 56.

Logically, alphanumeric fields are stored, like graphics, in pages by GDDM. When an alphanumeric field is created, it is added to the current page. A page can therefore contain both graphics and alphanumeric fields.

The way in which these fields combine depends on the device. On the 3472-G, 3192-G, 3179-G, 3270-PC/G and /GX family, and 5550 family, you can control the precedence of alphanumerics over graphics. See "Device variations with procedural alphanumerics" on page 82. On a 3279, the alphanumerics take precedence; no graphics appear in hardware cells that are part of an alphanumeric field.

On some terminals (such as the dual-screen configuration of the 3270-PC/GX, the 5080 and 6090 Graphics Systems), the graphics are displayed on one screen and the alphanumerics on another. See "IBM 5080 and 6090 graphics systems" on page 82 for details of procedural alphanumerics on the 5080 and 6090.

Defining an alphanumeric field, using call ASDFLD

This is a typical call to define an alphanumeric field:

```
/*      Field-id    Row    Column   Depth     Width    Type     */
CALL ASDFLD(3,      14,      5,       1,       21,       2);
```

© Copyright IBM Corp. 1982, 1996

This statement creates a new alphanumeric field on the GDDM page and assigns it the number 3 as an identifier. If there is a field with the same identifier on the GDDM page already, it is erased and when the new field is created.

The ASDFLD call also specifies that the new field should begin on the 14th line from the top of the page and in the 5th column. It is one row deep and 21 columns wide. The type parameter specifies how the field should be handled by the terminal. The field is to be a protected alphanumeric field, which makes the keyboard lock, if the operator tries to type into the field.

Note: Whereas the position of GDDM graphics on a page is defined in terms of a device-independent user-chosen coordinate system (or the default coordinates of 100 by 100), alphanumeric fields are positioned in row and column coordinates.

Sending and receiving alphanumeric data, using ASCPUT and ASCGET

To use a field for output, you must assign data to it. A typical statement would be:

```
CALL ASCPUT(3,21,'ENTER ACCOUNT NUMBER:');   /* Put data in field 3 */
```

This call requests GDDM to place 21 characters of data into the alphanumeric field with field identifier 3.

When an unprotected field is sent to the screen (by issuing an ASREAD), the terminal operator may type data into it. This data is sent to the program when the terminal operator presses the ENTER key (or causes any other interrupt). The program can then retrieve the data with a call such as:

```
CALL ASCGET(4,5,ACCOUNT_NO);         /* Retrieve data from field 4 */
```

This call requests GDDM to retrieve the data from field 4 and place the first 5 characters (typically the complete field) into the program variable called ACCOUNT_NO.

Breaking lines of alphanumeric text

Multiline fields can be created in two ways. You can define a field one line deep but long enough to extend beyond the edge of the page. GDDM wraps the field around the screen and continues it on the next line, and on following lines if necessary.

```
CALL ASDFLD(19,4,21,1,150,2); /* Field continues on lines 5 & 6 */
```

Or you can define the field to be narrow enough to fit onto the page, but more than one line deep:

```
CALL ASDFLD(20,4,21,2,7,2);   /* Field is 2 rows by 7 columns   */
```

The data of such a multiline field is considered as one long string:

```
CALL ASCPUT(20,14,'AccountProgram'); /* Put data in 2-row field */
```

Field 20 is to have its top left-hand corner character in row 4, column 21, and is to appear like this:

```
Account
Program
```

Were this field an input field, its contents would be retrieved by a call such as:

```
CALL ASCGET(20,14,INCHAR);
```

where INCHAR is the name of a character variable 14 bytes long.

Clearing an alphanumeric field, using call ASFCLR

To clear the data from a single alphanumeric field, you can issue this call:

```
CALL ASCPUT(6,0,'');    /* Assign null data to field 6  */
```

The previous content of field 6 is replaced with null characters.

When there are several fields to be cleared, you may issue one of these calls:

```
CALL ASFCLR(0);         /* Clear all unprotected fields */
CALL ASFCLR(1);         /* Clear all protected fields   */
CALL ASFCLR(2);         /* Clear all fields             */
```

Deleting an alphanumeric field, using call ASDFLD

To delete a single alphanumeric field (as opposed to clearing its contents), you must redefine it with a row-position of zero. This is a typical call:

```
/*      Field-id    Row    Column   Depth     Width    Type     */
CALL ASDFLD(3,       0,      0,       0,        0,       0);
```

After this call, field 3 ceases to exist.

To delete all the alphanumeric fields in the page (and the graphics too), you must call FSPCLR.

Positioning the alphanumeric cursor, using ASFCUR

You can set the position of the cursor with a call to ASFCUR. If the operator is expected to type some information, it is probably helpful to position the cursor at the start of the first input field:

```
CALL ASFCUR(4,1,1);      /* Position cursor at start of field 4 */
```

The first parameter is the field identifier. The other two parameters specify the row and column position of the cursor **within the field**.

Alternatively, if you specify a value of 0 for the first parameter, the second and third parameters then refer to the row and column position of the cursor **within the page**. For example:

```
CALL ASFCUR(0,20,1);     /* Position cursor at start of row 20  */
```

Querying the position of the alphanumeric cursor, using ASQCUR

In an alphanumeric application you can query the cursor position, by using this call:

```
CALL ASQCUR(CODE,F_IDENT,ROW,COLUMN);  /* Query cursor position */
```

If you set the first parameter (CODE) to 0, GDDM sets ROW and COLUMN to the page coordinates of the cursor; that is, its row and column numbers **within the page**.

If you set CODE to 1, the cursor position is returned in field coordinates. F_IDENT is set to the alphanumeric field identifier and ROW and COLUMN give the row and column position **within the field**.

If field coordinates are requested but the cursor does not lie within a field, F_IDENT is set to 0 and *page* coordinates are returned.

Where the above descriptions refer to the position of the cursor in the field, they mean the field on the screen, as opposed to your program's description of the field. In most cases, there is a one-for-one relationship between each character position of the field on the screen and each character position of the field in your program. An exception to this, and the use of ASFCUR and ASQCUR in that context, are described in “Using procedural alphanumerics for double-byte characters” on page 265.

Attribute bytes on 3270 terminals

The buffer in which a 3270-type terminal stores the data being displayed on the screen has one position for each screen cell. The data for each alphanumeric field is preceded in the buffer by a byte of information about the field's attributes. The screen position just before the actual data is therefore made inactive. Consequently, it is not good practice to define two alphanumeric fields that are horizontally adjacent. No error results but the last byte of the field on the left loses its data and appears blank.

When the data position starts in the leftmost cell of a row, the attribute byte occupies the last cell of the previous row, making that cell inactive.

The representation in the buffer includes trailing attribute bytes to end each field. The default setting for this trailing attribute is ‘auto-skip’, meaning that the cursor jumps automatically to the next unprotected field when the current field has been filled. It is permissible for the attribute byte of one field to share the same cell as the trailing attribute byte of the previous field. Therefore you need only allow a 1-column gap between your alphanumeric fields.

Alphanumeric attributes

There are two classes of GDDM alphanumeric attribute, *field attributes* that affect the whole of an alphanumeric field and *character attributes* that affect separately each character within a field.

Setting the attributes of alphanumeric fields

These attributes affect the way the terminal handles the fields, and also their appearance. Here are a some examples of the calls you can use to set the different attributes. Other possible settings of the parameters specified for these calls are described in the *GDDM Base Application Programming Reference* book.

- **Type**. This is the only attribute that has to be specified when the field is defined by an ASDFLD call (see “Defining an alphanumeric field, using call ASDFLD” on page 71). It defines handling characteristics such as whether the

field is to be protected, and whether it is a light-pen field. The type attributes can subsequently be altered by a call to ASFTYP.

For example:

```
CALL ASFTYP(21,2);     /* Change field 21 to protected type */
```

- **Intensity**. The intensity attribute of a field may be set with this call:

```
CALL ASFINT(39,2);         /*    Field 39 becomes bright   */
```

- **Color**. The color attribute of a field is set with this call:

```
CALL ASFCOL(77,1);         /*    Field 77 becomes blue     */
```

  The second parameter of this call can take any of the value 0 through 7, which produce the same colors as with the GSCOL call.•

  If the field's symbol set is multicolored, the color parameter must be set to 7 (neutral); see “Multicolored image symbols” on page 242 for more details.

- **Symbol set**. You can specify the symbol set to be used by a call such as CALL ASFPSS(6,196). Field 6 can then be displayed using the symbols of symbol-set 196 (see Chapter 12, “Using symbol sets” on page 233 for more details). A symbol set is typically a font, that is, a character set in a particular style.

- **Highlight**. This statement sets the highlighting of a field:

```
CALL ASFHLT(3,4);            /*    Field 3 is underscored    */
```

- **Field end**. The field-end attribute determines whether the next field should have the auto-skip attribute. This is a typical call:

```
CALL ASFEND(8,0);                /* Auto-skip after field 8 */
```

  0 is the default value. The alternative parameter value is 1, which specifies no auto-skip.

- **Output blank to null conversion**.

```
CALL ASFOUT(8,1)
```

  changes all the trailing blanks of field 8 to nulls on output. Trailing nulls enable the operator to use 3270 insert-mode on the field.

  A parameter setting of 0 would request no conversion (the default).

  This is an output function only, and does not affect field contents as returned by ASCGET.

- **Input null to blank conversion**.

```
CALL ASFIN(8,2)
```

  requests conversion of all nulls to blanks when field 8 is read from the screen. A parameter setting of 0 would request no conversion (the default). A setting of 1 would request conversion of all nulls except trailing ones.

  This takes place only when device input is received for this field. Otherwise, field contents remain as they are.

- **Translation tables**. A call to ASFTRN assigns tables to a field so that GDDM can translate the character strings on input or output (or both). The translation

• A suggested mnemonic for the codes for blue through neutral is: Boys Reading Politics Go To Yale Now

tables themselves are established by calling ASDTRN, as described in the *GDDM Base Application Programming Reference* book.

- **Transparency**. You can allow graphics on the screen to extend into the cells of alphanumeric characters using an ASFTRA call (see “Device variations with procedural alphanumerics” on page 82).
- **Mixed single- and double-byte characters**. A call to ASFSEN allows a field to mix double-byte character codes with single byte by using shift control codes (SO and SI), as described in “Performing output of strings mixing single- and double-byte characters” on page 267.
- **Field outlining**. An outline can be drawn around a field with an ASFBDY call (see “Field outlining on the IBM 5550 Multistation” on page 270).

The first three fields in Figure 18 illustrate the use of field attributes.

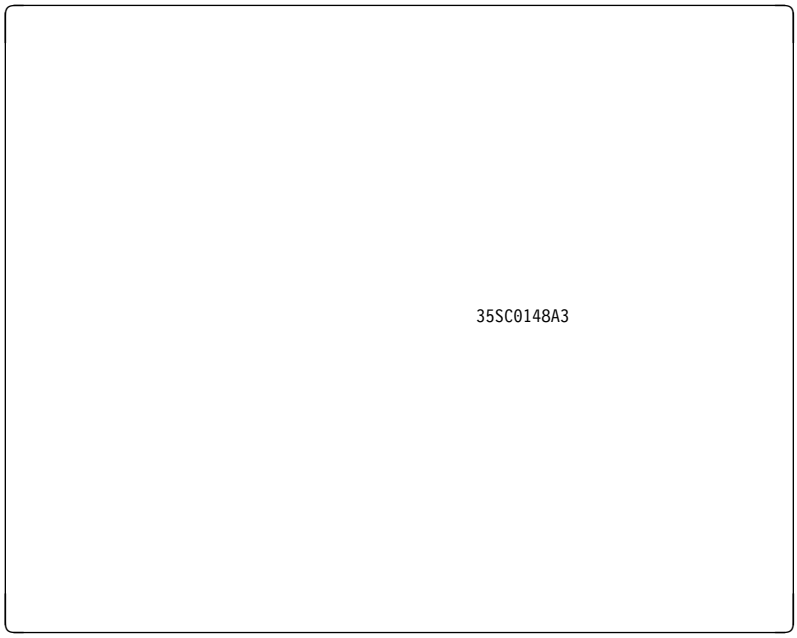


Figure 18. Using alphanumeric field attributes and character attributes

Setting the attributes of alphanumeric characters

For some of the attributes, such as color, it may be desirable to have different values within an alphanumeric field. GDDM enables you to set the character attributes for color, symbol set, and highlighting. The last three fields in Figure 18 show the effect of character attributes.

There are three calls that set character attributes. They all use a parameter containing a string of attributes; one for every character position to be specified. For example:

```
CALL ASCCOL(4,8,'2222  44');      /* Set character color    */
                                  /* attributes for field 4 */
```

The first four characters of field 4 would be set to color 2 (red). The next two would inherit the field-color attribute: this is the meaning GDDM assigns to the blanks. The seventh and eighth characters would be set to color 4 (green). Should

the field be longer than 8 characters, the remaining positions also inherit the field-color attribute.

Character attributes act on the field data rather than the field itself. They must therefore be set **after** the corresponding ASCPUT that assigns the data to the field. This rule does not apply to field attributes, which can be set at any time.

The equivalent call for setting highlighting for each character position is:

```
CALL ASCHLT(7,5,'444 1');         /* Set character highlight */
                                  /* attributes for field 7  */
```

Note: You cannot use a setting of 0 as a character-highlight attribute.

The third call used to set character attributes (ASCSS) is described in Chapter 12, “Using symbol sets” on page 233.

The terminal operator can set character attributes, if the device has the requisite keyboard. If the “red” button is pressed on the keyboard, every character entered subsequently has a character attribute of red until another color button is pressed.

The input of character attributes has to be explicitly enabled by issuing a

```
CALL ASMODE(2);
```

statement in the program. Otherwise, any input-character attributes are ignored.

The program can discover what character attributes were set by the end user by issuing these three calls:

```
CALL ASQCOL(8,6,INATR);  /* Requests that GDDM place the first  */
                         /* six color character attributes of   */
                         /* field 8 into the variable INATR     */

CALL ASQHLT(8,6,INATR);  /* Similar request for the first six   */
                         /* highlight character attributes.     */

CALL ASQSS(8,6,INATR);   /* Similar request for the first six   */
                         /* symbol-set character attributes.    */
```

For more informattion on ASQSS, see Chapter 12, “Using symbol sets” on page 233.

If no color button is pressed, newly-entered characters appear in the color of the field-color attribute. In other words, the original output-character attributes have no effect on the input to a field.

Example: Program using procedural alphanumerics to display a bank balance

Figure 19 on page 78 shows a simple alphanumeric program that gives details of a bank customer and his or her account balances in response to a typed-in account number. An example of output from this program is shown in Figure 20 on page 80.

If the account number is not recognized, an error message is issued. If an account is overdrawn, the balance is displayed in red.

```
ALPHA: PROC OPTIONS(MAIN);
DCL LIST_ACCOUNTS(4) CHAR(4) INIT('0001','0002','0005','0007');      A
DCL DEPOSIT(4) FIXED BIN(15) INIT(1247,23,-57,641);                  A
DCL CURRENT(4) FIXED BIN(15) INIT(17,-121,340,-8);                   A
DCL ADDRESS(4,3) CHAR(25) INIT(   /* Customer addresses        */    A
 'W.D.LANGHURST,','21 BLAKE COTTAGES,','ASHGROVE.',
 'G.HUCKLE,','THE RISE,','LITTLEHAMPTON.',
 'MRS. E.C.BOTERILL,','47 CURTIS ROAD,','SHERWOOD.',
 'L.M.FORRESTER,','6 VILLAGE ROAD,','ROMSEY.');
DCL ACCOUNT_NO CHAR(4);           /* Temporary variable.       */
DCL PICMONEY PIC'-$$$$$9';        /* PL/I picture variable for */
                                  /* arith to char conversion. */
DCL (AC,I) FIXED BIN(15);         /* Temporary variables.      */
DCL RED FIXED BIN(31) INIT(2);    /* Parameter constant.       */
DCL GREEN FIXED BIN(31) INIT(4);                                     B
DCL TURQ FIXED BIN(31) INIT(5);
DCL YELLOW FIXED BIN(31) INIT(6);
DCL (TYPE,MOD,COUNT) FIXED BIN(31);  /* Parameters for ASREAD  */
CALL FSINIT;                         /* Initialize GDDM        */
/******************************/
/* Define alphanumeric field */
/******************************/
/*      Field_id, Row   Column, Depth, Width,   Type */
CALL ASDFLD(1,     4,     25,     1,     21,     2);
/******************************/
/* Set field-color attribute */
/******************************/
CALL ASFCOL(1,GREEN);             /* Set field color to green   */    C
/******************************/
/* Assign data to field 1    */
/******************************/
CALL ASCPUT(1,21,'ENTER ACCOUNT NUMBER:');
CALL ASDFLD(2,4,47,1,4,1); /* Define a numeric-input-only field.*/
CALL ASFCOL(2,YELLOW);     /* Set field color to yellow.        */
DO I=1 TO 3;  /* Define alpha fields to hold customer's address */
CALL ASDFLD(I+2,I*2+13,I*4+25,1,25,2);
CALL ASFCOL(I+2,TURQ);          /* Set field color to turquoise */
END;                                 /* End of I-LOOP           */
CALL ASDFLD(6,25,5,1,16,2);          /* Define protected field. */
CALL ASCPUT(6,16,'CURRENT ACCOUNT:');/* Assign data to field.   */
CALL ASDFLD(7,25,22,1,7,2);  /* To hold current account balance */
CALL ASDFLD(8,25,45,1,16,2);         /* Define protected field. */
CALL ASCPUT(8,16,'DEPOSIT ACCOUNT:');/* Assign data to field.   */
CALL ASDFLD(9,25,62,1,7,2);  /* To hold deposit account balance.*/
CALL ASDFLD(10,32,16,1,48,2);        /* Define message field.   */
CALL ASFCOL(10,RED);                 /* Messages to be in red.  */
```

Figure 19 (Part 1 of 2). "Bank Account" program using alphanumeric functions

```
     /*****************************************/
     /* Top of loop to process account requests */
     /*****************************************/

OUTPUT:;
/****************************/
/* Position the cursor       */
/****************************/
CALL ASFCUR(2,1,1); /* Position cursor in ACCOUNT_NUMBER field. */
CALL ASREAD(TYPE,MOD,COUNT); /* Send output to screen and await */
                             /* a reply.                        */   D
IF TYPE¬=0 THEN GOTO ENDIT;  /* End if interrupt not ENTER.     */
/****************************/
/* Retrieve data from field  */
/****************************/
CALL ASCGET(2,4,ACCOUNT_NO); /* Retrieve entered account number.*/

DO AC=1 TO 4;       /* See if requested account number is valid.*/
IF ACCOUNT_NO=LIST_ACCOUNTS(AC) THEN GOTO VALID_ACCT;
END;                                              /* END AC-LOOP */
        /* Invalid or blank account number. Issue error message.*/
CALL ASCPUT(10,48,'INVALID OR BLANK ACCOUNT NUMBER. PLEASE RE-ENTER');
CALL FSALRM;                 /* Sound the alarm.                */   E
GOTO OUTPUT; /* Branch to top of loop to send message to screen.*/
VALID_ACCT:;                      /* Requested account is valid.*/
CALL ASCPUT(10,23,'PRESS ANY PFKEY TO QUIT'); /* Reset message  */
                                              /* field.         */
PICMONEY=CURRENT(AC);      /* Convert balance to character form */

IF CURRENT(AC)<0 THEN CALL ASFCOL(7,RED);   /* Red,   if debit. */
                 ELSE CALL ASFCOL(7,GREEN); /* Green, if credit.*/
CALL ASCPUT(7,7,PICMONEY);  /* Put current balance into field 7 */
PICMONEY=DEPOSIT(AC);      /* Convert balance to character form.*/
IF DEPOSIT(AC)<0 THEN CALL ASFCOL(9,RED);   /* RED,   IF DEBIT  */
                 ELSE CALL ASFCOL(9,GREEN); /* GREEN, IF CREDIT */

CALL ASCPUT(9,7,PICMONEY);  /* Put deposit balance into field 9 */
DO I=1 TO 3;          /* Put customer's address into fields 3-5 */
CALL ASCPUT(I+2,25,ADDRESS(AC,I));
END;                                                /* End I-LOOP */
GOTO OUTPUT;          /* Branch to top of loop to send out data.*/
ENDIT: CALL FSTERM;                            /* Terminate GDDM */
%INCLUDE ADMUPINA; /* Include declarations of GDDM entry points */
%INCLUDE ADMUPINF;
END ALPHA;
```

Figure 19 (Part 2 of 2). "Bank Account" program using alphanumeric functions

Points illustrated by the Bank Account program

The program uses account and customer data stored in program variables declared at **A**. In a real-life program the account data would probably be held on a data base. A read to the data base would follow the entry of the account number.

The three parameters returned by ASREAD **D** indicate the type of terminal interrupt caused by the operator. If the operator replies to the output by pressing the ENTER key, the parameter TYPE is set to zero. If TYPE is set to some other value, the operator must have pressed another key, such as a PF key; this is taken to mean that the program should terminate.

The program illustrates a technique for improving the readability of programs: for parameters that are constants, a variable is declared with an appropriate name and initialized to the constant value. At **B** there is an example of such a declaration, and at **C** of the use of such a variable.

The FSALRM call **E** does not cause the terminal alarm to sound immediately. It sounds on the next screen output.

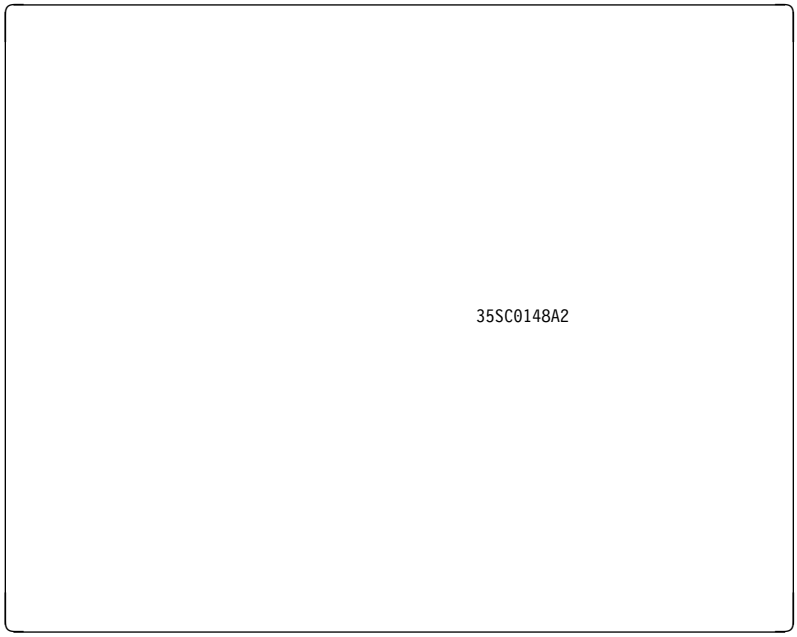


Figure 20. Output from "Bank Account" example alphanumerics program

Mixing alphanumeric and graphic functions

You can freely mix GDDM calls that refer to alphanumerics with those that refer to graphics. The alphanumeric and the graphical data are added to the current page, although GDDM holds them separately. The creation of a graphics segment, for example, has no bearing on the definition of an alphanumeric field. They are separate things.

When a screen transmission is requested (by calling ASREAD), GDDM sends first the graphics, then the alphanumerics. Those hardware cells that are part of alphanumeric fields contain no graphics at all – only the alphanumerics appear (except in the case described in "Device variations with procedural alphanumerics" on page 82).

The program shown in Figure 21 on page 81 is an example of mixing alphanumerics and graphics.

```
/**************************************************/
/*  This program accepts a typed-in part-number.   */
/*  It responds by sending a drawing of the part   */
/*  to the display screen.                         */
/**************************************************/
SPARES: PROC OPTIONS(MAIN);
DCL (TYPE,MODE,COUNT) FIXED BIN(31); /* Parameters for ASREAD.  */
DCL PART_NO CHAR(4);                 /* Temporary variable.     */
CALL FSINIT;                         /* Initialize GDDM.        */
/*     Field_id, Row, Column, Depth, Width,  Type               */
CALL ASDFLD(1,    2,    25,     1     20,     2); /* Define..   */
CALL ASDFLD(2,    2,    48,     1,     4,     1); /* ..alpha..  */
CALL ASDFLD(3,   30,    35,     1,    20,     2); /*   ..fields */
CALL GSUWIN(0.0,100.0,0.0,120.0);    /* Define coordinate system*/
CALL ASCPUT(1,20,'TYPE IN PART NUMBER:'); /* Prompt to operator */
LOOP:;
CALL ASFCUR(2,1,1);               /* Put cursor on input field. */
CALL ASREAD(TYPE,MODE,COUNT);     /* Send out data stream.      */
IF COUNT=0 THEN GOTO LOOP;        /* Try again if no part       */
                                  /* number typed.              */
IF TYPE¬=0 THEN GOTO ENDIT;       /* End run if PF key          */
                                  /* was pressed.               */
CALL GSCLR;                       /* Clear previous graphics.   */
CALL ASCGET(2,4,PART_NO);         /* Retrieve part number.      */
IF PART_NO='0001' THEN CALL WRENCH;       /* Draw part 0001,    */
                                          /* if required.       */
ELSE IF PART_NO='0002' THEN CALL HAMMER;  /* Draw part 0002,    */
  .                                       /* if required.       */
  .
ELSE GOTO LOOP;                   /* Part number was not valid. */
/***********************************/
/* This subroutine draws a wrench  */
/***********************************/
WRENCH: PROC;                     /* Subroutine to draw wrench. */
CALL GSSEG(0);                    /* Create graphics segment.   */
CALL ASCPUT(3,7,'WRENCH');        /* Display name below diagram.*/
CALL GSMOVE(20.0,95.0);           /* Move to top of wrench.     */
  .
  .
  .
CALL GSSCLS;                      /* Close graphics segment.    */
END WRENCH;                       /* End of subroutine.         */
```

Figure 21 (Part 1 of 2). Parts catalogue alphanumerics program

```
/***********************************/
/* This subroutine draws a hammer  */
/***********************************/
HAMMER: PROC;                      /* Subroutine to draw hammer. */
CALL GSSEG(0);                     /* Create graphics segment.   */
CALL ASCPUT(3,6,'HAMMER');         /* Display name below diagram.*/
CALL GSMOVE(42.0,90.0);            /* Move to top of hammer.     */
   .                               /* Continue drawing hammer.   */
   .
   .
CALL GSSCLS;                       /* Close graphics segment.    */
END HAMMER;                        /* End of subroutine.         */
   .
   .                               /* Other subroutines to draw  */
   .                               /* various spare parts.       */
ENDIT:;
CALL FSTERM;                       /* Terminate GDDM.            */
%INCLUDE ADMUPINA;                 /* Include GDDM entry points  */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END SPARES;                        /* End of program.            */
```

Figure 21 (Part 2 of 2). Parts catalogue alphanumerics program

Device variations with procedural alphanumerics

3179-G, 3192-G, 3472-G, 3270-PC/G and /GX, and IPDS printers

As explained in “Mixing alphanumeric and graphic functions” on page 80, hardware cells used for alphanumerics contain, by default, no graphics. On the 3270-PC/G and /GX workstations, 3179-G, 3192-G, and 3472-G color display stations, and on IPDS printers, you can allow the cell background to become transparent. The alphanumeric characters overpaint the graphics without blanking them out over the entire cell area.

You make a field transparent with the ASFTRA call:

```
CALL ASFTRA(19,1);   /* Make field 19 transparent */
```

When printing the current page on the 4224 printer, the transparency or opaqueness of an alphanumeric field is also honored.

IBM 5080 and 6090 graphics systems

Applications that use only the 3270 feature do not need GDDM/graPHIGS.

To use the GDDM alphanumeric input and output calls on the 5080, either the 5080 screen must be switched into 3270 mode by the user, or the 5080 must be associated with a 3270-family terminal to make up a dual-screen workstation. Alphanumeric output goes only to the 3270 screen. For more information about the 5080, see “On the IBM 5080 and 6090 graphics systems” on page 231.

Note: The information given here about the 5080 applies equally to the IBM 6090 graphics system.

5550-family multistations

If Japanese 3270-PC/G software Version 5 or later is used with the display memory expansion card, hardware cells used for alphanumerics contain, by default, no graphics. If you define a field as transparent, only blank or null characters are then transparent.

Chapter 6. Image basics

This section introduces the basic concepts of GDDM image processing and illustrates them with programming examples.

The main use of GDDM image processing is in electronic document handling, often called the “paperless office.” The document could be, for example, an office form of printed text complete with handwritten signature and annotation, a monochrome photograph, a service manual page, or an engineering drawing.

Hardware required for image processing with GDDM

Three devices that cater specifically for image processing are the IBM 3117 and 3118 Scanners and the IBM 3193 Display Station:

- The 3117 is a flat-bed scanner. The 3118 has a roller-feed mechanism. Both devices scan documents and convert them into electronic image data. They can each be attached to the 3193 terminal.
- The 3193 terminal not only displays image data on a screen, but can also carry out some image processing itself, taking some of the load off the host processor.

In this section and in Chapter 17, “Using GDDM’s advanced image functions” on page 339, the 3118 is the scanner assumed as the input device, and the 3193 the output device, except where stated otherwise.

Another device that is useful for image processing is the IBM 4224 printer, which can print image data in addition to alphanumerics and graphics.

GDDM also supports image functions on a range of other devices. These are covered in “Device variations for image” on page 370.

How images are defined for processing by GDDM

Images are pictures made up of two-dimensional arrays of dots called **pixels**. GDDM supports images comprising monochrome pixels that are either on or off. These are bi-level images, in which each pixel is represented by a single bit, which is set to 0 for “black” and 1 for “white.”

As shown in Figure 22 on page 86, in terms of GDDM processing, there are three kinds of image–device images, application images, and stored (GDDM) images.

Device images are those image arrays associated with an image scanner (input), or display, or printer (output). They are usually held in main storage, but can also exist in the 3193’s own storage, or in the 3117 or 3118.

Application images are intermediate image arrays in main storage, independent of any device. An application image can be a copy of some processed (for example, scaled) form of the device image captured by a scanner, or it may have been created or accessed by a program without reference to a scanner device. It may be an image in preparation for eventual display or printing.

© Copyright IBM Corp. 1982, 1996

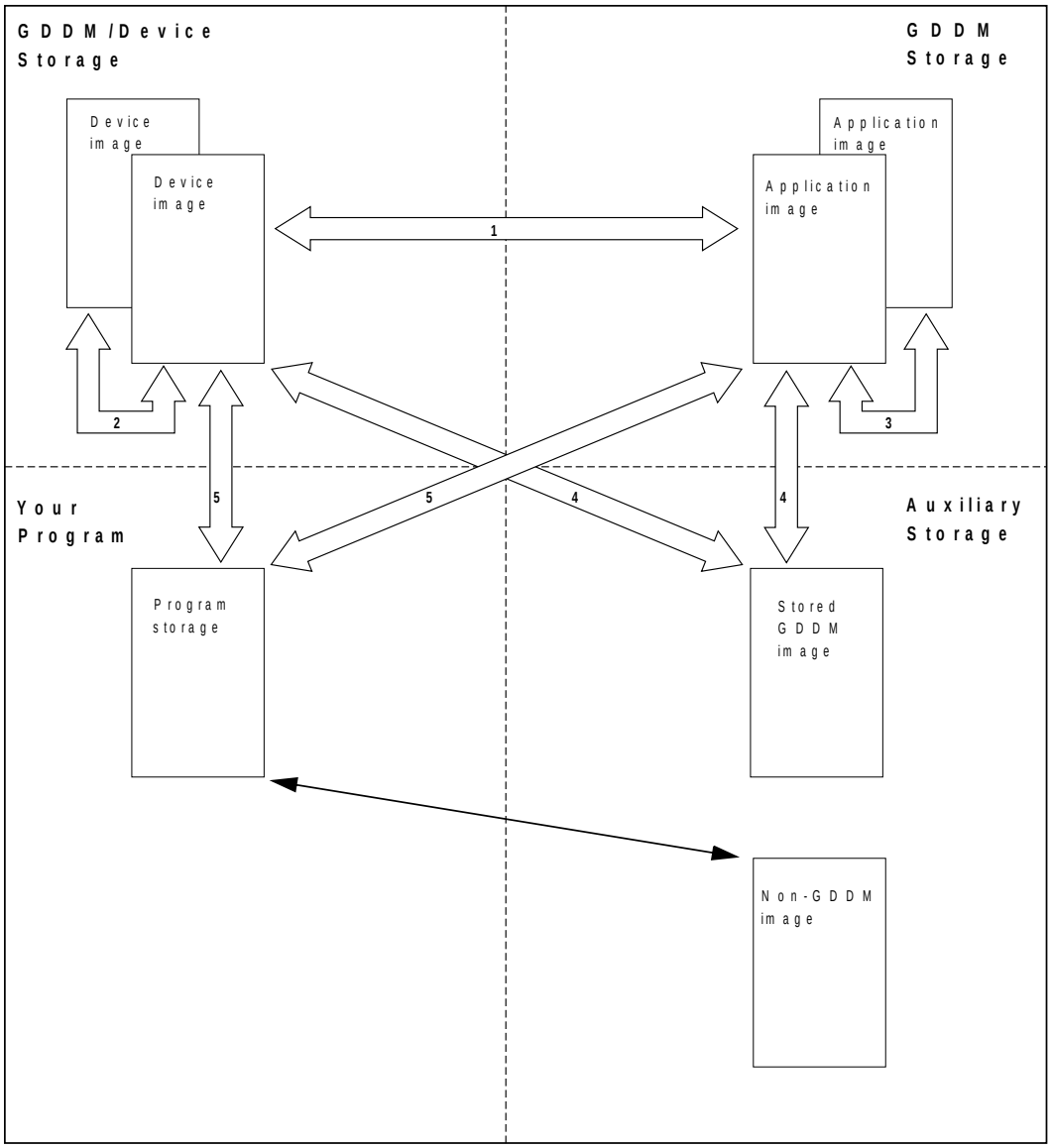


Figure 22. Image processing

Stored (GDDM) images are the result of transferring image data from either a device or application image to disk storage. Once you have stored image data, you must restore it to a device or application image before you can manipulate it. Stored images are sometimes referred to as GDDM image objects.

Device and application images are identified by fullword integers. The calls in the image application programming interface refer to images using these identifiers. In between capturing your image data, and displaying or storing it, you do most of your image processing using application images. Stored images are identified by 8-character names.

You can also hold images in your own nonGDDM image file format. You can read these files into your program using methods that are dependent on your programming language and subsystem. You can then use the image calls to transfer the image data from your program into a device or application image, and then, if you want, into a stored GDDM image. You could also do the reverse

operation, transferring image data from a device or application image into your program, using image calls, and writing it to files in your own format.

Transferring image data from one type of image to another

In order to process some image data that has been fetched by a scanner or is stored in auxiliary storage, you must transfer it into your application where you can make changes to it. When these changes have been made you can transfer the altered image into storage, to a device, or even to another program.

Transfer operations, as illustrated in Figure 22 on page 86, entail copying of GDDM image data from:

1. A device image to an application image, or the converse
2. A device image to another device image
3. An application image to another application image
4. Auxiliary storage to a device or application image, or the converse
5. A device or application image to storage arrays in your program, or the converse

The image from which data is fetched is called the **source** image, and the image to which data is sent is called the **target** image.

The first four operations in the above list are described in this section. The fifth operation is described in “Transferring images into and out of your program” on page 355.

In the course of a transfer operation, a **projection** is applied to the transferred image by GDDM. A projection is an image manipulation procedure that you specify by one or more **transforms**. Transforms are the edit operations applied to the image data during the transfer. Specifying the **identity projection** simply tells GDDM that no editing is to take place, and so a simple copy operation results. Projections can be saved and restored.

When a projection is applied in a transfer operation, the source image is unchanged, unless the target and source are the same image. Later sections in this section describe projections and transforms in more detail.

How to scan, display, and save an image

The next example program scans a 6-inch by 4-inch document on a 3118, to produce a scanner-device image. It then transfers the captured data to an application image, using the identity projection. During the transfer operation, the captured image is displayed on a 3193 so that an operator can view it. If the image on the display looks all right to the operator, it is saved on auxiliary storage.

The program is written with the assumption that the document to be scanned is in the feed tray of the scanner and there are no conditions to prevent a scan from taking place. “Querying image devices” on page 339, describes some of the scanner error conditions that can arise, and what you can do about them.

The example introduces some of the image processing calls. The next eight sections describe them in more detail, and at the same time introduce some more of the concepts of image processing.

```
IMPROG1: PROC OPTIONS(MAIN);
DCL (ATTYPE,ATTVAL,COUNT) FIXED BIN(31);
CALL FSINIT;
CALL ISESCA(1);          /* Echo scanner image on display screen*/  A

/*      image-id  h-pixels v-pixels type res unit h-res v-res   */
CALL IMACRT( -1,    1440,    960,     0,  1,  0,  240.0,240.0);    B
                       /* ...Creates a 1440 by 960 pixel image */
                       /* with defined resolution of           */
                       /* 240 pixels per inch,                 */
                       /* giving a 6 by 4 inches picture       */
CALL ISLDE(-1);        /* Load scanner paper                   */  C

/*       source-id  target-id      projection-id                */
CALL IMXFER( -1,        12,              0);                      D
                       /* Scan the image and transfer it to    */
                       /* application image 12                 */
                       /* (echoing it on display screen)       */
CALL IMADEL(-1);       /* Delete scanner image & eject paper   */  E
CALL ASREAD(ATTYPE,ATTVAL,COUNT); /* Read input from keyboard   */  F
IF ATTYPE=1 THEN;      /*   Exit if any PF key pressed         */
 ELSE                  /* Save the scanned image               */
  DO;
         /* im-id proj-id  name   count  description     protect*/
   CALL IMASAV(12,   0,  'MYIMAGE', 16, 'CLIENT SIGNATURE', 1);    G
                      /* Save image 12 in file 'MYIMAGE',     */
                      /* with protected write operation       */
   CALL IMADEL(12);   /* Delete image 12                      */  H
  END;
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;
END IMPROG1;
```

Figure 23. Simple image program – scan, display, and save an image

Scanner echoing

The ISESCA call at **A** switches on scanner echoing. This means that whenever a transfer operation from a scanner takes place, an echo (copy) of the target image is produced at the display device. Such a transfer operation takes place at **D**. An ISESCA parameter value of 1 enables echoing; a value of 0 disables it. Whenever possible, echoing is done by the 3193, and so requires no host processing.

Creating an image

You can use the IMACRT call to create a device or application image. In the example, the IMACRT at **B** creates a scanner device image that receives the image data from the scan of the document. This scanner image is the source of the impending transfer operation.

The parameters have the following meanings:

- The first parameter is the identifier of the image. You use image identifiers when creating images, and when transferring data between images. You always use the identifier –1 for a scanner device image. You use positive values for application images. The display device image (identifier 0) cannot be created using this call, but you can use the ISFLD call (define an image field) to create an image with identifier 0.
- The second and third parameters specify the horizontal and vertical size of the image, in pixels. The example is written assuming that documents to be scanned are 6 inches horizontally, and 4 inches vertically. You have a choice of three pairs of defined resolutions on the 3117 and 3118 scanners.

  This program uses 240/240, so the size of the image in pixels is fixed at 6 x 240 = 1440 horizontally, and 4 x 240 = 960 vertically.
- The fourth parameter, 0, indicates default image type, meaning this is a bi-level image.
- The fifth parameter indicates whether or not the image is to have a defined resolution.

  The value `1` indicates that a resolution for the image will be defined in the next three parameters. (A value of `0` is used for raw image data, for example a computer-generated array of pixels, of no particular physical size.)
- The sixth parameter specifies the units used to define the resolution of the image in the last two parameters:

  `0` Inches
  `1` Meters
- The last two parameters specify horizontal and vertical resolution, in this case both 240 pixels per inch.

So, now that the program has created the scanner device image, it is ready to scan the document.

Loading the document into the scanner using call ISLDE

For the 3118, the ISLDE call at **C** feeds the document from the feed tray into the scanner. If there is a document already present **inside** the scanner, it is ejected, and the document in the feed tray is fed in. The call has only one parameter, the identifier of the scanner device image. A scanner device image must have been created before this call can be used.

The 3118 aligns the top edge of the paper with the top of the (empty) created scanner device image, and centers the paper laterally.

For the 3117, the ISLDE call resets the scanner so that it is ready to scan from the top of the paper.

The paper size does not have to match that implied by the IMACRT parameters. For example, the program creates a scanner device image that is 6 inches horizontally by 4 inches vertically. If a document that is 8 inches horizontally by 5 inches vertically is fed top-first into the scanner, the program captures the middle 6 inches of the document horizontally, and the top 4 inches vertically.

Transferring images using call IMXFER

You can use the IMXFER call for transfer operations from:

- A device image to an application image, or the converse
- A device image to a device image
- An application image to an application image

In the example, the IMXFER call at **D** causes the scanner to scan the document into the scanner device image. It then transfers the data from that image to an application image that it implicitly creates.

Here is the call again:

```
CALL IMXFER(-1,12,0);
```

The parameters are as follows:

- The first parameter is the identifier of the source of the transfer. In the example it is –1, meaning the scanner device image.
- The second parameter identifies the target image. The target can be either an image that already exists, or an image that does not yet exist. If it does not yet exist, as in the example, IMXFER creates a target image of sufficient size in GDDM storage, and gives it the identifier in this parameter. (12 is arbitrarily chosen.) In the example, the target image is created with the same resolution as the source.
- The third parameter tells GDDM the identifier of a projection to be applied during the transfer. Projection 0 is the identity projection. This simply means that a copy takes place.

  Do not worry about projections at this stage. They are described later in this section.

The ISESCA at the beginning of the program sets scanner echoing on. In the example, it takes effect when IMXFER is called, and has the same effect as if a

```
/*    SCANNER-IMAGE-ID  DISPLAY-IMAGE-ID  PROJECTION-ID  */
CALL IMXFER( -1,                0,             0);
```

was processed simultaneously to the IMXFER that is already there. Image identifier 0 is always used for the display device image.

Scanner echoing gives you a copy at the screen of the target of the transfer operation from the scanner. The resolution of the 3193 display screen is 100 pixels per inch. In the example, GDDM implicitly creates a target with the same resolution as the source – 240 pixels per inch – but the 3193 performs the resolution conversion necessary to display the echo at the same size as the target. Therefore, because the identity projection is applied, a copy takes place, and the image echoed on the screen is the same physical size as the original document – 6 inches horizontally and 4 inches vertically.

Deleting images using call IMADEL

You can use the IMADEL call to delete an image. Its one parameter is the image identifier of the image that you want to delete. In the example, there are two IMADEL calls. The call at **E** has the scanner device image identifier of –1 as its parameter. When this is specified, for a 3118, the call not only deletes the scanner image but also ejects the document from the scanner.

You can use this call at any time after the scanner image transfer operation call; you can use it when you have completely finished scanning, or when you come to the end of scanning a particular type of document, when you want to create a new, different scanner image.

Or, if you required the same image size and resolution parameters to be used for further input documents, you could use ISLDE(-1) again, without having to explicitly delete or recreate the image.

The IMADEL call at **H** deletes the application image. Because of the large amounts of data involved in image processing, it is good practice always to delete an image as soon as you no longer need it. After you have deleted an image, you can reuse its identifier for another image.

Synchronizing output and input

The ASREAD call at **F** is still required, as with graphics, to handle the interaction with the operator. In the example, this alphanumeric input is performed using a PF key or the ENTER key.

Saving images using call IMASAV

You can use the IMASAV call, as at **G**, to copy image data from a device or application image to auxiliary storage. It is another of the calls used for transfer operations. Here is the call again:

```
/*      SOURCE-ID PROJ. FILENAME   STR.LEN.    DESCRIPTION      OVERW.*/
CALL IMASAV(12,    0,   'MYIMAGE',   16,    'CLIENT SIGNATURE',   1);
```

This call specifies that the source image with an identifier of 12 is to be copied, without undergoing any projection, to a file called MYIMAGE in auxiliary storage. A description of 16 characters in length is to be saved with the image file. It can be restored when the file is restored.

The last parameter, 1, specifies the action to be taken if a file already exists with an identical filename to that specified in the third parameter. A value of 0 means that the existing file is to be overwritten. A value of 1 specifies that an existing file with the same filename is to be protected. If you try to save to a protected file, GDDM issues an error message telling you that the file already exists.

On VM/CMS this call would create a file called:

```
MYIMAGE ADMIMG A1
```

The naming conventions for saved image files differ according to the subsystem. They are explained in the *GDDM Base Application Programming Reference* book.

Loading an image, using call IMARST

The image saved by the IMASAV example in the previous section can be loaded from auxiliary storage to a device or application image, using the IMARST call. Here is an example:

```
DCL DESCR CHAR(30);     /* File description             */
/*        ID   PROJ  FILENAME   STR.LEN. DESCRIPTION    */
CALL IMARST(0,   0,   'MYIMAGE',   30,       DESCR);
```

The parameters are similar to those of the IMASAV call. The first parameter of the IMARST call specifies an identifier for the image into which a saved image is to be restored. The value `0` used in the example specifies that the saved image is to be restored to the display. Alternatively, you can restore an image to an application image.

If you have created a projection (see “Creating a projection using call IMPCRT” on page 95), you can supply its identifier on the second parameter of the IMARST call to apply it to the image as it is restored. The call in this example applies the identity projection.

To restore the description of a saved image you need to specify the number of characters you want returned from the description, (30 in this case), and then supply a variable name into which these characters are to be returned.

Saving and restoring are transfer operations. You can, therefore, apply a projection during either or both of the operations.

That completes the description of the calls used in the first example, but, before you learn more about projections, here are two calls that were not used in the example:

Obtaining a new image identifier, using call IMAGID

When you **create** a device image, –1 is the only image identifier that you can use. A value of 0 cannot be used with IMACRT. It is reserved for the current display or printer device image, and can only be used in certain transfer operations that are described later in this section.

When you create an application image, you can use any unused integer value, in the range 1 through 2^{30}–1, as its identifier.

Another way is to use the IMAGID call to reserve a valid unused and unreserved identifier in the range 2^{30} through 2^{31}–1. The format of the call is:

```
CALL IMAGID(ID);
```

The identifier value is returned in ID, which must as usual be declared to be a fullword integer variable. You should use values in this range only if they have been returned by IMAGID, as GDDM internally uses some of the values in this range.

Querying image attributes

You can use the IMAQRY call to query device image or application image attributes. The most common use of this is to check the attributes of a target image following a transfer operation.

Here is an example:

```
CALL IMAQRY(ID,H_PIXELS,V_PIXELS,IM_TYPE,
                          RES_TYPE,RES_UNIT,H_RES,V_RES);
```

The parameter list matches that of the IMACRT call. ID and RES_UNIT are specified by the caller; the remaining parameters are returned by GDDM.

Projections

A **projection** is a sequence of changes to the image, defined by your program, that can be applied during any transfer operation. GDDM lets you define a projection in advance of its use, and independently of the image that it is to act upon. A projection can also be saved and restored.

You can use a projection to perform editing operations on an image during a transfer operation. For example, if you are processing a particular type of legal document, you know that each has some information in common with the rest, such as several paragraphs of legal jargon. They also contain information that is unique to each document, such as names, addresses, and signatures. You are interested only in extracting the unique information, as there is no point in keeping lots of copies of the same thing. You can use a projection to extract just the information you want, maybe from different parts of the document, and exclude the rest. For example, you can then rotate, reposition, or change the size of the extracted information in the target of the transfer operation.

Each individual operation in a projection is known as a transform. A projection containing a transform is illustrated in Figure 24 on page 94. A projection can contain more than one transform. This is illustrated in Figure 25 on page 95.

A transform is a composite editing function, consisting of several transform elements. GDDM applies the function to the source image, and creates a temporary intermediate image to hold the result of each transform element. The final result is subsequently placed in the target image. The temporary intermediate image is called the extracted image.

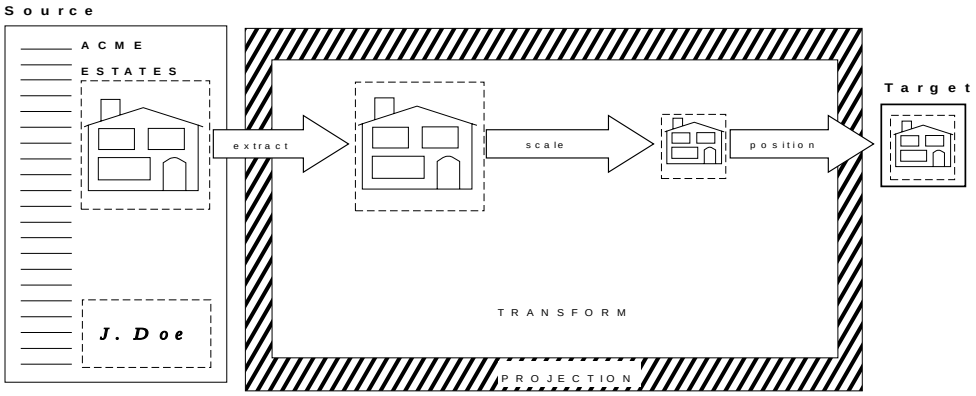


Figure 24. Projection containing a transform

Transform elements can define any or all of the following:

Extraction Defining a rectangular sub-image to be extracted from the source image

Scaling Changing the size of the extracted image

Rotation Reorienting the extracted image

Reflection Flipping over the extracted image

Negation Converting the extracted image to its “photographic negative.”

(There are also three transform elements that specifically relate to scanning. These are covered in “Scanning gray-scale images” on page 340.)

Apart from any transform elements, a transform **must** contain:

1. A definition of the location in the target image where the extracted image is to be placed. This definition also specifies how the extracted image is to merge with the target image.

And can contain:

2. A specification of the pixel generation/deletion algorithm to be used:
   - When the size of an extracted image is altered by a scaling operation
   - When image data is copied between two images whose resolutions differ.

Transform elements operate on the extracted image in isolation, independent of the target image. Items 1 and 2 above are not called transform elements, because they both affect the way that the extracted image combines with the target image.

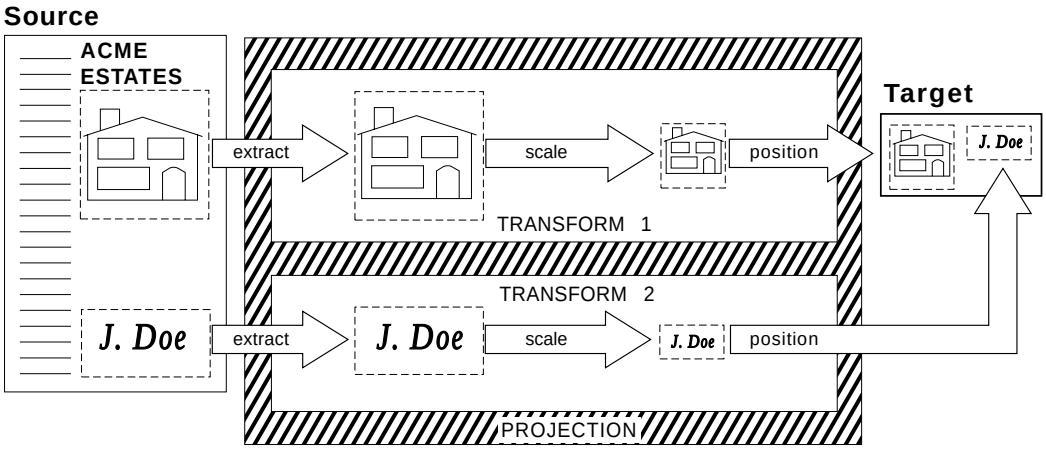


Figure 25. Projection containing two transforms

Example code to define and save a projection

Although you can define a projection in the program that uses it, in practice you would probably build up a library of projections for standard documents, and restore them as needed. Here is a piece of code that defines and stores the projection shown in Figure 24 on page 94. The projection contains one transform. The effect of the transform is to extract a 5 inches x 5 inches rectangular sub-image, alter its size, and position it in the top-left-hand corner of the target. Later on, another example program restores and uses the projection.

```
CALL IMPCRT(15);             /* Create projection with id of 15 */

     /* proj-id  coord-type  l-edge  r-edge  top-edge  bot-edge */
CALL IMREXR(15,       0,       3.0,   8.0,     2.0,    7.0);

     /* proj-id  h-scale  v-scale                               */
CALL IMRSCL(15,    0.5,     0.5);

     /* proj-id  coord-type  horiz-posn  vert-posn  mix-mode    */
CALL IMRPLR(15,       0,        0.0,       0.0,        0);

/* THIS CALL COMPLETES THE PROJECTION                           */

     /* proj-id     name    count     description       protect */
CALL IMPSAV(15,  'EXTRACT',   57,
     'Extract a 5 x 5 sub-image & convert it to 2.5 x 2.5 image');

CALL IMPDEL(15);             /* Delete projection 15            */
```

Creating a projection using call IMPCRT

The IMPCRT call begins a projection definition, so it must always be the first call in a projection definition. The single parameter specifies the projection identifier, by which you refer to the definition as you add transforms to it, and by which you identify it when applying it in a transfer operation, or saving it. All IMPxxx calls and all transform calls have the projection identifier as their first parameter.

Do not confuse projection identifiers with image identifiers; they are independent, so you could also use 15 as an image identifier in the same program.

See “Getting a new projection identifier, using call IMPGID” on page 105 for how to obtain an unused value from GDDM.

Extracting a rectangular sub-image using call IMREXR

You can use the IMREXR call to define a rectangular sub-image to be extracted from the source image. The left and right edges of the sub-image are defined in terms of their distance from the left edge of the source image. The top and bottom edges of the sub-image are defined in terms of their distance from the top edge of the source image.

The parameters are as follows:

- The first parameter, 15, is again the projection identifier.
- The second parameter specifies the type of coordinates used in the third, fourth, fifth, and sixth parameters:

  0 Inches
  1 Meters.
  2 Fractional. You specify at what fraction of the pixel dimensions the edges are to be, by values in the range 0.0 to 1.0.

- The last four parameters specify, respectively, the left edge, right edge, top edge, and bottom edge values in the stated coordinate type. So, in the above example:

  The left edge of the sub-image is 3 inches from the left edge of the source image.

  The right edge of the sub-image is 8 inches from the left edge of the source image.

  The top edge of the sub-image is 2 inches from the top edge of the source image.

  The bottom edge of the sub-image is 7 inches from the top edge of the source image.

There is another call that you can use instead of IMREXR. The call IMREX enables you to specify the sub-image boundary in pixel coordinates: Here is an example of its use:

```
CALL IMREX(24,0,499,20,249);
```

This list explains the parameters used here:

24 The projection identifier

0 The left edge of the required image, in pixel coordinates (0 is the left hand edge of the source image)

499 The right edge of the required image, in pixel coordinates

20 The top edge of the required image, in pixel coordinates

249 The bottom edge of the required image, in pixel coordinates

You can use either IMREX or IMREXR in a transform; you cannot use both. If you use one of them, it must be the first call of the transform. If you do not have either an IMREX or IMREXR call in a projection, or if you use the identity projection, the whole of the source image is extracted.

If the specified rectangle in a IMREX or IMREXR call lies wholly or partly outside the source image, the part outside the source image is filled with zeros. This is not an error condition.

Changing the size of an extracted image using call IMRSCL

You can use the IMRSCL call to alter the size of an image.

The parameters are as follows:

- The first parameter is again the projection identifier.
- The second and third parameters specify the x- and y-scaling factors respectively, x being horizontal and y being vertical.

Completing the image transform and positioning it in the target image

You can use either the IMRPL or the IMRPLR call to complete an image transform. Which call you choose affects how you position the transform in the target image. If you use the IMRPL call, you must specify the position in terms of pixels; and if you use the IMRPLR call you must specify the position in terms of real coordinates, such as inches or meters. With either call, you can specify how the transform will mix with any image data already in the target image. Either IMRPL or IMRPLR **must always** complete an image transform. Until a transform has been completed with one of these calls, the projection is not available for use in a transfer operation.

If the rectangle specified in an IMRPL or IMRPLR call extends outside the target image, the transformed image is clipped to the target rectangle boundaries.

If the target image does not exist before the transfer operation, it is created. For an IMRPL or IMRPLR call to other than the (0,0) pixel position, a target is created consisting initially of zero-value pixels, of the minimum size necessary to contain all the rectangles at the positions specified, without clipping. Remember that a projection can contain more than one transform, so there may be more than one sub-image rectangle.

For a description of how to define a projection comprising more than one transform. See “Putting transform calls in the right sequence” on page 103.

Saving a projection using call IMPSAV

Having defined a projection, you may want to save it. It could then be invoked later by a different program to that in which it was defined. In a way similar to images, projections can be saved on disk, using IMPSAV.

Here is the example call again:

```
CALL IMPSAV(15,'EXTRACT',20,'CREATE 5x5 SUB-IMAGE',0);
```

The parameters are as follows:

- The first parameter specifies the projection identifier.
- The second parameter specifies the filename. Naming conventions vary according to the subsystem. They are explained in the *GDDM Base Application Programming Reference* book. On VM, GDDM creates a file with the specified name as the file name, and a file type of ADMPROJ. So the example would create a file called

```
EXTRACT ADMPROJ A1
```

- The third specifies the length of the character string following.
- The next parameter gives a file description that is saved with the file.
- The last parameter specifies whether existing files with the same filename are to be protected. It has the same effect as the last parameter on the IMASAV call; 0 to allow overwrite, 1 to protect an existing file.

Projections are restored with the IMPRST call:

```
DCL DESCR CHAR(20);     /* For file description                */
CALL IMPRST(101,'EXTRACT',20,DESCR);
```

The parameters are as follows:

- The first specifies the projection identifier to be associated with the restored projection.

  You do not have to use the projection identifier that applied when the projection was saved.
- The second specifies the filename. The same remarks apply as for the equivalent parameter of the IMPSAV call, described above.
- The third gives a count of description characters to be used
- The fourth is the variable name, DESCR, to receive the string.

Deleting a projection, using call IMPDEL

The IMPDEL call deletes a projection from GDDM storage. It has one parameter, the identifier of the projection that you want to delete. The example deletes projection 15, because it has been stored away for later use. The projection identifier 15 can now be reused. It is good practice to delete projections that you no longer need.

How to apply a projection during a transfer operation

There are a few projection and transform calls that you have not yet seen. They are described after the next example.

The first programming example in this section contains a transfer operation, during which the identity projection was applied. Applying the identity projection means that image data from the whole of the source image is used and is not changed during the transfer operation.

What follows is a more complicated version of the earlier program. The most important difference is that this time the program restores the projection defined in “Example code to define and save a projection” on page 95. It contains a transform that specifies the extraction of a 5 x 5 inches rectangular sub-image from the captured 6 x 4 inches document. The projection is then applied during the transfer operation from the scanner device image to the application image.

Once again, the program is written assuming that there are no physical scanner conditions to prevent scanning of the document. See “Querying image devices” on page 339 for how you can check for those conditions.

```
IMPROG2: PROC OPTIONS(MAIN);

DCL P_WIDTH FLOAT DEC(6);     /* Scanner paper width           */
DCL P_DEPTH FLOAT DEC(6);     /* Scanner paper depth           */
DCL HOR_RES FLOAT DEC(6);     /* Scanner horizontal resolution */
DCL VER_RES FLOAT DEC(6);     /* Scanner vertical resolution   */
DCL APPL_ID FIXED BIN(31);    /* Application image identifier  */
DCL DEVICE_DEPTH FIXED BIN(31);  /* Device depth in rows       */
DCL DEVICE_WIDTH FIXED BIN(31);  /* Device width in columns    */
DCL ARRAY(2) FIXED BIN(31);      /* Array for device queries   */
DCL IMAGE_FIELD_DEPTH FIXED BIN(31);/* Image field depth       */
DCL IMAGE_FIELD_ROW FIXED BIN(31);  /* Image field top row     */
DCL TGIMNAME CHAR(8);            /* Saved image name           */
DCL TGIMDSC CHAR(20);            /* Saved image description    */
DCL PROTECT_FLAG FIXED BIN(31) INIT(0);/* Allow over-write of  */
                               /* existing image or projection*/
DCL SAVE_FLAG BIT(1);            /* On to save image           */
DCL PROJ_ID FIXED BIN(31);       /* Projection id              */
DCL PROJ_NAME CHAR(8);           /* Projection name            */
DCL PROJ_DSCR CHAR(60);          /* Projection description     */

CALL FSINIT;                     /* Initialize GDDM            */

/* Format the display screen for alphanumerics and image       */
/* Call A/N routine 1 to create alphanumeric fields            */
CALL ANR1;                       /*                            */  A
/* Fit an image field to the remainder of the screen           */
CALL FSQURY(0,3,2,ARRAY);        /* Query device default page  */  B
                                 /*         depth and width    */
DEVICE_DEPTH=ARRAY(1);           /* Depth in rows              */
DEVICE_WIDTH=ARRAY(2);           /* Width in columns           */
IMAGE_FIELD_DEPTH=DEVICE_DEPTH-2;/* For 2 alpha rows           */
IMAGE_FIELD_ROW=2+1;             /* For 2 alpha rows           */
CALL ISFLD(IMAGE_FIELD_ROW,1,    /* Create image field         */  C
           IMAGE_FIELD_DEPTH,DEVICE_WIDTH,1);

/* Scanner parameters                                          */
CALL ISESCA(1);                 /* Echo scanned image on screen*/
/* Set up scan paper size and scanner resolutions              */  D
P_WIDTH=6.0;                     /* Paper width in inches      */
P_DEPTH=4.0;                     /* Paper depth in inches      */
HOR_RES=240.0;      /* Horizontal and ...                      */
VER_RES=240.0;      /* vertical resolution (pixels per inch)   */
```

Figure 26 (Part 1 of 2). The "IMGPROG2" program

```
                                /* Create the scanner image   */
CALL IMACRT(-1,P_WIDTH*HOR_RES,
                      P_DEPTH*VER_RES,
                      0,1,0,
                      HOR_RES,VER_RES);
CALL ISLDE(-1);                 /* Load sheet of paper        */

/* Scan the image, and transfer to an implicitly created       */
/*  application image, using a restored projection             */
CALL IMAGID(APPL_ID);           /* Get a new image identifier */
CALL IMPGID(PROJ_ID);                                          E
CALL IMPRST(PROJ_ID,'EXTRACT',20,PROJ_DSCR);
CALL IMXFER(-1, APPL_ID,PROJ_ID);/* Transfer operation        */ F

/* Call A/N routine 2 to get name and description of image     */
CALL ANR2;                                                     G
/* Save the application image if user wants to                 */
IF SAVE_FLAG='1'B THEN          /* Save the image             */
   CALL IMASAV(APPL_ID,0,TGIMNAME,20,TGIMDSC,PROTECT_FLAG);    H

CALL IMADEL(APPL_ID);          /* Delete the application image */
CALL IMPDEL(PROJ_ID);          /* Delete the projection        */
/* Eject the scanner paper                                     */
CALL IMADEL(-1);
CALL FSTERM;                    /* Terminate GDDM             */

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;

END IMPROG2;
```

Figure 26 (Part 2 of 2). The "IMGPROG2" program

At **A**, an application subroutine ANR1 is called, but the coding of this is not shown because it is concerned solely with alphanumerics. It is required to create, on the default GDDM page, four alphanumeric fields, two being for output prompting messages, and the other two for alphanumeric input. You can create these using procedural alphanumerics calls. high-performance alphanumeric calls, or mapped alphanumerics, whichever you prefer. See "Comparison of the three methods of implementing alphanumeric functions" on page 56.

The rest of the code in IMPROG2 assumes that the ANR1 routine has created the necessary alphanumeric fields using only the top two rows of the screen. The code between the points marked **B** and **C** then creates an image field, using the remainder of the screen, however many rows that implies on the display device used. This is an example of device-independent coding, and is good practice.

The FSQURY call, used at **B**, is used to query the default page depth and width for the device. The depth and width are then used to set the size of the image field so that it fills the screen space that is not used by the alphanumeric fields. Other uses of the FSQURY call, in image processing, are discussed in "Querying image devices" on page 339.

The ISFLD call, used at **C**, is also discussed in “Combining an image with text or graphics” on page 365.

At **D**, the scanner image size and resolutions specified are the same as in the previous program, but they have been assigned to variables that are then used in parameter expressions in the subsequent IMACRT statement. The resulting image is the same as before.

At **E**, the IMPGID call is used. This is described in “Getting a new projection identifier, using call IMPGID” on page 105, and is similar to IMAGID, already met.

At **F**, IMXFER transfers the scanner image to a target application image implicitly created by GDDM, applying the projection just restored.

The earlier ISESCA call ensures that the extracted images are echoed on the display screen.

The application image is used as the source image of the subsequent IMASAV at **H**. Remember that image save and restore are transfer operations.

Some further remarks on transfer operations follow:

- It is an error to invoke a projection without first having created it.
- If the target image exists before a transfer operation, its attributes override those of the transformed image. Here it does not previously exist, with the effects noted under the first example program (“Transferring images using call IMXFER” on page 90) – it is created with the same attributes as the transformed image.

At **G**, another alphanumerics routine, ANR2, also not shown, is called. This routine is required, by use of procedural or mapped alphanumerics, to allow the terminal user to key the file name, and optionally a file description, under which the image is to be saved. These are to be supplied in the variables TGIMNAME and TGIMDSC respectively. For example, you can use the ASCPUT, ASREAD, and ASCGET procedural alphanumerics calls for this purpose.

The routine ANR2 is also required to set a program flag, SAVE_FLAG, on when the currently displayed image is to be saved, or off when it is not. Again, the alphanumerics sections of this book illustrate the use of ENTER and PF keys for this kind of end-user choice.

The remaining transform elements

In addition to the transform elements already covered in the example projection, there are three more that you can use.

Turning (reorienting) the image through multiples of 90 degrees

The IMRORN call is the transform element call for reorienting images.

An example of this call is:

```
CALL IMRORN(9,2);
```

The parameters are as follows:

- The first parameter, 9, is the projection identifier.
- The second parameter specifies the change in orientation of the extracted image:

  0 No rotation
  1 90 degrees clockwise rotation
  2 180 degrees rotation
  3 270 degrees clockwise rotation (90 degrees counterclockwise).

Reflecting the image about a chosen axis, using call IMRREF

You can use the transform element call IMRREF to reflect an extracted image about a chosen axis. Here is an example:

```
CALL IMRREF(99,1);
```

The parameters are as follows:

- The first parameter, 99, is the projection identifier.
- The second parameter specifies how the extracted image is to be reflected:

  1 Horizontal reflection (left to right)
  2 Vertical reflection (top to bottom).

  Some other settings of this parameter are permitted and are defined in the *GDDM Base Application Programming Reference* book.

Getting the negative of an image, using call IMRNEG

Here is an example of the IMRNEG transform element call, that you use to get the “photographic” negative of an extracted image:

```
CALL IMRNEG(2);
```

This negates each pixel in the extracted image so that “black” pixels become “white,” and conversely.

Defining the resolution conversion algorithm, using call IMRRAL

A transform can contain not only transform elements and a definition of the position in the target image for the extracted image, but also an algorithm to be used where the size or resolution of an image are altered.

Figure 27 on page 103 diagrammatically shows the pixels making up the top-left-hand corner of a black square displayed at:

- Same size, but different resolution
- Same resolution, different size

The diagram is not to scale. It simply shows that, if you change the size of an image at constant resolution, or change the resolution at constant size, new pixels must be generated, or existing ones deleted.

For a size or resolution increase, pixels must be generated, and may simply be replications. However, for a size or resolution reduction, pixels must be deleted.

There are then different effects according to whether “black” or “white” pixels are deleted when they are adjacent.

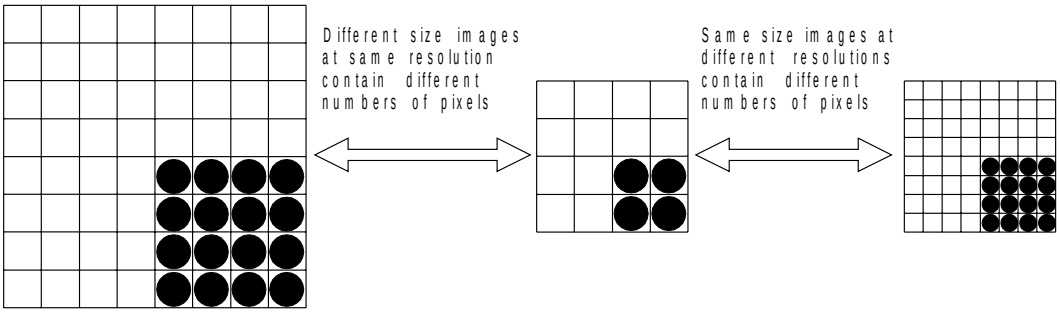


Figure 27. Resolution conversion

If the extracted image and target image in a transfer operation have different defined resolutions, GDDM automatically converts the data from the extracted image resolution to the target image resolution and applies the algorithm.

If the source or target image has undefined resolution, image manipulations are done using pixel to pixel mapping.

You can use the IMRRAL call, always within a projection definition, to set the resolution/scaling algorithm of a transform, before the transform is completed by an IMRPL or IMRPLR call. Here is a typical call:

```
CALL IMRRAL(101,2);
```

The parameters are as follows:

- The first parameter, 101, is the projection identifier.
- The second parameter specifies one of the following algorithms, which are further defined in the *GDDM Base Application Programming Reference* book:

  0 The default algorithm, the same as 1.
  1 Pixel replication when scaling up, deletion when scaling down.
  2 Pixel replication when scaling up, black pixel retention when scaling down. This is an improvement on the default algorithm, for images containing black on white text or graphics.
  3 Pixel replication when scaling up, white pixel retention when scaling down. This is an improvement on the default algorithm, for images containing white on black text or graphics.

Putting transform calls in the right sequence

Remember that:

- A projection definition must begin with the IMPCRT call
- A projection definition can contain one or more transform sequences, each of which must contain as a minimum an IMRPL or IMRPLR call, and can contain other transform calls.
- If an IMREX or IMREXR call is used, it must be the first call in a transform sequence.
- An IMRPL or IMRPLR call must be the last call in a transform sequence.

This then is an example of a valid projection definition:

```
CALL IMPCRT(...); /* Begin projection definition                */
CALL IMREX(...);  /* Transform 1: extract sub-image 1           */
CALL IMRPL(...);  /* Transform 1: place sub-image 1 in target   */
CALL IMREX(...);  /* Transform 2: extract sub-image 2           */
CALL IMRSCL(...); /* Transform 2: scale sub-image 2             */
CALL IMRORN(...); /* Transform 2: reorient sub-image 2          */
CALL IMRPLR(...); /* Transform 2: place sub-image 2 in target   */
                  /* (end of projection, or further transforms  */
                  /*   can follow)                              */
```

Note that although a projection cannot be changed once it has been defined, it can be added to.

Order of evaluation in projections

When a projection is invoked, the operations specified within it are evaluated. The order of evaluation is as follows:

1. The transforms that comprise the projection are evaluated in turn, in the order in which they were specified. Evaluation of a transform involves evaluation of the transform elements that it contains, in the order in which these were specified.
2. If the target of the transfer operation already exists, there may be implied global operations, such as “convert to target resolution,” that are needed for the transformed image to be merged with the target. These implicit operations are performed at this stage for each transform independently.
3. The result of each transform is merged into the target, in the order in which the transforms were specified.

Some other facilities

Gray-scale image manipulation

See “Scanning gray-scale images” on page 340 for calls that enable you to control the brightness and contrast processing and the conversion of image type. The associated calls are IMRBRI, IMRCON, and IMRCVB respectively.

Applying a projection during image save and restore

The previous program example invoked a projection in the IMXFER call statement. You can also invoke a projection when using the IMASAV or IMARST calls, for example:

```
CALL IMASAV(-1,101,'DOCIMAGE',12,'SCALED IMAGE',0);
```

to modify the scanner image by projection 101 before saving it.

Getting a new projection identifier, using call IMPGID

When you create a projection, you can use any unused integer value, in the range 0 through 2^{30}–1, as a projection identifier.

Another way is to use the IMPGID call to reserve a valid, unreserved projection identifier in the range 2^{30} through 2^{31}–1. The format of the call is:

```
CALL IMPGID(ID);
```

The identifier value is returned in ID. You should use values in this range only by calling IMPGID, as GDDM internally uses other values in this range.

Changing the image resolution type, using call IMARF

You have seen that the fifth parameter of the IMACRT call specifies an image as having defined or undefined resolution. You can put resolution values in the IMACRT call, and specify in the fifth parameter that the resolution is undefined. Those resolution values are not used unless you use the IMARF call to change this image attribute. Suppose you have previously created image 12 with undefined resolution. Then

```
CALL IMARF(12,1);
```

would change it to having defined resolution. A value of 0 in the second parameter would do the reverse. You can use IMAQRY to query the existing image attributes.

Editing images without a transfer operation

There are three calls not involving a transfer operation, that you can use to alter an image “in-place.” The calls are known as **in-place transforms**, and are evaluated immediately. That is, they are not coded within projections.

Clearing a rectangle in an image, using call IMACLR

Here is a typical call:

```
CALL IMACLR(7,110,500,0,325);
```

The above example would clear, within image 7, the rectangle extending from pixel column 110 through column 500, and from pixel row 0 through row 325, inclusive.

- The first parameter, 7, is the image identifier.
- The second and third parameters, 110 and 500, are the left edge and right edge of the rectangle to be cleared, in pixel coordinates.
- The last two parameters, 0 and 325, are the top edge and bottom edge of the same rectangle, in pixel coordinates.

Trimming an image, using call IMATRM

Here is a typical call:

```
CALL IMATRM(12,55,700,10,156);
```

- The first parameter, 12, is the image identifier.
- The next two parameters, 55 and 700, are the left and right edges of the required image, in pixel coordinates.

- The last two parameters, 10 and 156, are the top and bottom edges of the required image, in pixel coordinates.

This call is useful for reducing the amount of data to be processed and stored.

Converting the resolution of an image, using call IMARES

You may recall that the IMACRT call specifies whether or not an image has defined resolution, and if so, what are the resolution values. You can use IMARES to change the resolution of an image. Here is a typical call:

```
/*          ID  UNIT   HORIZ_RES   VERT_RES   ALGOR
CALL IMARES(51,   0,      300,       180,       3);
```

If you code -1 to specify the image scanner, you can only specify scanner-supported resolutions in the third and fourth parameters. You can use the ISQRES call to query the scanner-supported resolutions. See “Querying image-related device characteristics” on page 343.

The permitted values for the parameters and their meanings are described fully in the *GDDM Base Application Programming Reference* book.

The effect of IMARES depends on whether the image was created with defined or undefined resolution, or subsequently changed to defined or undefined resolution by the IMARF call.

If the image has undefined resolution, the image data itself is not changed, but the resolution returned by a subsequent IMAQRY call reflects the new values.

If the image has defined resolution, the image data is converted to the new resolution.

Using IMXFER with target image the same as source image

This is a permitted use of the IMXFER call. Suppose you have an application image 12 and want to operate on it using projection 3, for example, to derive its negative image (by use of the IMRNEG call within the projection). Then if you code

```
CALL IMXFER(12,12,3);
```

the image as processed by projection 3 replaces the source image in application image 12.

This is true if the projection specifies overpaint mixing mode. Otherwise, the source and target images are merged according to the mixing mode specified (in IMRPL or IMRPLR).

Chapter 7. Hierarchy of GDDM concepts

You have already seen that you can code simple graphics programs with only a little knowledge of graphics concepts. This section looks at some of the concepts that you can use in more advanced programs. GDDM has a hierarchy of objects, that governs the physical subdivision of the screen or printer page. Each object in the hierarchy nests inside the one above.

These are the objects in the hierarchy:

1. The device
2. The partition set
3. The partition
4. The page
5. The graphics field, alphanumeric field, or image field
6. The picture space
7. The viewport
8. The graphics window

   When these eight objects have been defined or defaulted, you can open:

9. The graphics segment
   This cannot be defaulted, and it is not mandatory.

10. The graphics primitive
    You can draw graphics primitives without explicitly opening a graphics segment on the current page. If you do this, it causes objects above the segment in the hierarchy that have not been defined to take default values. You cannot save such primitives.

Notes:

1. Objects that are specified explicitly must be defined in the appropriate order (that is, moving down the hierarchy). For example, you can define object 2 and then object 3, or objects 1, 4, and then object 6. You cannot define object 6 and then object 4.
2. The first eight objects are present in all GDDM programs, whether the output is graphics, alphanumeric text or image. You do not need to specify any of them explicitly in your program. You can leave them to take the default values.
3. Virtual devices and operator windows fit into the hierarchy at the level of the device. They are introduced in “The device” on page 108.
4. Nongraphics objects such as alphanumeric fields, alphanumeric maps and image fields fit into the hierarchy at the same level as the graphics field. Only the graphics hierarchy extends below this level.
5. The graphics objects 1 through 7 in the above hierarchy are concerned with the physical subdivision of the display screen (or printer page). Objects 8, 9 and 10 are concerned with drawing graphics primitives in the viewport.

   The objects 1, 4, 5, 6, and 7 are shown in Figure 33 on page 117.

To understand which of all the objects in the hierarchy you need to define explicitly for a given program, you need to consider the nature and purpose of each one.

© Copyright IBM Corp. 1982, 1996

The device

The device is the highest level object in the hierarchy. Your program can select a device to be used either as the current **primary device**, or as the current **alternate device.**

Except for the few calls that refer to the current alternate device, all commands refer to the current primary device unless the program selects a different device as the current primary device. Likewise, the calls that are specific to the alternate device apply to the current primary alternate device unless a different alternate device is made current.

The call that opens a device is DSOPEN. The call that makes a device the current primary or secondary device is DSUSE. If you want your program's output to appear only on the screen of the invoking device, then you don't need to issue either of these calls. This is because GDDM issues internal DSOPEN and DSUSE calls to make the invoking terminal the default current primary device. However, if your program is to communicate with devices other than the invoking terminal, or if you want a greater degree of control over *any* device, including the invoking terminal, you can issue your own DSOPEN with different parameters or use **nicknames** to modify the internal DSOPEN.

For more information on how to use and control different devices, see Chapter 18, "Device support in application programs" on page 371.

Virtual devices

If the DSOPEN call for the current primary device uses processing option 24, (or is modified by a nickname with the (WINDOW,YES) option) you can divide the screen of the display device into one or more **operator windows**. Operator windows are rectangular subdivisions of the screen that have a different virtual device appearing in each.

Each virtual device can belong to a different instance of GDDM, and each instance of GDDM can belong to a different application program. This means that you can have a different GDDM application running in each operator window, sharing a display device concurrently.

The first DSOPEN that specifies (WINDOW,YES) opens the real device. Subsequent calls of DSOPEN for that same device open **virtual devices**. Each application program then communicates with the terminal operator through an operator window conceptually situated in front of a virtual screen, and can behave as if it had sole control of a real screen. Therefore, the terminal user can communicate through several operator windows with **several** applications that are running at the same time. An important use of this function is in task manager programs. You can find more information on this under "Using operator windows to write task-manager programs" on page 479.

By means of the User Control facility, the end user of your application program can control the size, position, and viewing priority of operator windows on the screen.

You can also use operator windows to provide the various functions of a single application in separate and independent areas of the screen. With such an application, the terminal user can communicate through several operator windows with a single application.

Real or virtual screens, viewed through operator windows, can themselves be subdivided into **partitions.**

The partition set

For a device in your application, you can create several logical screens. Each logical screen is called a **partition set**.

The partition set defines a grid of rows and columns on which you can specify the size and position of one or more rectangular partitions. When a partition set is created, it becomes immediately current and is associated with the primary device.

The partition set is created by the PTSCRT call. It is used to define a grid of rows and columns for specifying the sizes and positions of all the partitions that it contains.

Only one partition set per virtual device can be shown to the terminal user at any one time, but the terminal user can view and interact with an application through more than one partition within each partition set.

The partition

Partitions are independent rectangular subdivisions of the screen, which can overlap each other. You can create a partition using the PTNCRT call. If you issue a PTNCRT call without first creating a partition set, a default partition set is created, covering the complete screen.

The default partition size fills the partition set grid. For details of the default partition-set grid size, see the description of the FSQURY call in the *GDDM Base Application Programming Reference* book. One use of partitions is to clearly define subsets of output from an application.

For example:

Partition 1 containing an alphanumeric menu of actions on a picture

Partition 2 containing a graphics picture being changed

Partition 3 containing graphics editing help information

In your program, you can use the PTNMOD call to change the attributes (size, position, viewing priority, and visibility) of partitions.

Figure 28 on page 110 illustrates a partitioned screen based on the default partition set grid.

GDDM partitioning is supported on all directly attached display devices. The IBM 3193 Display Station, 3290 Information Panel, and 8775 Display Terminal each have a hardware-partitioning facility. For terminals in the 3270 family, such as the IBM 3472-G, 3192-G, 3179-G, and 3279, GDDM emulates hardware partitioning. For all display devices, partitions are emulated when operator windows are used, when partition overlap is specified, or when User Control has been made available to the terminal user.

35SC0867E1

Figure 28. Creating partitions

A partition belongs to the partition set that is current when the PTNCRT call is issued. Each partition can have one or more **pages** associated with it.

35SC0867E2

Figure 29. Defining pages within partitions (each partition has its own pages)

For more information on using partitions in programs, see Chapter 22, “Designing end-user interfaces for your applications” on page 453 and the *GDDM Base Application Programming Reference* book.

The page and page window

A page is a rectangular area that can contain the program's alphanumeric, graphic, and image output, and any alphanumeric or graphic input entered by the terminal user. It is the basic unit of display. The calls, ASREAD, GSREAD, MSREAD, and FSFRCE send the current page of each partition in the current partition set to the primary device, and (except in the case of FSFRCE) await the terminal user's input.

A page belongs to the partition that is current when it is created. On devices with hardware partitions, input can come from one partition. When GDDM emulates the partitioning, the input can come from more than one partition. In both cases, the cursor position determines the current partition. You can have more than one page belonging to each partition, but only the current page in each partition is displayed.

Usually the page size matches the partition (or the printer page), but when you create the page, using the FSPCRT call, you can request a size that is smaller than the partition (or printer page) or, on display devices, larger than the partition. With the FSPWIN call, you can control how much of the page is shown on a display device by setting the **page window**. When you create a page, GDDM sets the page window to the same size as the page or partition, whichever is smaller. The page window always lies within the page and partition boundaries and is positioned where you specify it on the page. The top-left-hand corner of the page window coincides with the top-left-hand corner of the partition. If your program specifies a page that is larger than its partition, the terminal user can view the page through the page window, by scrolling the page up and down, and from side to side, behind the page window. The relationship between the page and the page window is described in “Large and small pages” on page 473.

A page may be subdivided into any or all of these:

- A graphics field
- An image field
- Procedural alphanumeric fields
- Mapped alphanumeric fields.
- High-performance alphanumeric fields.

A graphics field and an image field can exist on a page at the same time, but cannot overlap. Most devices cannot display both an image field and a graphics field. See “Combining an image with text or graphics” on page 365 for details. An alphanumeric field cannot overlap another alphanumeric field, but can overlap a graphics field or an image field.

If the page is to be mapped, it must be created by an MSPCRT call, rather than an FSPCRT, as explained in “A mapping application that sets up a dialog with the end user” on page 289.

Figure 29 on page 110 shows a screen (assumed to be 32 by 80 for the example) divided into two partitions, with a page open in each one.

When a page is created or defaulted it becomes the current page. The size of a page cannot be altered after creation.

If you do not explicitly define a page but nevertheless issue a call that needs a containing page (for example, define graphics field), GDDM uses the default page. The default page covers the whole partition or printer page and has a page identifier of zero.

If, in a program, you delete the current page, the default page becomes current. It is impossible to delete the default page.

A discussion of a programming example using two pages is given in “Concepts introduced by the TWOPAGE program” on page 123.

The graphics field and the image field

One step below the page in the hierarchy is the graphics field. This is used when the graphics is to occupy only part of the current page – for example, when alphanumeric text is to occupy the rest of it exclusively. You can define the size and positioning of the graphics field on the page using the GSFLD call. Figure 30 shows a graphics field of 22 rows by 60 columns lying inside a page of 27 rows by 80 columns.

In this and later illustrations, the partition is not shown. A single partition occupying the complete screen is assumed.

35SC0867E3

Figure 30. GSFLD – defining a graphics field

If no graphics field is specified, but reference is made to some object lower in the hierarchy (such as a picture space), the default graphics field is created. This covers the whole of the current page.

Only one graphics field is permitted per page. If you define a second, it deletes the first one and all its contents.

Notes:

1. On some terminals, such as the 3278 and 3279, the bottom right-hand hardware cell contains an attribute byte and is therefore not available for graphics. If your graphics extend to the bottom right-hand corner of the graphics field (for example, if you have a frame round the edge), you may want to exclude the bottom row (or the rightmost column) of the screen from the GSFLD specification. Otherwise, you get a permanent one-cell blank on this edge.
2. If your application requires the graphics field to be framed by thick lines, you should remember that thick lines, where they touch on the edge of partitions, may be subject to clipping. See “Graphics clipping” on page 125. A line is thickened by drawing a second line either above or to the right of it, so when a graphics field with a thick frame is the same size as the partition, the upper and right-hand sides are clipped.

The picture space

This is the part of the graphics field in which graphics are drawn. Because the graphics field is specified in terms of physical rows and columns, its aspect ratio (that is, the ratio of its width to its depth) varies from device to device.

If, however, your program creates a drawing in which dimensions and proportions are important, such as a map or a building plan, you need a picture space that has the same ratio regardless of the device used for output. To ensure a particular ratio for your drawing, you must define a picture space explicitly, using the GSPS call.

Subject to the requested ratio, GDDM creates the largest possible picture space within the graphics field. Either the horizontal or the vertical boundaries coincide with those of the graphics field.

Figure 31 on page 114 shows the effect of three different picture-space definitions, each of which lies inside the graphics field that was defined in the previous section.

35SC0867E4

Figure 31. Defining a picture space

If the picture space is not explicitly defined and an object lower in the hierarchy is defined, the picture space defaults to the size of the whole graphics field

The viewport

The viewport is the area of the screen to which the current graphics are to be sent. It is defined as part of the picture space. If it is not explicitly defined, using the GSVIEW call, the viewport covers the whole picture space by default.

The viewport is the lowest physical division of the screen in the hierarchy. Figure 32 shows the positioning of the viewport within the picture space defined with the ratio 1:0.5 in Figure 31.

35SC0867E5

Figure 32. Defining a viewport

A viewport must be defined in terms of the picture space that contains it. So if you want to define the viewport you need to know the coordinates of the picture space.

Whether you allow the picture space to take the default size or explicitly specify a ratio for it, its coordinates are set by GDDM. You must, therefore, take special action to discover the coordinates of the picture space. GDDM sets the dimensions of the picture space to maximize its size subject to the size of the graphics field and the ratio defined (explicitly or by default). Since the size of the graphics field is dependent on the device, so too is the size of the picture space.

GDDM provides API calls that allow you to query the characteristics of every object in the graphics hierarchy from the device down to the graphics segment. By issuing the GSQPS call, you can first query the coordinates of the picture space at execution time and then define the viewport in terms of the parameters returned by the query. This is a typical combination of calls:

```
CALL GSQPS(WIDTH,HEIGHT);          /* Ask GDDM what ratio   */
                                   /* the picture-space has.*/
CALL GSVIEW(0.0,WIDTH*0.5,0.0,HEIGHT*0.5);
                                   /* Place the viewport    */
                                   /* in the bottom-left    */
                                   /* quarter of the        */
                                   /* picture-space.        */
```

You do not need to explicitly define a viewport in a graphics program but it can be very useful if you need to include several pictures in one graphics field. Only one viewport can exist (per page) at a given time; but it can be redefined over and over again. Whether you need to present the same picture in several places on the screen or several different pictures, you can define the viewport's size and position on the picture space, call a graphics subroutine for each picture, and redefine the viewport in the position where you want the next picture.

Assume, for example, that SUNSHINE, CLOUD, and RAIN are three subroutines that draw emblems on the weather map of a local T.V station. On a day when the forecast says “Rain in the South West of the region, cloudy skies in the West and North West, and sunshine in all other areas”, the weatherman might program his chart in the following way:

Note: The subroutines have to be reexecuted for each new viewport.

```
CALL GSQPS(WIDTH,HEIGHT);
                    /* Ask GDDM what ratio the picture space has. */
CALL GSVIEW(0.0,WIDTH*0.33,0.0,HEIGHT*0.33;
                                        /*Viewport bottom left   */
CALL CLOUD;                             /* Draw the cloud emblem */
CALL RAIN;                              /* Draw the rain emblem  */
CALL GSVIEW(0.0,WIDTH*0.33,HEIGHT*0.33,HEIGHT*0.66);
                                        /* Viewport center left  */
CALL CLOUD;                             /* Draw the cloud emblem */
CALL GSVIEW(0.0,WIDTH*0.33,HEIGHT*0.66,HEIGHT);
                                        /* Viewport top left     */
CALL CLOUD;                             /* Draw the cloud emblem */
CALL GSVIEW(WIDTH*0.33,WIDTH*0.66,HEIGHT*0.66,HEIGHT);
                                     /* Viewport center top      */
CALL SUNSHINE;                       /* Draw the sunshine emblem */  A
CALL GSVIEW(WIDTH*0.33,WIDTH*0.66,HEIGHT*0.33,HEIGHT*0.66);
                                     /* Viewport center center   */
        :
        :
 and so on for other areas
        :
        :
CALL ASREAD(TYPE,VALUE,COUNT);         /* Send picture to device. */
  /*** Subroutine to draw sunshine emblem ***/
  SUNSHINE: PROC;
  CALL GSSEG(0);        /* Open unnamed segment                   */
  CALL GSCOL(6);        /* Set current color to yellow            */
  CALL GSBMIX(5);       /* Set background mix mode to transparent */
  CALL GSAREA(1);
        :
    and so on
        :
  CALL GSSCLS;                       /* Close the current segment */
  END SUNSHINE;
```

When the viewport is redefined, it can occupy some or all of the position of a previous definition. If the weatherman forecast both cloud and sunshine for the North central area, he could add some lines such as this to his program at **A**:

```
CALL GSVIEW(WIDTH*0.36,WIDTH*0.69,HEIGHT*0.66,HEIGHT);
                           /* Viewport slightly right of center top. */
CALL CLOUD;                /* Draw the cloud emblem */
```

35SC0867E6

Figure 33. Defining a graphics hierarchy (with default partitioning)

The objects in the hierarchy considered so far are concerned with the physical subdivision of the display device's screen (or the printer page). The remaining objects in the hierarchy are different.

The graphics window

The graphics window is the name given to the system of coordinates used to position graphics primitives in the viewport.

Using the GSWIN call, you can set up the graphics coordinate system with whatever axis measurements you like:

```
CALL GSWIN(0.0,80.0,-30.0,70.0); /*Define graphics coordinate system*/
```

This call sets up a graphics coordinate system with an x-axis extending from 0 to 80 and a y-axis extending from –30 to 70.

The coordinate system fits exactly over the viewport. The x range covers the width of the viewport and the y range covers the depth.

If you choose not to define the graphics window explicitly, and then open a graphics segment or draw a primitive, GDDM uses the default coordinate system of 0 to 100 on each axis.

In the programming examples shown in earlier sections, the coordinate system addressed the whole screen. That was because all the graphics objects in the hierarchy had been allowed to take the default values. This is often the case – the partition, the page, the graphics field, the picture space, and the viewport all then occupy the whole screen. The graphics primitives may be clipped at the edges of the graphics window, as explained in “Graphics clipping” on page 125.

Note: Do not confuse the graphics window with the “page window” used in the context of scrolling. (The scrolling page window is explained in “Large and small pages” on page 473.)

Uniform world coordinates

You might expect these statements always to draw a square on the screen:

```
CALL GSMOVE( 0.0, 0.0);
CALL GSLINE(10.0, 0.0);
CALL GSLINE(10.0,10.0);
CALL GSLINE( 0.0,10.0);
CALL GSLINE( 0.0, 0.0);
```

But on most devices, they produce a rectangle with one side 10 x-units long and the other 10 y-units long. A square results only if one x-unit on the screen is physically equal to one y-unit; that is, if the height and width of the display screen are equal.

Few, if any, devices have square screens, so GDDM provides the GSUWIN call, which enables you to draw your graphics with uniform world coordinates. The GSUWIN call opens a uniform graphics window on the viewport:

```
CALL GSUWIN(0.0,80.0,-30.0,70.0); /*Define uniform coordinate system*/
```

This call has the same parameters as GSWIN and the same effect, except that GDDM ensures that the resulting world coordinates are uniform. The uniform set of coordinates is such that the specified x range and the specified y range are both contained within the viewport, and either the x range just fits the width of the viewport, or the y range just fits the height.

In general, this means that one axis contains slack: if the x range fits the width, the y range is less than the height, or if the y range fits the height, the x range is less than the width. GDDM centers the slack axis in the viewport and extends its range in both directions to the edge of the viewport. Your program can therefore draw in the slack area. To discover the actual x and y ranges, you can execute a GSQWIN call:

```
DECLARE (XMIN,XMAX,YMIN,YMAX) FLOAT DEC(6);
CALL GSQWIN(XMIN,XMAX,YMIN,YMAX);
```

By using a uniform graphics window for drawing graphics, you ensure that squares appear square, circles appear round, and the graphics you draw will appear the same no matter what device they are displayed or printed on.

An alternative to GSUWIN is to define a viewport with a width-to-height ratio (aspect ratio) equal to the ratio of the x range to the y range. If the viewport is allowed to default to fill the picture space, then the picture space must be of the same aspect ratio as the world coordinates:

```
CALL GSPS (   0.8,        1.0    );
CALL GSWIN( 0.0,80.0, -30.0,70.0 );
```

How to avoid inverting the graphics window

For all objects in the hierarchy from the partition down to the viewport, each is defined in terms of rows and columns of the one above it, with the origin in the **top** left-hand corner. The graphics window, however, has its origin at its **bottom** left-hand corner.

Some people, new to programming with GDDM, try to define coordinates for the graphics window that match the rows and columns of the viewport, like this

```
CALL GSWIN(0.0,80.0,32.0,0.0)
```

They are then rather surprised when their graphics and mode-3 graphics text appear upside-down! This is because the value specified for the bottom of the y-range, (32.0) is actually greater than that specified for the top of the y-range, (0.0).

GDDM doesn't issue an error message when a program defines a graphics window with its origin positioned somewhere other than at the bottom left-hand corner. It is valid to define the origin of the graphics window in any position. The program in Figure 34 illustrates the possibilities:

```
WTELL: PROC;
CALL FSINIT;          /* Initialize GDDM                       */
CALL GSWIN(0.0,200.0,0.0,120.0);
                      /* Define a normal graphics window.      */
CALL WILLIAM_TELL;    /* Call user subroutine that draws picture*/
                      /* of William Tell (on the left) aiming  */
                      /* crossbow at apple (on the right).     */
CALL ASREAD(TYPE,VALUE,COUNT); /*Send picture to screen and wait*/
                              /* for acknowledgement.          */
CALL GSCLR;           /* Clear the graphics field.             */
CALL GSWIN(200.0,0.0,0.0,120.0);
                      /* Reverse x-boundary coordinates.       */

CALL WILLIAM_TELL;    /* Call the same subroutine under the    */
                      /* influence of a window with its        */
                      /* x-boundary coordinates reversed.      */
                      /* William Tell is now on the right,     */
                      /* aiming crossbow at apple on the left. */
CALL ASREAD(TYPE,VALUE,COUNT);
```

Figure 34 (Part 1 of 2). Program showing how the definition of the graphics window affects the position of graphics

```
CALL GSCLR;
CALL GSWIN(0.0,200.0,120.0,0.0);
                     /* Reverse y-boundary coordinates instead.*/
CALL WILLIAM_TELL;   /* Call the same subroutine, this time    */
                     /* under the influence of a window with   */
                     /* its y-boundary coordinates reversed.   */
                     /* William Tell is still on the left,     */
                     /* but he is upside-down, aiming          */
                     /* at an upside-down apple!               */
CALL ASREAD(TYPE,VALUE,COUNT);

CALL GSCLR;
CALL GSWIN(200.0,0.0,120.0,0.0);
                     /* Both boundaries reversed.              */
CALL WILLIAM_TELL;   /* Call the same subroutine, this time    */
                     /* with both window boundaries reversed.  */
                     /* William Tell is upside-down on         */
                     /* the right-hand side of the picture.    */
CALL ASREAD(TYPE,VALUE,COUNT);
CALL FSTERM;


WILLIAM_TELL: PROC;
CALL GSSEG(0);       /* Open graphics segment.                 */
CALL GSMOVE(24.0,8.0);/* Move to start of left boot.           */
   and so on...      /* Continue drawing W.Tell and the apple. */

CALL GSLINE(175.0,80.0);/* End outline of apple's stalk.       */
CALL GSENDA;           /* Close area (apple's stalk).          */
CALL GSSCLS;           /* Close the graphics segment.          */
END WILLIAM_TELL;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END WTELL;
```

Figure 34 (Part 2 of 2). Program showing how the definition of the graphics window affects the position of graphics

The graphics segment

A segment is a collection of primitives and their associated attributes. It is **not** a physical subdivision of the screen. You can put all the primitives in your picture into one segment, or you can divide them into several segments, if that is more convenient.

Segments have attributes determining such characteristics as detectability and visibility. You can set these explicitly with a GSSATI call, or use the default settings. The attributes of an existing segment can be changed with a GSSATS call. Segments can be stored in a library on external storage, and retrieved from it when required.

Creating a graphics segment using GSSEG or specifying attributes for a segment using GSSATI, causes any undefined objects above it in the hierarchy to take default values.

The uses applications can make of graphics segments are described in Chapter 9, “Manipulating graphics segments” on page 145. It includes a section on primitives outside segments.

Redefining objects in the hierarchy

Viewports and graphics windows

After creating some graphics and closing any open segment, your application can redefine the viewport or the graphics window (still under the same higher objects in the hierarchy). This defines a new environment, and further graphics segments may be added to the overall picture. It is important to realize that changing a viewport or graphics window affects only subsequent graphics. It has no effect on graphics already drawn in a previous environment.

Picture space and graphics field

You can only redefine a picture space if it contains no graphics. To clear all existing graphics in the graphics field, you can use the GSCLR call. It is not possible to redefine the graphics field without destroying any graphics contained within it.

Other objects

The higher-level objects (page, partition, partition set, and device) have their own identifiers. You cannot define a second object using an identifier already assigned to an existing object of the same type and belonging to the same higher-level object. For instance, if device 2 already has a page 3, it is an error to attempt to create a new page with an identifier of 3. However, a higher-level object can be deleted (or closed, in the case of a device), after which its identifier can be reused.

Example: Program using the GDDM hierarchy

As an illustration of the GDDM hierarchy at work, the program in Figure 35 on page 122 creates a hierarchy with two pages and switches from one page to the other adding graphics to both and graphics text to one of the pages. It then redefines the graphics window on one page and adds more graphics to those already on that page.

```
TWOPAGE: PROC OPTIONS(MAIN);

CALL FSINIT;                                /* Initialize GDDM */
CALL FSPCRT(1,32,80,1);/* Create a page, 32 rows by 80 columns. */
                       /* This causes the device to default to  */
                       /* the invoking device.                  */
                       /* The default partition set and         */
                       /* partition will be used.               */

CALL GSSEG(0);         /* Open segment on first page, causing   */
                       /* the graphics field, the picture-space */
                       /* and the viewport all default to       */
                       /* the whole screen.                     */
                       /* A 100 by 100 window is used           */
CALL GSLINE(60.0,60.0);            /* Draw one line on page 1   */

CALL ASREAD(TYPE,VALUE,COUNT);     /* Send output from page 1   */

CALL FSPCRT(2,0,0,1);  /* Create a second page (page 2).        */   A
                       /* All further graphics or alphanumerics */
                       /* calls refer to page 2 (the new page). */

CALL GSSEG(0);         /* Open segment on second page, causing  */
                       /* the graphics field, the picture space,*/
                       /* the viewport and the window to        */
                       /* default again                         */

CALL GSCHAR(20.0,48.0,28,'GRAPHICS TEXT SENT TO PAGE 2');
                                                 /* Write text */
CALL GSSCLS;           /* Close the graphics segment on page 2  */

CALL ASREAD(TYPE,VALUE,COUNT); /* Send output from second page  */


CALL FSPSEL(1);        /* Select the first page. It is still    */   B
                       /* exactly as it was when the program    */
                       /* left it to create a second page.      */
                       /* The segment is still open and it      */
                       /* contains only one line.               */
CALL GSSCLS;           /* Close the graphics segment on page 1  */   C

CALL GSWIN(0.0,1000.0,0.0,2000.0);                                   D
                       /* Redefine the graphics window.  All    */
                       /* existing primitives on this page stay */
                       /* the same.  Any new ones are expressed */
                       /* in terms of the new world coordinates.*/
```

Figure 35 (Part 1 of 2). Program creating two GDDM pages

```
CALL GSSEG(0);          /* Open new segment on first page.       */   E
CALL GSMOVE(600.0,1200.0);                                            F
                        /* This point is equivalent, under the   */
                        /* new coordinate system, to end point   */
                        /* of the line in the first segment.     */
CALL GSLINE(700.0,1800.0);
                        /* Draw a line in the new segment        */

CALL FSPSEL(2);         /* Reselect the second page. This has    */
                        /* one (closed) segment in it,           */
                        /* containing a graphics text string     */
CALL GSSEG(0);          /* Open a new segment (unnamed).         */
CALL GSLINE(40.0,50.0);/* This line is drawn from (0,0).         */   G


CALL ASREAD(TYPE,VALUE,COUNT);
                        /* Send output from second page.         */
                        /* This output consists of two segments  */
                        /* containing two (joined) lines         */
                        /* (the first containing one text string,*/
                        /* the second containing one line).      */


CALL FSPSEL(1);         /* Reselect the first page.              */

CALL ASREAD(TYPE,VALUE,COUNT);
                        /* Send output from first page to the    */
                        /* screen. This consists of two segments */
                        /* each containing one of two lines, */
                        /* which converge on the screen.         */

CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END TWOPAGE;
```

Figure 35 (Part 2 of 2). Program creating two GDDM pages

Concepts introduced by the TWOPAGE program

- You can make a page in your program current either by creating it as at **A** or by issuing an FSPSEL call to select it as shown at **B** in the program. When you select another page for processing (or create another page, which then becomes the current page), the first page is left as it was.
- You can leave a graphics segment open on a page even when another page has become current. At **B**, when page 1 is selected as the current page again, the segment on the page is still open and graphics can be added to it if desired.
- When you redefine the graphics window as at **D**, you cannot continue using the segment that was defined in the previous environment. You must open a new segment at **E**; but before doing this you must close the first segment at **C**. You cannot open a new segment without previously closing any existing open segment.

- Whenever you open a new graphics segment, the point (0,0) becomes the current position. For this reason, a GSMOVE call is required at **F** and the line drawn at **G** emanates from the origin of the graphics window.

If you want to start a new picture on a page that already has some graphics (or alphanumerics, or both) on it, you can issue an FSPCLR to clear the page.

An alternative but less efficient method would be to delete the page (FSPDEL) and then to create it again (FSPCRT).

A graphics hierarchy with two devices

An example of the hierarchy that a program might address is shown in Figure 36. The figure shows a program communicating with two devices - device 12 and device 27. The first device has two pages, one with a graphics field and two alphanumeric fields, the other with just a graphics field. The second device has only one page, containing a graphics field and three alphanumeric fields. Neither device is partitioned.

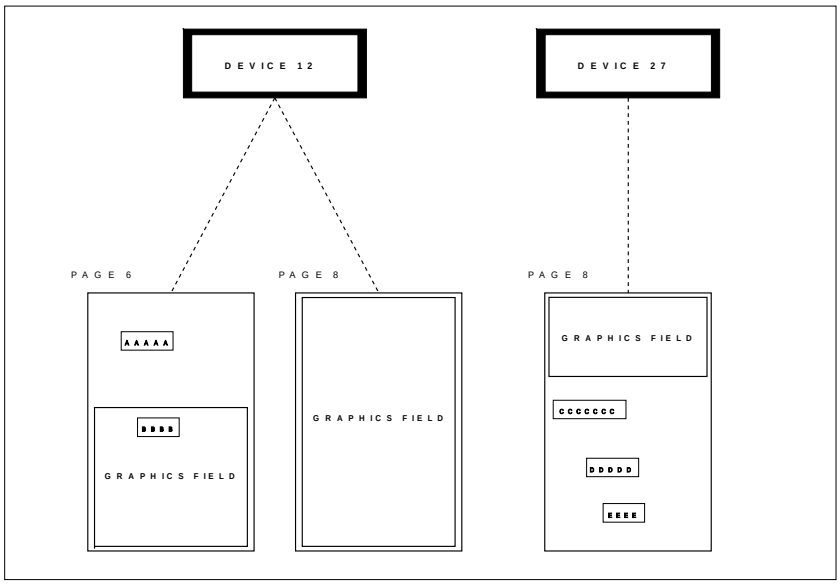


Figure 36. Example of a graphics hierarchy with two devices

While referring to the figure, note the following points:

- Symbol sets are loaded **per device**. This is the case even if the device is partitioned, and even for graphics symbol sets that are used in main storage by GDDM and not really sent to the device. If you want to use, for instance, the Italian Gothic symbol set (ADMUVGIP) on both the target devices, you have to load it twice, once when device 12 is current and again when device 27 is current.
- Alphanumeric fields and sets of default alphanumeric field attributes apply **per page**. Once you assign a number to an alphanumeric field as its identifier you cannot use the same number to identify another alphanumeric field on the same page, unless you delete the first field. You can, however, use the same identifier for alphanumeric fields that are on different pages.

- One graphics field is allowed **per page**. That graphics field can then contain several graphics segments. As with alphanumeric fields, the same identifier can be used for graphics segments that are on different pages.
- All segment attributes apply **per graphics field**. When you create a graphics field, the segment attributes always start at their default values unless you have set them previously using a GSSATI call.
- All logical input devices are associated with a **graphics field**. (Logical input devices are explained in Chapter 11, “Writing interactive graphics applications” on page 197.) If you redefine the graphics field after an input device has been enabled, or select another page, the device is disabled.
- All primitive attributes apply **per graphics segment**. When you open a new segment the primitive attributes always start at their default values (except in the case of called segments).

Graphics clipping

If you are using the default graphics window coordinates of 0 to 100 in both directions, what happens if you draw a line to a point that is outside this range? You could issue this statement:

```
CALL GSLINE(150.0,85.0);         /* Draw a line to (x=150,y=85) */
```

The call is valid, but the result depends on whether or not you have requested **clipping**, and what outer limits you have set. The limits are the **data boundary** and **segment viewing limits**.

The **data boundary** sets the outer limits, in world coordinates, of graphics primitive data to be retained by GDDM. You can use it to restrict the amount of data sent to a device. You cannot set the data boundary while a graphics segment is open.

You can set the data boundary explicitly like this:

```
CALL GSBND(0.0,100.0,0.0,100.0); /* Set data boundary */
```

The default data boundary is the graphics window. Setting the data boundary establishes a default graphics field if one has not already been created or defaulted.

Clipping at the data boundary is controlled by the GSCLP call:

```
CALL GSCLP(1);  /* Enable precise clipping for the current page */
```

The effects that the various types of graphics clipping have on a segment containing three primitives are shown in figures 37 through 39.

Note: Clipping is enabled or disabled **per page**.

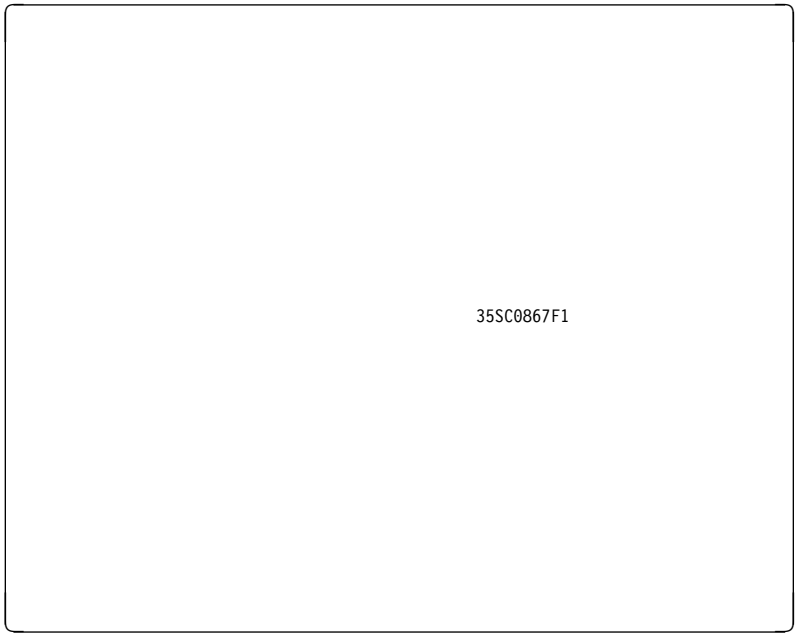


Figure 37. Graphics on both sides of the data boundary with clipping switched off

By default, the data boundary has no effect on graphic primitives that extend outside it; that is clipping is switched off. This is because most programs do not create primitives that stray outside the graphics window, and when clipping is on, it results in some extra processing by GDDM.

If you do decide to use clipping at the data boundary, you can use either precise clipping or rough clipping.

Precise clipping at the data boundary

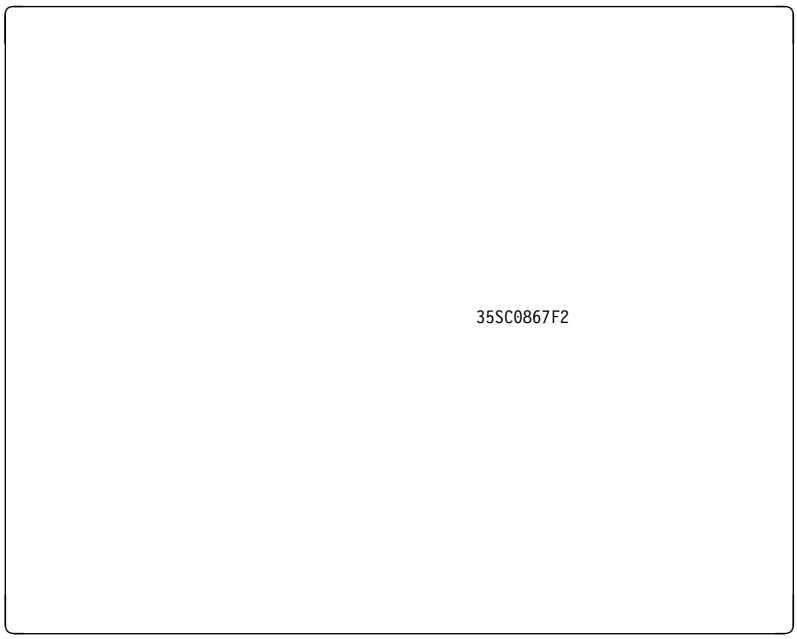


Figure 38. Graphics crossing the data boundary with precise clipping

If you specify precise clipping, all primitives inside the boundary are preserved. If a primitive lies across the boundary, the part outside the boundary is clipped away and only the part inside it is retained. Any primitives that lie completely outside the boundary are clipped away.

Rough clipping at the data boundary

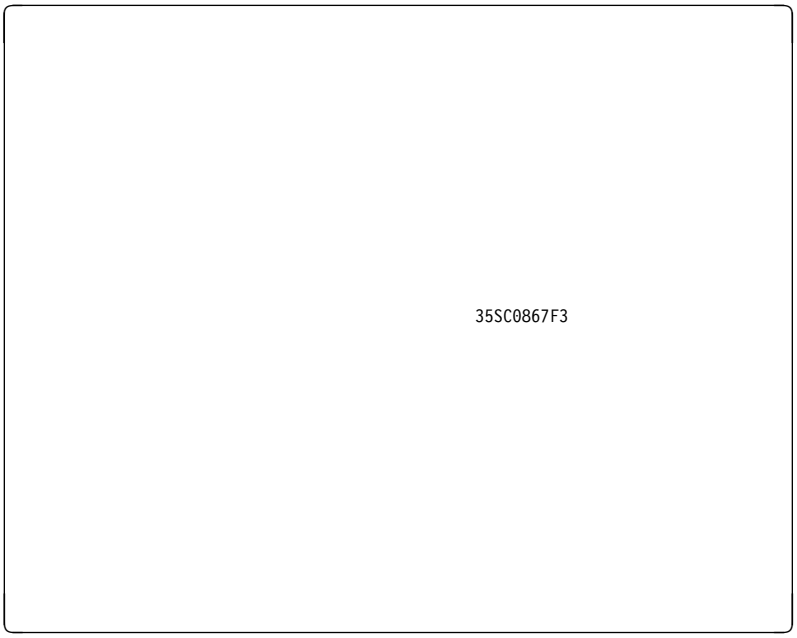


Figure 39. Graphics crossing the data boundary with rough clipping

If you specify rough clipping at the data boundary, all primitives inside the boundary are retained. If a primitive lies across the boundary, the whole primitive, including the part outside the boundary, is usually retained. In general, any primitives that lie completely outside the boundary are not retained.

More complex rules apply to the way some primitives and graphics text characters are clipped. Details of these can be found in the *GDDM Base Application Programming Reference* book.

If you use rough clipping with a data boundary that is larger than the graphics window, it is more likely that the completeness of graphics segments overlapping the graphics window is maintained when segments are manipulated in and around the graphics window. See Figure 39 for an illustration of this.

If clipping is disabled, primitives may extend to the boundary of the graphics field. They may therefore be drawn outside the graphics window if this does not fill the graphics field.

Clipping has different effects on each of the three modes of graphics text. For details of these effects, see the description of the GSCHAR call in the *GDDM Base Application Programming Reference* book.

Drawing graphics outside the segment viewing limits

Graphics primitives that you draw may also appear truncated if they cross the **viewing limits** specified for the segment to which they belong. Segment viewing limits are also specified in world coordinates. They determine how much of the current segment, and any segments it calls, are seen on the display at any one time. Figure 40 and Figure 41 on page 129 illustrate their effect.

You can define the segment viewing limits with this call:

```
CALL GSSVL(0.0,30.0,0.0,50.0); /* Set segment viewing limits */
```

The GDDM default segment viewing limits coincide with the graphics field.

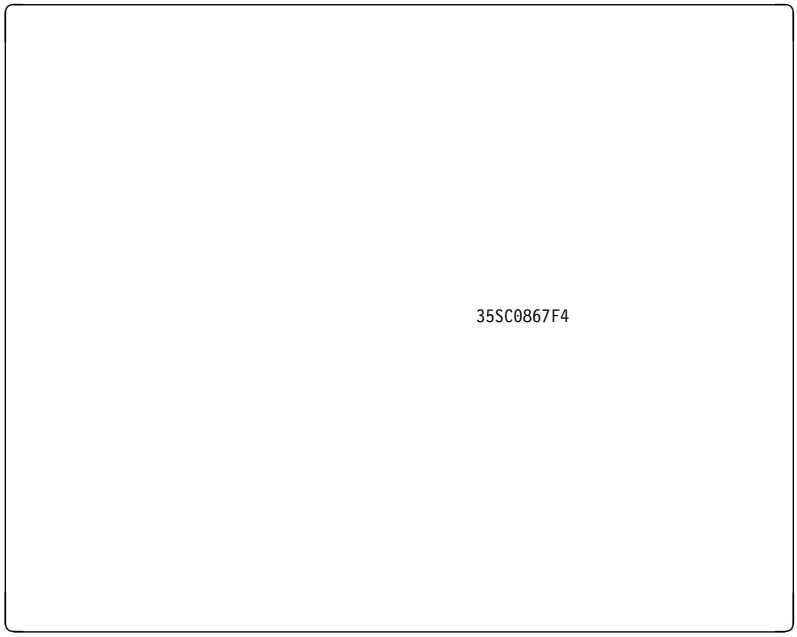


Figure 40. The effect of segment viewing limits on a primitive exceeding them

Whether clipping to the data boundary is enabled or not, primitives are **always** clipped precisely to the specified or defaulted segment viewing limits.

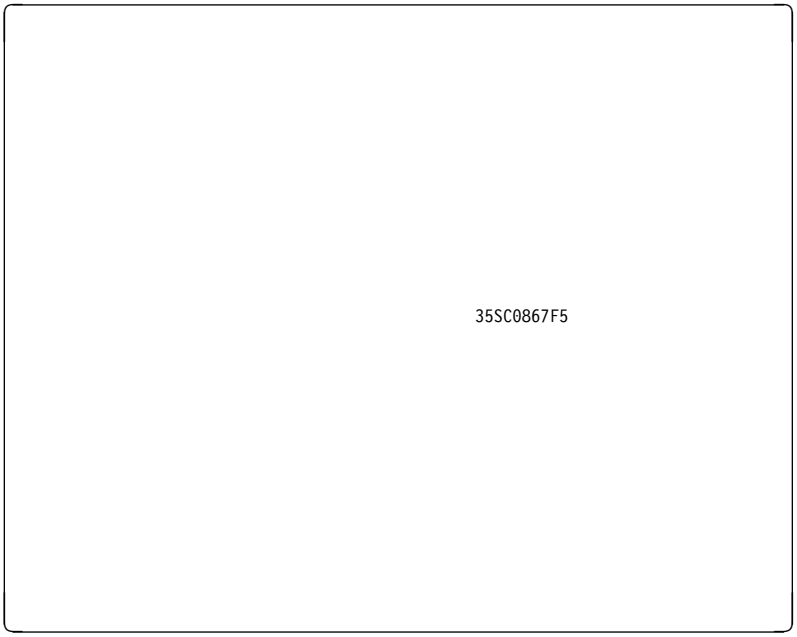


Figure 41. Moving the primitives in a segment within the viewing limits

If segment viewing limits have been set and an object is moved around the screen, the object is only partly or wholly visible when part or all of it lies within the limits. This can be used as a way of panning around a graphics picture that is much larger than the viewing limits of the segment that contains it.

Chapter 8. Error handling and debugging

Chapter 19, “Designing device-independent programs” on page 391 offers some good advice that can help your programs to avoid being in error when used with various devices. However, no matter how carefully you write a program, errors inevitably occur.

This section explains some of the causes of errors in GDDM application programs. It also describes facilities supplied by GDDM to help you eliminate errors from programs and write programs that can cope with errors.

The causes of errors in GDDM application programs

The cause of a GDDM error can be internal or external to the application program.

Internal errors These errors are intrinsic to the program. They occur, for example, if incorrect parameters are passed to API calls or if correct, but unsuitable, parameters produce unexpected output. The common name for such an error is a “bug”.

External errors These occur, if the environment specified by the end user’s external defaults or the input provided by the end user does not match the programs needs.

You should aim to eliminate all internal errors from your programs and GDDM provides a number of debugging aids to help you in this task:

- Error messages
- Error records
- FSQERR, a call that queries the most recent error message and returns an error record to your program for analysis
- An external default option that enables you to trace the flow of GDDM calls in your program.

For advice about using these facilities to debug programs, see “Identifying bugs in your program” on page 133. For a description of GDDM trace facilities, see the *GDDM Diagnosis* book.

External errors present a different problem to application programmers. Handling such errors means anticipating them in your program and taking action that either corrects the error or at least informs the end user that there is a problem.

GDDM provides a number of a calls that you can use to handle external errors:

- Query calls can enable your application to take account of the environment in which it is running. Chapter 19, “Designing device-independent programs” on page 391, shows how you can use calls such as FSQURY, FSQDEV, and ESQEUD to help programs avoid errors.

  If you include an FSQSYS call in an application, it can determine the operating system on which it is being run and then invoke the routines that suit that system.

  Using the FSQERR call, application programs can check for GDDM error messages and records when critical API calls, (such as DSOPEN), or groups of calls have been issued. This call is most often used for debugging purposes,

© Copyright IBM Corp. 1982, 1996

but the program can also test the error record returned and call a suitable error handling routine if necessary.
- The FSEXIT call provides you with a simpler way of performing this task. With this call, you can tell GDDM to pass control to an exit routine in your program whenever an error occurs that is above a specified threshold.

GDDM error messages

When an error occurs, whether internal or external, GDDM usually issues a pair of error messages.

These messages usually appear on the end user's console except in the following circumstances:

- If the application is running on the TSO subsystem and invokes the GDDM print utility. Error messages for this utility are then sent to the system console.
- If the application invokes the Composite Document Print Utility, error messages are printed on extra pages at the end of the document.
- If GDDM-GKS is invoked, GDDM sends error messages to the GKS error file. Detailed information about this can be found in the *GDDM-GKS Programming Guide and Reference* book.
- If the application uses batch processing, all error messages are sent to the batch log when the program is run.

Here is an example of a pair of error messages.

```
ADM0055 E DSUSE, AT X'4E0202FE'

ADM0082 E DEVICE DOES NOT EXIST
```

The first message gives the name of the erroneous call and its address in main storage, and the second describes the error. Each message consists of the error number, a letter indicating the severity of the error, and the error-message text. The severity of the error can be indicated by any one of the following:

Severity letter	Severity code	Meaning
I	0	Information
W	4	Warning
E	8	Error
S	12	Severe error
U	16	Unrecoverable error

A full list and description of the error messages, and suggested programmer/user responses to them, are given in the *GDDM Messages* book.

After issuing the error messages, GDDM returns control to the application program and execution continues with the next statement.

Analysis of the error information returned by GDDM is the basis for most error handling routines in application programs.

Identifying bugs in your program

If your program produces GDDM error messages when you run it or if the output it produces is not what you expect, these are some of the techniques you can use to find out what is going wrong.

Querying the GDDM error record, using FSQERR

In addition to issuing an error message, for each error, GDDM also builds an error record that can be accessed by the program. The error record contains the same sort of information as the error message.

The FSQERR call returns to the program the error record that reflects the most recent error of warning level or above. Informational messages are not counted as errors. Here is the error record layout, and a typical call:

```
DCL 1 ERROR_RECORD,
     2 SEVERITY FIXED BIN(31),      /* Severity code 0|4|8|12|16 */
     2 NUMBER FIXED BIN(31),        /* Error number              */
     2 FUNCTION_NAME CHAR(8),       /* Function name             */
     2 MSG_LENG FIXED BIN(31),      /* Message length            */
     2 MSG_TEXT CHAR(80),           /* Message text              */
     2 FUNCTION_CODE FIXED BIN(31), /* Request Control Parameter */
     2 PARMLIST_PTR POINTER,        /* Address of user's params  */
     2 RET_ADDR POINTER,            /* Return address to program */
     2 ARITH_INSERT1 FIXED BIN(31), /* First message insert      */
     2 ARITH_INSERT2 FIXED BIN(31), /* Second message insert     */
     2 CHAR_INSERT1 CHAR(20),       /* Character message insert  */
     2 CHAR_INSERT2 CHAR(20);       /* Character message insert  */

CALL FSQERR(160,ERROR_RECORD);      /* Return whole error record */
                                    /* for the most recent error */
```

With the FSQERR call, you can specify whether GDDM is to return all or part of the error record. The example returns the **complete** error record. It would be used by an advanced program that wanted to analyze errors, or perhaps to present its error messages in some unusual format. You can also use it to maintain a record of errors occurring with the program on auxiliary storage.

More commonly, you may decide to test whether a particular GDDM call (or group of calls) executed successfully. You need to request (and declare) only that part of the error record in which you are interested.

Alternatively, you can use the standard system return code that GDDM sets in register 15. The top (high order) half of register 15 is set to the severity code of the error and the bottom (low order) half of register 15 contains the error number. The best policy may be to issue FSQERR following calls that return a nonzero return code to register 15.

If you write programs using the C/370 language, you can use a template provided in the ADMTSTRC.H header file to declare the structure for the GDDM error record.

The severity code in the error record is a numeric value of either 4, 8, 12, or 16 corresponding exactly to the codes W, E, S, or U in the error message. If there has been no error of warning level or above (since initialization or a previous

FSQERR call) GDDM returns a severity value of 0. It is good practice to test the severity code field after critical GDDM calls, (such as DSOPEN), or groups of calls and invoke your own error-handling routines as required.

The programming examples in other sections of this book do not test the return codes because it would obscure the main points they are designed to illustrate.

As mentioned above, FSQERR returns the most recent error since initialization or **since the previous FSQERR**. It is therefore not enough, in general, to place an FSQERR after the call in question. You may be given an error record corresponding to a GDDM call made some time before. To ensure that the error record (if any) corresponds to the particular call that you want to verify, you must execute an FSQERR as the most recent GDDM call that occurred before the one you want to test (except in the case of the first GDDM call in the program).

```
DCL 1 ERROR_RECORD,
      2 SEVERITY FIXED BIN(31),
      2 ERROR_NUMBER FIXED BIN(31);

/*******************************/
/* Clear error record (if any) */
/*******************************/
CALL FSQERR(8,ERROR_RECORD);       /* Clear previous error record  */

/******************************/
/* Execute call to be checked */
/******************************/
CALL ASDFMT(7,8,DFMT_ATTRS);       /* Redefine page's alpha fields */

/**********************/
/* Query error        */
/**********************/
CALL FSQERR(8,ERROR_RECORD);       /* See if ASDFMT resulted       */
                                   /* in an error                  */

IF SEVERITY > 4 THEN GOTO ENDRUN;  /* If alpha redefine failed,    */
                                   /* then end run.                */

    Continue normal processing...
```

Using GDDM trace to debug application programs

There are many debugging tools available to programmers enabling them to trace the execution of instructions in their programs. GDDM offers its own trace facilities, so that you can follow the step-by-step execution of GDDM calls in your program.

If you receive a number of GDDM error messages when you run your program, or if the program produces unexpected GDDM output, you can use the GDDM trace facility to examine the GDDM API calls in the program. To produce a file containing trace output from your program, you need to specify some GDDM external defaults before executing your program.

Note: If you write GDDM programs using the REXX language, you can avail of the REXX language trace, the GDDM trace facility, and the GDDM-REXX trace facility, which you can switch on using the GXSET subcommand. For information on this subcommand, see the *GDDM Base Application Programming Reference* book.

The external defaults that initiate a trace for the GDDM calls in a program can be passed to GDDM from any of the following locations:

1. The external-defaults module
2. The user's defaults file
3. The SPIB control block passed on the initialization call SPINIT (if the system programmer interface is used)
4. Any ESSUDS or ESEUDS calls in the application

The most convenient way is to specify them in your own user defaults file before you run the program.

A major advantage of this method of tracing is that you do not have to change your program to use it. Here are some example tracing defaults:

```
ADMMDFT TRCESTR='IF API THEN PARMSF'
```

On the CMS subsystem, this statement in your defaults file produces a print file called ADM00001 ADMTRACE on your A disk.

On the TSO subsystem, the same statement sends the trace output to ddname ADMTRACE.

Printing trace output

You can specify that the parameter values of every GDDM call in your program be output to a trace print file.

On CMS, you can send the trace output directly to the system printer by coding the CMS external default like this:

```
ADMMDFT CMSTRCE=(,)
```

To send trace output directly to a printer on TSO, you can associate the ddname specified on the TSOTRCE external default with a system output stream (SYSOUT).

Specifying when trace is to be invoked

You can confine the GDDM trace to occurrences of an individual call in the program, an ASREAD for example, by coding the default statements like this:

```
ADMMDFT TRCESTR='IF API THEN'
ADMMDFT TRCESTR='   IF (1 GR+4)%%=X''C100000'' THEN PARMSF'
```

The single call is checked for by checking the contents of a register for the hexadecimal value of the call's request control parameter (RCP) code, in this case, X'C100000' for ASREAD. The RCP codes of all GDDM Base calls, and full information about the defaults mechanism are given in the *GDDM Base Application Programming Reference* book.

Or, if you want to trace a call only every nth time that a particular set of conditions occurs, these statements, for example, trace every fourth occurrence of ASREAD:

```
ADMMDFT TRCESTR='IF API THEN                        '
ADMMDFT TRCESTR='   IF (1 GR+4)%%=X''C100000'' THEN '
ADMMDFT TRCESTR='      IF COUNT(4) THEN PARMSF      '
```

Format of the trace output file

Here is a small extract from a trace file, showing the format of the information output for a single ASREAD call:

```
00000415 01 CPNIN  ASREAD ('0C100000'X) - READ
PTRACE    2 FIXED  ---OUTPUT ONLY PARAMETER-----
PTRACE    3 FIXED  ---OUTPUT ONLY PARAMETER-----
PTRACE    4 FIXED  ---OUTPUT ONLY PARAMETER-----
00000544 01 CPNOUT ASREAD ('0C100000'X) - READ
PTRACE    2 FIXED              0
PTRACE    3 FIXED              0
PTRACE    4 FIXED              0
```

Full information about GDDM tracing under the various operating systems, and the format of the trace output file, is given in the *GDDM Diagnosis* book.

Error information returned in a control block

You can tell GDDM to return error information in a control block instead of sending messages to the terminal. You specify your requirement using the GDDM ERRFDBK external default. This can be done by means of a SPINIT call or an ESEUDS call, or in the GDDM defaults module. Detailed information about these calls is given in the *GDDM Base Application Programming Reference* book.

Information returned in register 15

If you are using a programming language that allows you access to registers, you can get error information from register 15. On return from a call to GDDM, the top half of this register contains the error severity code and the bottom half the error number.

Error information for the reentrant and system programmer interfaces

Error information, consisting of an error code and a severity code, is supplied by GDDM in the application anchor block (AAB). Details are given in the *GDDM Base Application Programming Reference* book.

Writing programs that can cope with error conditions

If you are writing a program for use by other people, you may need it to be more robust than one written for just your own purposes. You can include routines in it that can be invoked in different operating environments. However, you also need to cater for occasions when errors occur for which your program has no corrective actions. Your program may continue to function when warning messages are issued for a GDDM call and may even have routines that cope with some errors, but some errors of high severity cannot be corrected by any action in the program. In such situations, it is best to send a message to end users, indicating that there is a problem and suggesting remedial action for them to take. One way to do this is to specify an error exit routine.

Specifying an error exit and threshold, using call FSEXIT

This call specifies a user routine that gains control when an error of specified severity occurs. This is a typical call:

```
CALL FSEXIT(DIAG66,8); /* Give control to routine DIAG66 if any */
                       /* error of severity 8 or higher occurs  */
```

If an application program is using the nonreentrant interface, the named routine is passed just one parameter – the GDDM error record, described in “Querying the GDDM error record, using FSQERR” on page 133. If either the reentrant or the system-programmer interface is used, the routine is passed two parameters. The first of these is the application anchor block, previously passed by the application program to GDDM; the second is the GDDM error record.

Using the default error-exit routine

If you don’t issue an FSEXIT call in your program to specify an error exit explicitly, the default error exit applies. The default error exit becomes current at the moment your program initializes GDDM. GDDM uses the default error threshold specified, usually by the systems-support personnel who customize GDDM, on the ERRTHRS external default. The usual setting for this default is 4. (On the IMS subsystem, it is 8.) This causes the default error exit to be called following all errors of severity 4 or higher (8 or higher on IMS).

The default error exit displays the error message to the end user and returns control to the program. (For this to function under IMS, you need to take special actions. See “Specifying the default error exit under IMS” on page 559.)

You may need to specify the default error exit explicitly, if you want the program to continue using it with a different error threshold. If you want only messages about errors, severe errors and unrecoverable errors to appear on the user's screen, you can specify 0 on the first parameter of FSEXIT and 8 (error) on the second parameter.

To ensure the correct data type for this parameter, the call should be coded in the following way in PL/I:

```
CALL FSEXIT(BINARY(0,31),8);  /* Call default exit to present   */
                              /* error messages if severity     */
                              /* is 8 or more.                  */
```

This call would suppress messages of “warning” level. Only messages of higher severity would be sent to the user console.

When a new program is being tested, it may prove useful to call the default exit after every GDDM call. This, in effect, sends a trace to the display of all the GDDM calls that have been executed. This is the statement needed:

```
CALL FSEXIT(BINARY(0,31),0);  /* Call default exit after every  */
                              /* call to trace the program flow */
```

Language considerations for specifying error exit routines

PL/I applications: The name of the error exit routine must be declared as an external entry, otherwise GDDM is unable to pass the error record as a parameter.

In the above example, the 0 constant used to specify the default error exit is coded such that it can only be FIXED BINARY(31). Alternatively, you can declare a variable of type FIXED BINARY(31), initialize it to 0, and specify it as the first parameter of the FSEXIT call.

It is possible to use an internal procedure as the error exit under very restrictive circumstances:

- If the parameters are not referred to and FSQERR is used to obtain error information.
- If no reference is made to nonlocal variables. Because the routine is called by GDDM rather than the application, the environment is not correctly set for referring to nonlocal variables.

  If you include the statement SIGNAL ERROR in the error-exit routine, you can use PL/I error handling to identify and print the call statement that generated the error.

If you intend referring to the parameters of an external procedure that you are using as an error-exit routine, you should ensure that it contains a procedure statement with OPTIONS(COBOL) specified. With this statement, the parameters are passed in Assembler-language format rather than PL/I format.

This does not apply under CICS. The parameters of the error-exit routine should be declared as FIXED BINARY(31) variables and the real definitions of the parameters should be based on the addresses of these variables.

If the main application program uses the nonreentrant programming interface, it can communicate with the error-exit routine by means of external variables In a reentrant program, the Application Anchor Block (AAB) that passes from the program to GDDM is also passed to the error-exit routine. The application extension to the AAB can be used for communication.

FORTRAN applications: There is no restriction on the use of subroutines as error-exit routines in FORTRAN. The error-exit parameters must be declared as dimensioned INTEGER*4 variables. EQUIVALENCE statements can be used to extract data such as error-message text.

COBOL applications: You can specify only the default error exit in COBOL programs and use the FSEXIT call to specify a threshold for invoking it.

If an error exit other than the default is required, you need to give the required error-exit routine the same name as the default error-exit routine and link edit it with the application. The name of the default error exit routine is different on each of the supported subsystems. You can select one from the following list.

Subsystem	*Error-exit name*
CICS	ADMASXC
IMS	ADMASXI
TSO	ADMASXT
CMS	ADMASXV
VSE Batch	ADMASXD

Example of an error exit routine, using FSEXIT

Here is an example of an error exit routine:

```
DCL DERROR EXTERNAL ENTRY;
CALL FSEXIT(DERROR,8);

/*        .         */

DERROR: PROC(ERROR_RECORD) OPTIONS(COBOL);
DCL DCODE FIXED BIN (31) EXTERNAL;/* Communicate with program   */
DCL 1 ERROR_RECORD,               /* GDDM error record.         */
      2 SEVERITY FIXED BIN (31),  /* Severity (range 0-16).     */
      2 NUMBER   FIXED BIN (31),  /* Error message number.      */
      2 FUNCTION CHAR(8),         /* GDDM function giving error.*/
      2 MSGLEN   FIXED BIN (31),  /* Length of message text.    */
      2 MSGTEXT  CHAR(80),        /* Message text.              */
      2 RCP      FIXED BIN (31),  /* GDDM RCP.                  */
      2 PLISTPTR FIXED BIN (31),  /* Parameter list pointer.    */
      2 RETADDR  FIXED BIN (31),  /* Return address.            */
      2 AI1      FIXED BIN (31),  /* Message insert 1.          */
      2 AI2      FIXED BIN (31),  /* Message insert 2.          */
      2 CI1      CHAR(20),        /* Character message insert 1.*/
      2 CI2      CHAR(20);        /* Character message insert 2.*/
IF FUNCTION = 'DSOPEN'            /* DSOPEN has failed because  */
 & NUMBER =  97 THEN              /* there is not a plotter.    */
  DCODE = 4;                      /*                            */
ELSE IF FUNCTION = 'GSLOAD'       /* GSLOAD has failed with an  */
 & NUMBER = 303 THEN              /* unrecognized file format.  */
  DCODE = 8;                      /*                            */
END DERROR;                       /******************************/
```

Example of an error exit routine, without using FSEXIT

Instead of declaring the error routine to be an external entry, you may choose to execute an FSQERR call to obtain the error record:

```
CALL FSEXIT(EERROR,8);             /* Specify error exit.       */

/*        .         */
/*        .         */
/*        .         */
                                   /*                           */
EERROR: PROC(DUMMY);               /* Trap GDDM error.          */
DCL DUMMY CHAR(*);              /* Not used for internal routine*/
DCL DCODE FIXED BIN (31) EXTERNAL; /* Communicate with program. */
DCL 1 ERROR_RECORD,                /* GDDM error record.        */
     2 SEVERITY FIXED BIN (31),    /* Severity (range 0-16).    */

/*         .       */
/*         .       */
/*         .       */

     2 CI2      CHAR(20);          /* Character message insert 2*/

CALL FSQERR(160,ERROR_RECORD);     /* Get error record structure*/
IF FUNCTION = 'DSOPEN' THEN        /* DSOPEN for plotter has    */
  DCODE = 4;                       /* failed.                   */
ELSE IF FUNCTION = 'GSLOAD' THEN   /* GSLOAD has failed.        */
  DCODE = 8;
END EERROR;
```

Bypassing GDDM's parameter checking to improve the speed of applications

Whenever a program makes a GDDM call, all the parameters attached to the call are checked for validity on entry to GDDM. The cost of doing this can be quite substantial. GDDM allows you to omit this initial parameter checking and so save processor resource. The way to do this is to define an external control section (CSECT) named ADMUFO to be link-edited with the application program and the GDDM interface module. The contents of the CSECT do not have to be defined.

Note: Parameter checking is still done by the various subcomponents of GDDM, so error messages may still be issued with ADMUFO active.

The ADMUFO CSECT can be defined using standard assembler language facilities, thus:

```
ADMUFO CSECT
       END
```

Alternatively, you can use high-level language constructs, where such are available. In PL/I, you can generate the CSECT with a declaration of the form:

```
DECLARE ADMUFO STATIC EXTERNAL;
```

The important point to make about ADMUFO is that it you should only include it in programs that are free of error. Otherwise, ADMUFO masks any errors caused by invalid parameters on GDDM calls. If you bypass GDDM's parameter checking too

soon in the development of your program, it can take a lot longer to identify problems that are usually diagnosed quickly.

ADMUFO gives most benefit where parameter checking constitutes a significant proportion of the total time taken to process the calls made. Programs that use GDDM procedural alphanumeric calls are likely to show up to 15% reduction in the processor usage attributable to GDDM. The saving for programs that display graphics on the 3279 is more likely to be up to 3%. The saving on 3270-PC/G and 3270-PC/GX work stations is likely to be up to 20%.

On MVS/XA, the fast bypass is not invoked if the application-addressing mode requires a change (that is, if the application call is in 24-bit mode). In this instance, it is necessary to generate a parameter-list copy, with the top bytes of each address word cleared.

This generally means that the User Fast Option functions on MVS/XA only for 31-bit mode applications. An application program executing in AMODE(24) will execute, but with the fast path disabled.

Note: If GDDM is installed using the OS LOAD macro, applications that access GDDM's system programmer interface by dynamic load, cannot use the user fast option.

Part 2 Advanced GDDM functions

© Copyright IBM Corp. 1982, 1996

Chapter 9. Manipulating graphics segments

A segment is a group of graphics primitives and attributes that can be handled as an entity, separate from other segments and primitives. This section describes the calls that create, delete, and copy segments, and those that change a segment's appearance by moving, rotating, rescaling, or shearing it. Chapter 10, "Storing and retrieving graphics pictures" on page 173 explains how to store segments on external storage.

Segments can also call other segments. This means that you can organize your graphics segments into a structure or hierarchy. Just like well-structured programs, well-structured data has the advantages of increased clarity and ease of maintenance. You do not have to divide a picture into segments. The complete picture can be a single segment, or primitives can be drawn outside segments altogether. A segmentation scheme should be the most convenient and efficient implementation of the functions that the end user requires.

Segments have major uses in interactive graphics applications. Such applications generally enable the end user to manipulate parts of pictures. For instance, a program for designing the external appearance of a house might have the house outline, the doors, and the windows as separate segments. It would then be relatively simple to enable the end user to position, rescale, and manipulate each of these items independently.

Chapter 11, "Writing interactive graphics applications" on page 197 describes calls and techniques for making a graphics application interactive.

Creating segments, using GSSEG

Segments are opened by executing a GSSEG call. They can be named:

```
CALL GSSEG(24);      /* Define named segment with identifier 24 */
```

or unnamed:

```
CALL GSSEG(0);       /* Define an unnamed segment (in other     */
                     /* words, one with a zero identifier)      */
```

Unnamed segments are not recommended if you are going to use GSSAVE, and GSLOAD. See Chapter 10, "Storing and retrieving graphics pictures" on page 173.

By default, created segments are appended by GDDM to a *drawing chain*, containing all the segments that you create in the order that you create them. Only the segment data held in the drawing chain appears after a complete regeneration of the screen.

Primitives belong to the currently open segment. This can be closed with a GSSCLS call:

```
CALL GSSCLS;         /* Close current segment                   */
```

Issuing this call does not delete the graphics primitives enclosed in the segment, it means that you do not intend to add any further primitives to them.

© Copyright IBM Corp. 1982, 1996

A segment must be closed before another one can be opened. It must also be closed before respecifying any object that is above it in the graphics hierarchy (as described in Chapter 7, “Hierarchy of GDDM concepts” on page 107). For example:

```
CALL GSVIEW(0.0,1.0,0.0,0.5);        /* Define first viewport   */
CALL GSSEG(1);                       /* Open a graphics segment */
CALL GSMOVE(20.0,30.0);/*Start drawing picture in first viewport*/
 .
 .
 .
CALL GSSCLS; /* Must close segment before defining new viewport */

CALL GSVIEW(0.0,1.0,0.5,1.0);        /* Define second viewport  */

CALL GSSEG(2);                       /* Open a graphics segment */
CALL GSCOL(3);                       /* Start drawing picture   */
       .                             /* in second viewport      */
       .                             /* and so on...            */
       .                             /*                         */
```

You cannot reopen a named segment, once closed. But you can create as many unnamed segments as you may choose, as explained in “Unnamed segments” on page 149.

You can still draw primitives when there is no segment open. The effects are described in “Primitives outside segments” on page 170.

Within a page you may have as many named or unnamed segments as you choose, but each named segment must have a different nonzero identifier. For example, it would be an error to issue `CALL GSSEG(24)` if the current page already has a segment with that identifier.

Graphics primitive attributes are associated with the segment that is current when they are defined. If you issue `CALL GSCOL(2)` to change the current color to red, all later primitives in the segment (such as lines and arcs) are drawn in red. If you then close the segment and open a new one, all the graphics attributes (including color) are usually reset to the defaults. A called segment, however, does not assume the default attributes on being opened. Instead, it inherits the current attributes. These remain current until changed within the called segment. See “Calling segments from other segments, using GSCALL” on page 165.

Once a primitive has been drawn with an **explicitly** defined attribute, as in the above GSCOL call, it cannot be altered. You cannot normally change, a line's color from red to blue, unless the line was drawn with the default color attribute. This can affect already-drawn pictures, as described in “Changing default attribute values” on page 47.

It is important to remember that segments are collections of primitives, not areas of the screen. You could, for instance, create one segment comprised of some primitives in each corner of the screen, and another comprised of some other primitives in the middle. And you can overlap primitives from different segments. Figure 42 on page 147 comprises only three segments. For identification, all the primitives of each one are the same color.

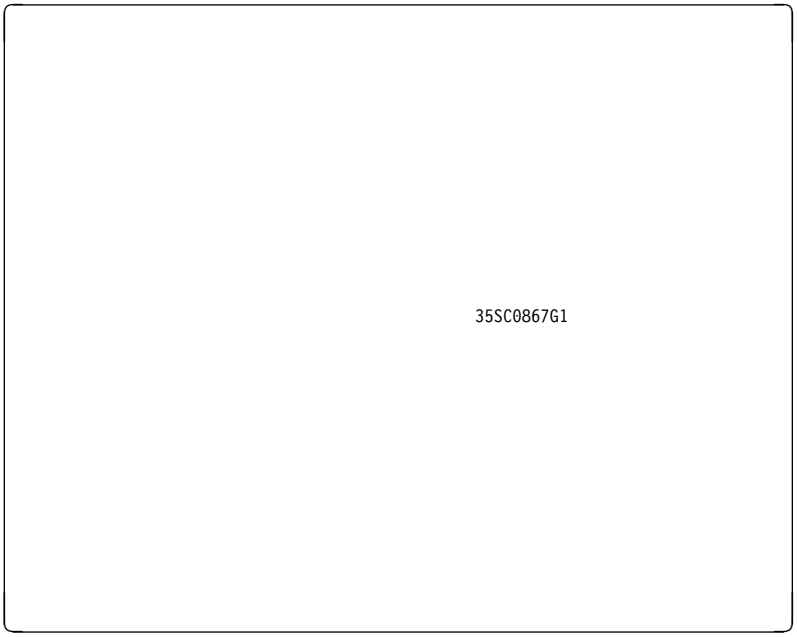


Figure 42. Primitives in the same segments share the same attributes.

Deleting segments, using GSSDEL

You can delete only a named segment–that is, one with a nonzero identifier. This is the call:

```
CALL GSSDEL(15);      /* Delete segment 15 and all its contents */
```

A later ASREAD, GSREAD, MSREAD, or FSFRCE causes the picture to appear without those primitives that belonged to segment 15.

Having deleted segment 15, you may open a new segment with identifier 15.

To delete all the segments in the graphics field, you can issue this call:

```
CALL GSCLR;                        /* Clear the graphics field */
```

Segment attributes

Every segment has a set of attributes. These should not be confused with the graphics attributes of the contained primitives. Segment attributes are properties of the group of primitives as a whole. They determine such things as whether the segment can be transformed (that is, moved, scaled, or rotated) and whether it is visible or invisible.

A segment acquires the attributes that are current when it is opened. Opening a segment creates a default graphics field if none exists already. A default set of segment attributes becomes current initially, when a graphics field is defined or created by default.

You set the current segment attributes one at a time, using the GSSATI call. For instance, this call sets the current value of the visibility attribute to invisible:

```
CALL GSSATI(2,0);  /*Make subsequently opened segments invisible*/
```

The first parameter specifies the type of attribute that is being set, and the second the value it is being set to. Valid types and values are:

1 Detectability. This determines whether a segment can be selected by a pick graphics input device (described in Chapter 11, “Writing interactive graphics applications” on page 197). The second parameter means:

- 0 Segment cannot be picked. This is the default.
- 1 Segment can be picked.

2 Visibility. The second parameter means:

- 0 Segment is invisible.
- 1 Segment is visible. This is the default. Only a visible segment can be selected by a pick device.

3 Highlighting. The second parameter means:

- 0 Segment is not highlighted. This is the default.
- 1 Segment is highlighted by being made white.

4 Transformability. The second parameter enables you, for your own reference, to mark segments as transformable or nontransformable. It does not actually affect the transformability of segments – all segments can be transformed.

- 1 Segment is marked as nontransformable. Segment is not to be moved, scaled, rotated, or sheared. This is the default.
- 2 Segment is marked as transformable. The segment can be moved, scaled, rotated, or sheared.

5 This type has no effect in the current release. It should always be set to either 0 or 1.

6 Chained or nonchained attribute. This determines whether a segment is included in the drawing chain. By default, segments are added to the drawing chain when they are created. They are subsequently drawn in the order that they appear on the drawing chain, unless you change their priority (see “Drawing chain and segment priority” on page 164). An example of the use of the chaining attribute is to exclude called segments from the drawing chain until they are called. See “Calling segments from other segments, using GSCALL” on page 165 for more details. The second parameter means:

- 0 The segment is excluded from the drawing chain. It can be included in the drawing chain only when called by another segment.
- 1 The segment is included in the drawing chain. This is the default.

You can change the attributes of the current segment or any other segment in the current graphics field by a GSSATS call. A typical call is:

```
CALL GSSATS(7,2,0) /* Make segment 7 invisible */
```

The first parameter is the segment identifier. The second and third parameters can have the same values, with the same meanings, as the two parameters of GSSATI.

Unnamed segments

If you do not need to manipulate groups of primitives, but nonetheless want to avoid the disadvantages of primitives outside segments, you can put the primitives into unnamed segments.

You create an unnamed segment by specifying an identifier of zero in the GSSEG call. You can use as many unnamed segments as you need. You can mix unnamed segments with named ones and, if you choose, with primitives outside segments. This, for example, is a valid sequence:

```
CALL GSSEG(1);       /* Segment 1               */
/*   .   */
/*   .   */
CALL GSSCLS;

CALL GSSEG(0);       /* An unnamed segment      */
/*   .   */
/*   .   */
CALL GSSCLS;

CALL GSSEG(0);       /* Another unnamed segment */
/*   .   */
/*   .   */
CALL GSSCLS;

CALL GSSEG(2);       /* Segment 2               */
/*   .   */
/*   .   */
CALL GSSCLS;

CALL GSSEG(0);       /* Another unnamed segment */
/*   .   */
/*   .   */
CALL GSSCLS;
```

Unnamed segments cannot be manipulated like named ones. They cannot be deleted, transformed, copied, or included. They cannot be detected by a pick input device. Neither their priorities nor their origins can be changed.

In many ways, using unnamed segments is like drawing primitives outside segments. The main difference is that unnamed segments are retained by GDDM, and they are redisplayed when the screen is regenerated. Another is that they can be highlighted or made invisible with a GSSATI call before the GSSEG(0) call (although the attributes cannot be changed with a GSSATS call after the segment has been created).

Transforming segments, using GSSAGA or GSSTFM

Segments can be transformed in four ways, as shown in Figure 43 on page 150:

Displaced	Moved to another x,y location
Scaled	Made larger or smaller in the x or y direction, or in both
Rotated	Moved about a fixed point on the x y plane
Sheared	Sloped to one side.

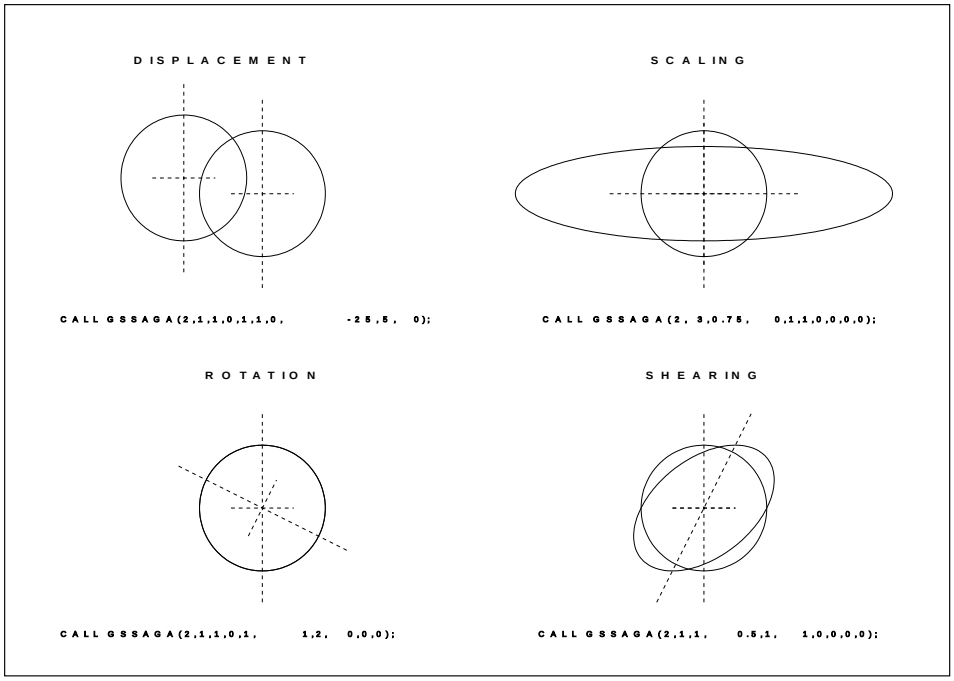


Figure 43. The four segment transformations

You can specify one or more transformations in a GSSAGA call. Or, instead, you may use a transformation matrix and a GSSTFM call (see “Transforming segments using call GSSTFM” on page 154). A transformation can also be applied when you call a segment (see “Calling segments from other segments, using GSCALL” on page 165).

Typical GSSAGA calls for single transformations are shown in Figure 43. In each case, the original, untransformed segment is shown in red. The segment origin (that is, the position x=0,y=0 in world coordinates when the segment was drawn), is at the center of the circle.

You must always specify all the parameters of GSSAGA, including null specifications for those transformations you do not want to be performed. Here is a call with a complete set of null specifications:

```
   /* Segment-id Scaling  Shearing  Rotation  Displacement Type */
CALL GSSAGA( 7,    1,1,     0,1,      1,0,        0,0,       1);
```

The parameters of GSSAGA have these meanings:

- The first is the identifier of the segment to be transformed.
- The next two are the *scaling* parameters. They are multipliers to be applied to the x and y coordinates, respectively. The segment is expanded or contracted, leaving its origin unchanged. You can specify a negative scaling parameter, to reflect primitives about the other axis.
- The next two are the *shearing* parameters. GDDM shears the positive y axis of the segment to pass through the point defined by these parameters. The illustration in Figure 44 on page 151 shows the effect of shearing by dx and dy. The shearing is carried out about the segment origin, the position of which remains unchanged.

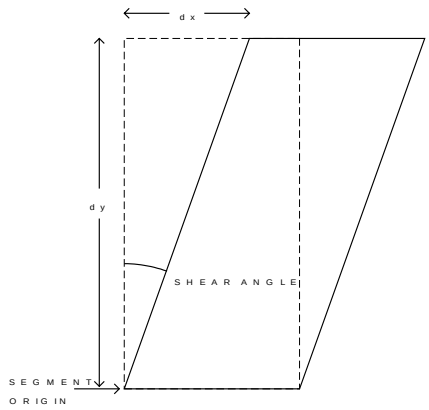


Figure 44. Shearing

If dx and dy have the same sign, the shear is to the right (clockwise), as shown. If they have different signs, the shear is to the left.

If you know the angle of shear (call it S) in degrees or radians, and you have uniform window coordinates, you can specify a dx of sin(S) and a dy of cos(S).

- The next two are the *rotation parameters*. GDDM rotates the positive x axis of the segment to pass through the point defined by these parameters. The illustration in Figure 45 on page 152 shows the effect of rotating by dx and dy. The rotation is carried out about the segment origin, the position of which remains unchanged.

  Notice that dx and dy define the position of the **x axis**. This means that a positive dx and dy define a counterclockwise rotation.

  Negative values of dx and dy are valid as well as positive, allowing rotations in the full range from 0 through 360 degrees. Some example rotations are:

  1,0 No rotation
  0,1 90 degrees counterclockwise
  1,1 45 degrees counterclockwise
  0,–1 90 degrees clockwise
  –1,0 180 degrees (clockwise or counterclockwise – same result)

  If you know the angle of rotation (call it R) and you have uniform window coordinates, you can specify a dx of cos(R) and a dy of sin(R).

- The next two are the *displacement* parameters in world-coordinate units. They specify values that are to be added to, respectively, the x and y coordinates of all the primitives in the segment. So the segment is moved, but the position of its origin remains unchanged.

- The last parameter specifies the type of transformation:

  0 **New**. The specified transformations are applied to the original primitives; any previous GSSAGA or GSSTFM calls for this segment being nullified.

  1 **Additive**. Any previous transformations for this segment are applied first, and then the ones specified in this call are applied to the result.

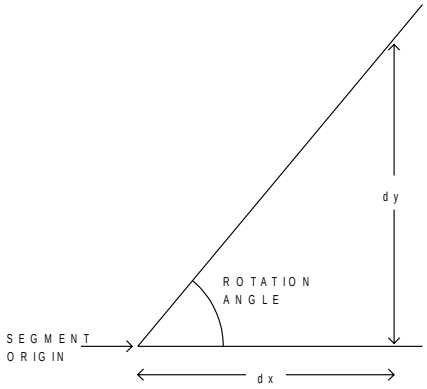


Figure 45. Rotation

2 **Preemptive**. The transformations specified in this call are applied first, and then any previously specified ones are applied to the result.

The transformations in a single GSSAGA call are applied in the order in which the parameters are coded: scaling, shearing, rotation, displacement.

The order in which transformations are performed becomes important when displacement or scaling is combined with rotation or shearing. To understand why, imagine a picture of a standing human figure of normal proportions. If it were first scaled by 2 in the y direction, and then rotated by 90 degrees, the result would be a picture of a very tall person lying down. If, instead, it were first rotated by 90 degrees and then scaled by 2 in the y direction, it would become a picture of a very fat person lying down.

To back out of all previous transformations for a segment, and display it as originally drawn, you can execute a transformation call such as GSSAGA or GSSTFM with all transformations and type set to null:

```
   /* Segment-id Scaling  Shearing  Rotation  Displacement Type */
CALL GSSAGA( 7,    1,1,     0,1,      1,0,        0,0,       0);
```

Notice that GSSAGA does not move the segment origin. All the transformations in Figure 43 on page 150 leave the segment origin at the point marked by the red cross. You can change the position of a segment's origin with a GSSORG call (see “Moving the origin of a segment, using GSSORG” on page 158).

How and when transformations take effect

GDDM applies transformations when a page is sent to the display printer or plotter not when the transformation calls are issued.

Each transformable segment has an object called a *transform* associated with it. This is a matrix that records the net result of all the transformation calls for the segment. Initially, when the segment is opened, the transform is set to identity, giving no transformations. Each later transformation call updates it.

On output, the transform is applied to the segment's graphics primitives to create the display. Instead of the segment as originally drawn, the display contains the transformed version. GDDM's record of a segment's graphics primitives is never altered. While the segment exists, GDDM retains this record.

A segment's transform, together with its graphics primitives, graphics attributes, and segment attributes, are held by GDDM as Graphics Data Format (GDF) orders (see “Saving pictures in Graphics Data Format, using call GSSAVE” on page 175).

Transforming text, markers, and graphics images

GDDM applies the transform to all vectors (straight lines and arcs) in the segment. For instance, to perform a displacement of 10,–20, GDDM adds 10 world-coordinate units to the x values of the start and end points of all lines in the segment, and subtracts 20 units from their y values.

Graphics text is transformed by modifying its attributes – its character box size, character angle, and so on. This means that the range of possible transformations is limited. The limitations are the same as when setting the attributes with calls such as GSCB and GSCA. These are explained in Chapter 4, “Creating graphics-text output in your application” on page 57.

Briefly, for mode-3 text, all the transformations can be fully implemented; for mode-2, only the position of each character can be transformed; and for mode-1, only the position of the start of each string. Considering, for example, just displacement and rotation, this means:

- A mode-1 text string can be displaced but not rotated;
- Individual mode-2 characters within a string can be displaced, and the base line of the string can be rotated about the segment origin;
- Individual mode-3 characters can be displaced and individually rotated about the segment origin.

Images created by the GSIMG or GSIMGS call behave like single characters of mode-2 text: they can be displaced, but not transformed in any other way.

Markers behave like single characters of mode-3 text if they are vector symbols, or of mode-2 text if they are image symbols.

Moving a segment and its origin using call GSSPOS

This call moves a segment, in the same way as a displacement transformation using the GSSAGA call. The difference is that GSSPOS moves the segment origin, whereas GSSAGA leaves it unchanged. GSSPOS looks like this:

```
    /* Segment-id  New position */
CALL GSSPOS(3,      35.0,-15.0);
```

The first parameter is the identifier of the segment to be moved, and the other two are the x and y coordinates of its new position. GDDM moves the segment so that its origin is in this position. The segment must have the transformable attribute.

Suppose the segment contains a line that was drawn by executing these calls:

```
CALL GSMOVE(-5.0,-5.0);
CALL GSLINE(10.0,10.0);
```

After the GSSPOS call, the line extends from (30,–20) to (45,–5) as shown in Figure 46 on page 154.

Note that the segment origin is the one that was in use when the segment was drawn – **not** the window origin at the time of the GSSPOS call. The difference is important if more than one GSSPOS is issued for a segment. For example, if the program that issued the previous GSSPOS example now executes this call:

```
CALL GSSPOS(3,-20.0,20.0);
```

the segment origin moves from (35,–15) to (–20,–20). The line then extends from (–25,15) to (–10,30).

You can query the results of GSSPOS calls with a GSQPOS call. However, the GSQORG call (see “Moving the origin of a segment, using GSSORG” on page 158) is recommended in preference. It gives the same result, but is more versatile because it can be used to query both transformable and nontransformable segments.

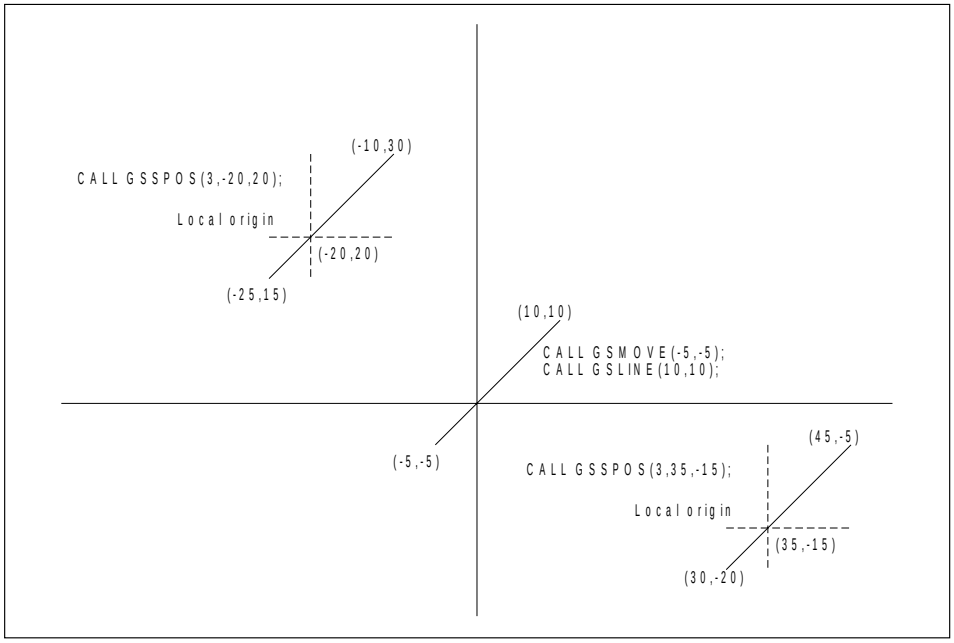


Figure 46. Effects of GSSPOS calls

Transforming segments using call GSSTFM

If you have a mathematical background or are an experienced graphics programmer, you may prefer to manipulate the transformation matrix directly. The display position of every point (x,y) in a segment is given by the matrix expression:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

You can set the values in the matrix by the GSSTFM call:

```
DCL MATRIX(6) FLOAT DEC;

/* Set Values of Matrix in Row-Major Order */

MATRIX(1) = ...;                                    A
MATRIX(2) = ...;                                    B
MATRIX(3) = ...;                                    C
MATRIX(4) = ...;                                    D
MATRIX(5) = ...;                                    E
MATRIX(6) = ...;                                    F

    /* Segment-id Elements  Values  Type */
CALL GSSTFM(3,        6,    MATRIX,   0);
```

The parameters of GSSTFM have the following meanings:

- The first one is the identifier of the segment whose transform is being defined.
- The second is the number of elements being supplied.
- The third is the array in which the elements of the matrix are specified. The order is row-major (**A** through **F**).
- The fourth is a type parameter with the following possible values and meanings:

  `0` New. The specified matrix replaces the existing transform.

  `1` Additive. The specified matrix is to premultiply the existing transform, with the effect that the specified transform is applied after the existing one.

  `2` Preemptive. The specified matrix is to postmultiply the existing transform, with the effect that the specified transform is applied before the existing one.

  These type values have exactly similar effects to the type values in the GSSAGA call: see page 151.

The default values for the third parameter correspond to the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix resets all the transformations for the specified segment. If the value of the second parameter of GSSTFM is less than nine, omitted elements are taken from the default matrix. The last three values, if specified, must always be the same as their defaults, so in practice you need never specify more than six values.

If you specify zero elements in the second parameter, GDDM assumes the identity matrix. This enables the transformations to be set to null, simply by letting the segment be displayed as originally drawn:

```
DCL DUMMY(1) FLOAT DEC;
CALL GSSTFM(3,0,DUMMY,0);     /*Reset transform for segment 3 to null*/
```

Notice that the last parameter specifies a new (0-type) transformation. Either of the other types (1 or 2) would leave the segment's transform unchanged, because they would either premultiply or postmultiply it by the identity matrix.

GSSTFM, GSSAGA, and GSSPOS all modify the segment's transform, and they can be mixed freely.

Querying transforms

There are two calls for querying the transform of a segment, corresponding to two of the transform-setting calls. This one corresponds to GSSAGA:

```
DCL (SCAX,SCAY,SHEX,SHEY,ROTX,ROTY,DISX,DISY) FLOAT DEC(6);

    /* Segment-id  Scaling    Shearing   Rotation  Displacement */
CALL GSQAGA( 7,   SCAX,SCAY, SHEX,SHEY, ROTX,ROTY,  DISX,DISY);
```

The first parameter identifies the segment whose transform is being queried. The other eight are variables in which GDDM returns values that would have to be specified in a new-type GSSAGA call to create the transform. Note, though, that GSSAGA has one more parameter than GSQAGA, namely the last one, which specifies the type of transformation required. The values returned by GSQAGA are not necessarily the same as any that may have been specified in earlier GSSAGA calls, but they give the same results.

This is the query call that corresponds with GSSTFM:

```
DCL MATRIX(6) FLOAT DEC;

    /*Segment-id  Elements  Values  */
CALL GSQTFM(3,        6,    MATRIX);
```

The first parameter is again the segment identifier. The second specifies how many elements of the transformation matrix are being requested, and the third is an array in which GDDM returns them.

The elements are returned in row-major order, the same as is used in the GSSTFM call. A maximum of nine elements can be requested. The seventh, eighth, and ninth are always 0, 0, and 1.

Examples of transformations

To help you understand the GSSAGA call, Figure 47 on page 157 illustrates the effects of several transformations.

The diagram labeled START shows the starting position for each of the seven transformation sequences that follow.

The first transformation, diagram 1, is a simple displacement. The square moves 30 units to the right and 30 units upward.

Diagram 2 shows the effect of following this displacement with a rotation. The square does not rotate about its center; it rotates about the segment origin which is still in its default position of (0,0). The rotation therefore causes the square to change position.

In diagram 3 the segment origin is set to the center of the square before the rotation is performed. The square therefore maintains its position when it is rotated.

Diagram 4 shows the effect of scaling by 2 in the x direction. Because the scaling is about the segment origin at (0,0), the left-hand bottom corner of the box has its x coordinate increased from 10 to 20. So, in addition to becoming twice its original width, the box also changes position.

Diagram 5 shows how you can scale the box without changing its position. You set the segment origin to the center of the box before performing the scaling operation.

The first two transformations in diagram 6 displace the box by (30,30), then rotate the box about its center. The angle of rotation is that given by dx=10, dy=4. After the rotation, a scaling is applied in the x-direction. This distorts the original shape, giving the same effect as a shear operation.

If you want to double the width of the box without the shearing effect, you must perform the scaling **before** you rotate it. Either apply the scaling GSSAGA first, or (as shown in diagram 7) set the last parameter of the scaling GSSAGA to 2. This ensures that the scaling is done before all the other transformations. Note that the segment origin has to be reset to the original center of the box before the prescaling is performed.

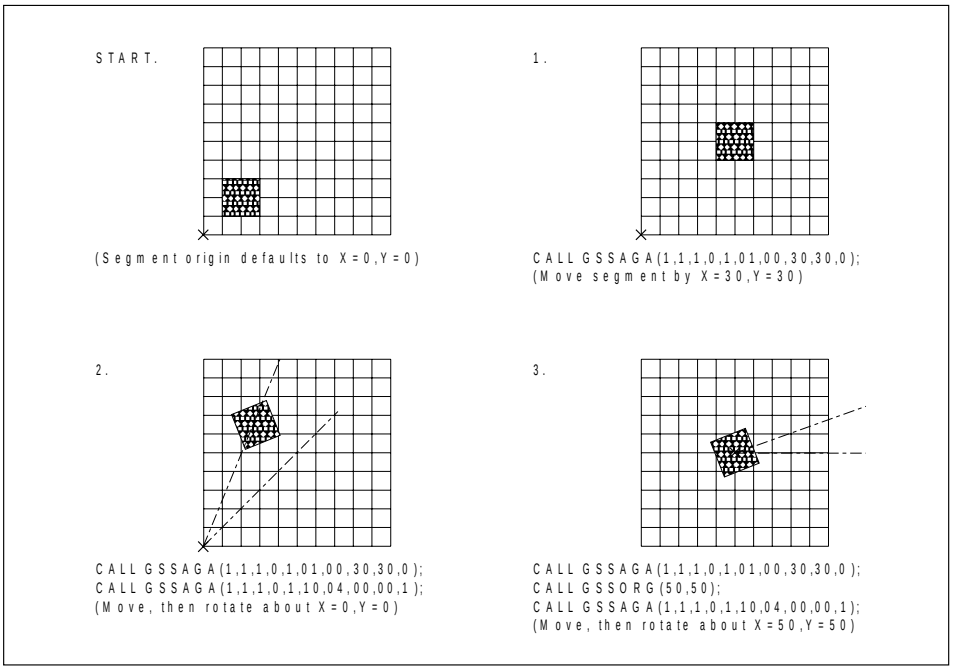


Figure 47 (Part 1 of 2). Results of example transformations

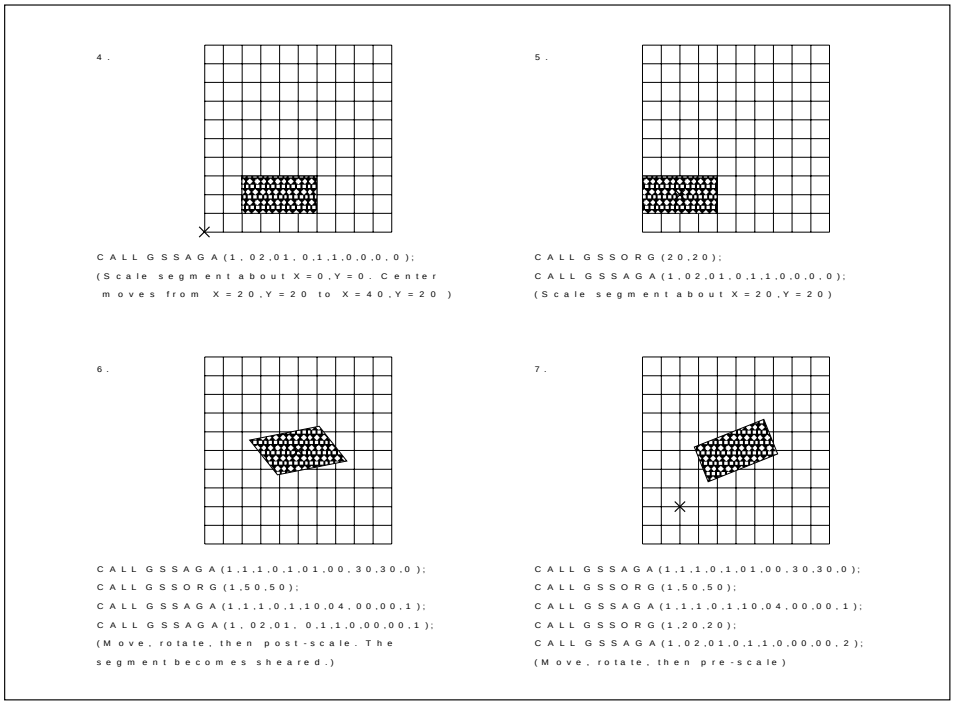


Figure 47 (Part 2 of 2). Results of example transformations

Moving the origin of a segment, using GSSORG

The local origin of a segment is the origin of the world-coordinate system in which the segment was originally drawn. Transformation with the GSSAGA (or GSSTFM) call leaves the local origin unchanged. You can move the local origin with a GSSORG call:

```
     /*Segment-id   New position for local origin */
CALL GSSORG(5,              20.0,40.0);
```

The first parameter is the identifier of the segment, and the other two are the x and y coordinates of the new position for its local origin. The effects of GSSORG are illustrated in Figure 48. This shows the origin of the world-coordinate system, and the local origin of the segment before and after the GSSORG.

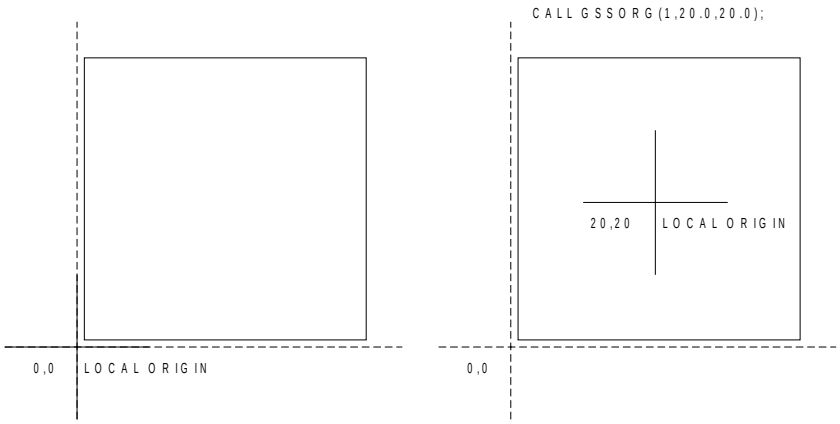


Figure 48. The GSSORG call

The GSSORG call **does not move the segment**. On its own it produces no visible change to the picture. Its effects are noticeable only if you subsequently specify scaling, rotation, or shearing transformations, or use the segment as a locator echo, as explained in Chapter 11, “Writing interactive graphics applications” on page 197. The transformations take place about the new segment origin. Displacements are unaffected by changing the segment origin.

You can query the location of a segment's origin with a GSQORG call:

```
DCL (X,Y) FLOAT DEC(6);
    /* Segment-id   Local origin */
CALL GSQORG(5,           X,Y    );
```

The segment is identified in the first parameter. In the second and third parameters, GDDM returns the position of its origin in world coordinates.

Transforming primitives within a segment, using GSSCT

In the same way that you can apply a transform to a segment, the call GSSCT sets a current transform that is applied to all the primitives that follow the call. The current transform is a primitive attribute, but is described in this section because the call can only be issued within a currently open segment, and is carried out in relation to the origin of the segment. Here is a typical call:

```
       /* Scaling  Shearing  Rotation  Displacement Type */
CALL GSSCT( 1,1,     0,1,      1,0,        0,0,       0 );
```

The parameters are similar to those for the call GSSAGA, covered in “Transforming segments, using GSSAGA or GSSTFM” on page 149. See that section for an illustration of the effect of transforms, and the meaning of the parameters. The last parameter specifies the type of transformation:

0 **New**. The specified transformations are applied to the original primitives; any previous GSSCT call for this segment is ignored.

1 **Additive**. Any previous current transforms for this segment are applied first, and then the ones specified in this call are applied to the result.

2 **Preemptive**. The transformations specified in this call are applied first, and then any previously specified current transforms are applied to the result.

The transformations in a single GSSCT call are applied in the order in which the parameters are coded: scaling, shearing, rotation, displacement.

If you want to save the old current transform that was in existence before a new GSSCT call, you can do so by initially ensuring that attribute mode is set to preserve attributes, by either using the GSAM call, or allowing GSAM to default if it has not been previously set. The old transform is stored when you call GSSCT, and can subsequently be restored using GSPOP. GSAM and GSPOP are covered in “Storing and restoring graphics-attribute values, using GSAM and GSPOP” on page 46.

Copying segments, using GSSCPY

You can copy any closed segment with a GSSCPY call:

```
CALL GSSCPY(3); /* Copy segment 3 */
```

The local origin of the copy is placed at the current position. If the copied segment is transformable, its current transform is applied before copying. The primitives in the copied segment become part of the currently open segment, if there is one; otherwise, they become primitives outside segments. The current position and graphics attributes are not affected by copying.

In effect, a call to GSSCPY is like a call to a subroutine that reexecutes the graphics calls that created the segment being copied. It also has a number of other effects:

1. Before copying, the current transform is applied (if the segment is transformable), and the primitives are displaced by an amount equal to the coordinates of the current position.
2. After copying, the current position, and the current graphics attributes, are restored to what they were before the call.

The effects of the following calls are shown in Figure 49 on page 161:

```
/*****************************************************************/
/*                    SEGMENT 1                                  */
/*****************************************************************/

CALL GSSEG(1);                /* Open segment,current pos.= 0,0*/
CALL GSAREA(0);
CALL GSLINE(0.0,20.0);        /* Draw a square (in the default */
CALL GSLINE(20.0,20.0);       /* color, green).                */
CALL GSLINE(20.0,0.0);        /*                               */
CALL GSLINE(0.0,0.0);         /*                               */
CALL GSENDA;
CALL GSCOL(7);                /* White ...                     */
CALL GSMARK(10.0,10.0);       /* ... marker at center of square*/

CALL GSSCLS;                  /* Close the segment.            */
```

```
/*************************************************************/
/*                    SEGMENT 2                              */
/*************************************************************/

CALL GSSEG(2);                 /* Open segment,current pos.= 0,0*/
CALL GSCOL(2);                 /* Set current color.            */

/*                 Rotate square before copying               */
    /* Segment-id  Scale     Shear    Rotate   Displace   Type  */
CALL GSSAGA(1,    1.0,1.0,  0.0,1.0, 1.0,1.0,  0.0,0.0,     0);

CALL GSCHAR(0.0,60.0,21,'GSLINE BEFORE GSSCPY ');
CALL GSLINE(70.0,60.0);        /* Draw first line               */

CALL GSSCPY(1);                /* Copy the rotated square       */

CALL GSLINE(70.0,25.0);        /* Draw second line              */
CALL GSCHAP(19,'GSLINE AFTER GSSCPY');

CALL GSSCLS;                   /* Close the segment             */

/*************************************************************/
/*           Undo rotation of original square                */
/*************************************************************/

/*     Segment-id  Scale     Shear     Rotate    Displace  Type */
CALL GSSAGA( 1,   1.0,1.0,  0.0,1.0,   1.0,0.0,  0.0,0.0,   0);

CALL ASREAD(ATTYPE,ATTVAL,ATTCNT);  /* Send to terminal         */
```

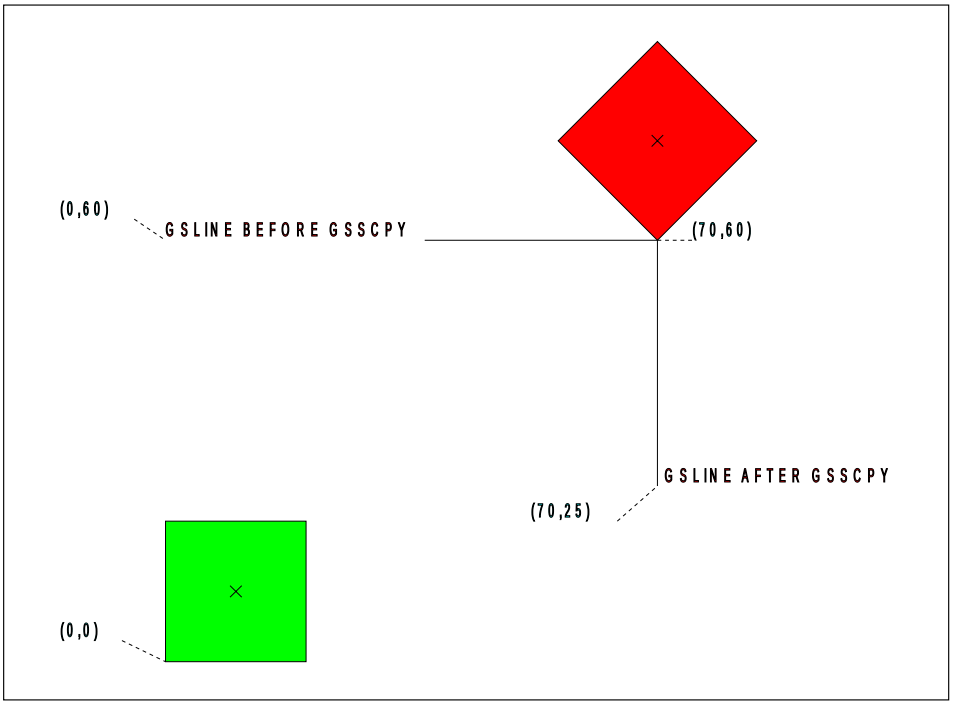


Figure 49. Copying segments, using GSSCPY

The square changes color during copying. This is because no color was set before it was drawn in segment 1, whereas in segment 2, the color was set to 2 (red) before the GSSCPY call. The copied graphics primitives inherit this explicit graphics attribute. If the color had been set explicitly in segment 1, the copy would not inherit the value set in segment 2–it would be in the same color as the original.

Notice that the line drawn after copying starts where the one drawn before copying ends. This illustrates that the current position is not affected by copying. Notice, too, that both lines are red–the line drawn after the copy is not affected by the color setting for the marker in segment 1.

Including segments, using GSSINC

Including a segment with a GSSINC call creates a copy, as does GSSCPY:

```
CALL GSSINC(5);  /* Include segment 5 */
```

but a GSSINC call behaves **exactly** like a call to a subroutine that reexecutes the segment-creation statements. GSSINC therefore differs from GSSCPY in these ways:

- The segment is not transformed before it is included.
- The segment is not moved to the current position: the copy is, in effect, drawn on top of the original (assuming the default mix mode of overpaint).
- The current position changes to the one that was in effect when the included segment was closed.
- The current attributes change to those that were in effect when the included segment was closed.

A major use of GSSINC is to specify prepackaged graphics attributes. For instance:

```
DCL BLU_SOL_SOL FIXED BIN(31) INIT(10);
DCL RED_SOL_SOL FIXED BIN(31) INIT(11);
 ...
DCL BLU_DOT_SOL FIXED BIN(31) INIT(20);
 ...
DCL BLU_DOT_DOT FIXED BIN(31) INIT(30);

 ...
CALL GSSEG(BLU_SOL_SOL);
CALL GSCOL(1);            /* Blue                              */
CALL GSLT(7);             /* Solid line type                   */
CALL GSPAT(0);            /* Solid shading pattern             */
CALL GSSCLS;

CALL GSSEG(RED_SOL_SOL);
CALL GSCOL(1);            /* Red                               */
CALL GSLT(7);             /* Solid line type                   */
CALL GSPAT(0);            /* Solid shading pattern             */
CALL GSSCLS;
```

```
 ...
CALL GSSEG(BLU_DOT_SOL);
CALL GSCOL(1);            /* Blue                              */
CALL GSLT(2);             /* Dotted line type                  */
CALL GSPAT(0);            /* Solid shading pattern             */
CALL GSSCLS;
 ...
CALL GSSEG(BLU_DOT_DOT);
CALL GSCOL(1);            /* Blue                              */
CALL GSLT(2);             /* Dotted line type                  */
CALL GSPAT(7);            /* Dotted shading pattern            */
CALL GSSCLS;
 ...

                          /* Create segment using one of the     */
                          /* standard set of graphics attributes */

CALL GSSEG(100);
CALL GSSINC(BLU_DOT_SOL); /* Include standard attributes    */
 ...                      /* Create the graphics primitives */
CALL GSSCLS;
```

Combining segments, using GSSINC and GSSDEL

You can combine two or more segments into a single one by opening a new segment, copying them into it using GSSINC, and then deleting the originals using GSSDEL. You can use this technique to combine all segments on a page into a single segment. Then you can transform and otherwise manipulate the picture as a whole.

Unnamed segments cannot be copied, so you must use a nonzero identifier for any segment that might be combined. You should normally preserve segment priority by copying the segments in priority order. (You can use the GSQPRI call to query all existing segments in priority order.)

Here is an example that combines all segments on a page.

```
DECLARE (SEG,NEXT_SEG) FIXED BINARY(31);

CALL GSSEG(100);                      /* 100 is reserved id   */
                                      /* for combined segment.*/
CALL GSQPRI(0,SEG,-1);                /* Find lowest-priority */
                                      /* segment.             */
DO WHILE SEG¬=0;
  CALL GSSINC(SEG);                   /* Include the segment. */
  CALL GSQPRI(SEG,NEXT_SEG,1);        /* Find next-highest    */
                                      /* priority segment.    */
  CALL GSSDEL(SEG);                   /* Delete copied        */
                                      /* segment.             */
  SEG = NEXT_SEG;
END;

CALL GSSCLS;                          /* Close segment 100    */
```

If you want the segments to have existing transformations applied when they are combined, you should use GSSCPY in place of GSSINC.

Drawing chain and segment priority

As mentioned earlier in this section, segments are normally added to the drawing chain when they are created, and subsequently drawn in the order that they appear on the drawing chain. Later segments are said to have higher priority than earlier ones. If you visualize segments as being drawn in layers, with one segment per layer, each new one overlays all the existing ones. Where primitives from different segments occupy the same location, the later one obscures the earlier (assuming the default mix mode of overpaint).

Graphics primitives within a segment follow the same rule: later primitives are drawn on top of earlier ones. When a segment is copied or included, its primitives are drawn on top of any existing primitives, and any later primitives are drawn on top of the copied or included ones.

You can change the priorities of existing segments in the drawing chain with the GSSPRI call:

```
    /* Segment-id  Ref-seg-id   Order  */
CALL GSSPRI(3,         7,         1); /* Put seg 3 after seg 7  */
CALL GSSPRI(9,         2,        -1); /* Put seg 9 before seg 2 */
```

The first parameter specifies the segment whose priority is to be changed. The second specifies another segment called the reference segment. The third parameter must be either 1, meaning the first segment is to become the next higher in priority to the reference segment, or –1, meaning the first segment is to become the next lower in priority to the reference segment.

In addition to altering the drawing order, GSSPRI can affect which primitives are detected by a pick input device (see Chapter 11, “Writing interactive graphics applications” on page 197).

You cannot change the priorities of graphics primitives within a segment, nor of primitives outside segments, nor of segment 0.

One use of the GSSPRI call is in three-dimensional applications, when it is used to ensure that hidden surfaces are not visible or detectable. Another is in drawing layered pictures such as microchip layouts.

You can query segment priorities with the GSQPRI call:

```
DCL NEXT_SEG FIXED BIN(31);

   /* Ref-seg-id     Seg-id    Order */
CALL GSQPRI(3,      NEXT_SEG,   1); /* Which seg follows seg 3? */
```

In the second parameter, GDDM returns the segment next to the one specified in the first parameter – its successor if the last parameter is 1, or its predecessor if this is –1. If the last parameter is 1 and the specified segment is the latest one, GDDM returns 0 in the second parameter. And GDDM similarly returns 0 if the last parameter is –1 and the specified segment is the earliest one.

You cannot query the position of segment 0. A value of 0 in the first parameter has a special meaning, which depends on the value of the last parameter. If this is 1, GDDM returns the identifier of the latest segment, or if it is –1, of the earliest segment.

Querying the order of all segments, using GSQPRI

You can use the GSQPRI call to query all segments existing on the current page in priority order:

```
DECLARE SEG(100) FIXED BIN(31);   /* Store up to 100 seg-ids    */

CALL GSQPRI(0,SEG(1),-1);         /* Find segment with lowest   */
I = 1;                            /* priority                   */
DO WHILE (SEG(I)¬=0) & (I<=99);
  CALL GSQPRI(SEG(I),SEG(I+1),1); /* Query next segment         */
  I = I+1;                        /* identifier                 */
END;
```

Calling segments from other segments, using GSCALL

You can call a segment from a segment, and apply a transform to the called segment, with a GSCALL call. This is a typical call:

```
    /* Seg-id Flag Scaling  Shearing  Rotation  Displacement Type */
CALL GSCALL(2,  0,   1,1,     0,1,      1,0,        0,0,       1);
```

The parameters are identical to those for GSSAGA, except for an extra parameter, the flag. For the current release of GDDM this should always be set to 0. You can only issue a GSCALL from a segment, and the transform applies to the called segment only. When control returns from the called segment to the calling segment, the transform that was in operation before the GSCALL applies.

The concept of GSCALL is, like GSSCPY and GSSINC, similar to calling a subroutine. However, with GSCALL the calling segment contains only a call order at the point of invocation. Contrast this with GSSCPY and GSSINC, where the drawing orders of the copied or included segment are actually repeated in the segment that contains the copy or include call.

The following example program calls various segments to produce the building plan in Figure 51 on page 168.

```
BLDPROG: PROC OPTIONS(MAIN);
DCL(TYPE,MOD,COUNT) FIXED BIN(31);
CALL FSINIT;
CALL GSUWIN(0.0,100.0,0.0,100.0);

CALL GSSEG(1);   /* * * * * *  Open segment 1, the building,    */
CALL GSLW(2);               /* the top segment in the hierarchy */
CALL GSCOL(5);              /*                                  */
CALL GSLINE(0.0,100.0);     /* Draw outline of building         */
CALL GSLINE(100.0,100.0);   /* .                                */
CALL GSLINE(100.0,0.0);     /* .                                */
CALL GSLINE(0.0,0.0);       /* .                                */
CALL GSCALL(2,0, 1,1, 0,1, 1,0, 0,80, 2 );  /* Call offices     */ A
CALL GSCALL(2,0, 1,1, 0,1, 1,0, 20,80, 2 ); /* .                */ A
CALL GSCALL(2,0, 1,1, 0,1, 1,0, 40,80, 2 ); /* .                */ A
CALL GSCALL(2,0, 1,1, 0,1, 1,0, 60,80, 2 ); /* .                */ A
CALL GSCALL(2,0, 1,1, 0,1, 1,0, 80,80, 2 ); /* .                */ A
CALL GSCALL(3,0, 1,1, 0,1, 1,0, 0,0, 2 );   /* Call meeting-    */ B
CALL GSCALL(3,0, -1,1, 0,1, 1,0, 100,0, 2 );/*           rooms   */ B
CALL GSSCLS;     /* * * * * * * * * * * * * *  Close segment 1  */

CALL GSSATI(6,0);/* Leave following segments off drawing chain  */ C

CALL GSSEG(2);   /* * * * * * * * * * * * * *  Open segment 2   */
CALL GSLINE(0.0,20.0);                      /* Draw outline     */
CALL GSLINE(20.0,20.0);                     /* of office...     */
CALL GSLINE(20.0,0.0);                      /*                  */
CALL GSLINE(16.0,2.0);                      /*                  */
CALL GSMOVE(15.0,0.0);                      /*                  */
CALL GSLINE(0.0,0.0);                       /* ...containing a  */
CALL GSCALL(4,0, 1,1, 0,1, 1,0, 7,6, 2 );   /* table (segment 4)*/ D
CALL GSCALL(5,0, 1,1, 0,1, 1,0, 9,15, 2);   /* & chair (segt 5) */ D
CALL GSSCLS;     /* * * * * * * * * * * * * *  Close segment 2  */

CALL GSSEG(3);   /* * * * * * * * * * * * * *  Open segment 3   */
CALL GSLINE(0.0,70.0);                      /* and draw a       */
CALL GSLINE(40.0,70.0);                     /* meeting room     */
CALL GSMOVE(41.0,68.0);                     /* .                */
CALL GSLINE(45.0,70.0);                     /* .                */
CALL GSLINE(45.0,0.0);                      /* containing a     */
CALL GSLINE(0.0,0.0);                       /* table            */
CALL GSCALL(4,0, 4,2, 0,1, 0,1, 29,19, 2);  /* (segment 4)      */ E
CALL GSCALL(5,0, 1,1, 0,1, 0,1, 16,28,  2); /* and four chairs  */ F
CALL GSCALL(5,0, 1,1, 0,1, 0,1, 16,40,  2); /* (segment 5)      */ F
CALL GSCALL(5,0, 1,1, 0,1, 0,-1, 30,31, 2); /* .                */ F
CALL GSCALL(5,0, 1,1, 0,1, 0,-1, 30,43, 2); /* .                */ F
CALL GSSCLS;     /* * * * * * * * * * * * * *  Close segment 3  */
```

Figure 50 (Part 1 of 2). Example program using called segments

```
CALL GSSEG(4);   /* * * * * * * * * * * * * *  Open segment 4,  */ G
CALL GSLINE(0.0,6.0);                       /* and draw a table */
CALL GSLINE(8.0,6.0);                       /*                  */
CALL GSLINE(8.0,0.0);                       /*                  */
CALL GSLINE(0.0,0.0);                       /*                  */
CALL GSSCLS;     /* * * * * * * * * * * * *  Close segment 4  */

CALL GSSEG(5);   /* * * * * * * * * * * * *  Open segment 5,  */ H
CALL GSARC(1.5,0.0,-180);                   /* and draw a chair */
CALL GSLINE(0.0,0.0);                       /*                  */
CALL GSSCLS;     /* * * * * * * * * * * * *  Close segment 5  */

CALL ASREAD(TYPE,MOD,COUNT);                /* Send to screen   */
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END BLDPROG;
```

Figure 50 (Part 2 of 2). Example program using called segments

You can set up a structure for your segment data using GSCALL. For example, your application could produce a building plan like that in Figure 51 on page 168, containing standard-size offices and meeting rooms, that themselves contain several standard-size desks, tables, and chairs. You need to define each of these items only once in your application. That is, one office segment, one meeting room segment, one desk segment, one table segment, and one chair segment. You can then GSCALL each of these segments as many times as you like, and position them wherever you want them in your plan by applying a transform to each segment when you call it.

The table and chair segments are created at **G**. and **H** respectively, and result in segments with their local origins as shown in Figure 52 on page 169.

These are then positioned in an office segment by GSCALL statements that apply transformations containing only displacements, as the table and chair are already correctly oriented for use here. The offices, complete with table and chair, are then displaced by GSCALL statements at **A** into five different positions within the building.

For the meeting-room segment definition, the table segment is called at **E**, where scaling and rotation are applied to the table. It is scaled by a factor of 4 in the x direction, and 2 in the y direction, and is rotated counterclockwise by 90 degrees. Displacements are chosen to center the scaled, rotated table in the meeting room. Remember that scaling, shearing, rotation, and displacement are applied in that order (the order in which they are specified).

The chair segment is called four times at **F**. Two of the chairs are rotated clockwise, and two counterclockwise, and displaced to positions either side of the table.

Finally, in the building segment, the meeting room segment is called twice at **B**. The first instance, for the left-hand room, applies the identity transformation – no scaling, shearing, rotation, or displacement take place. This is because the

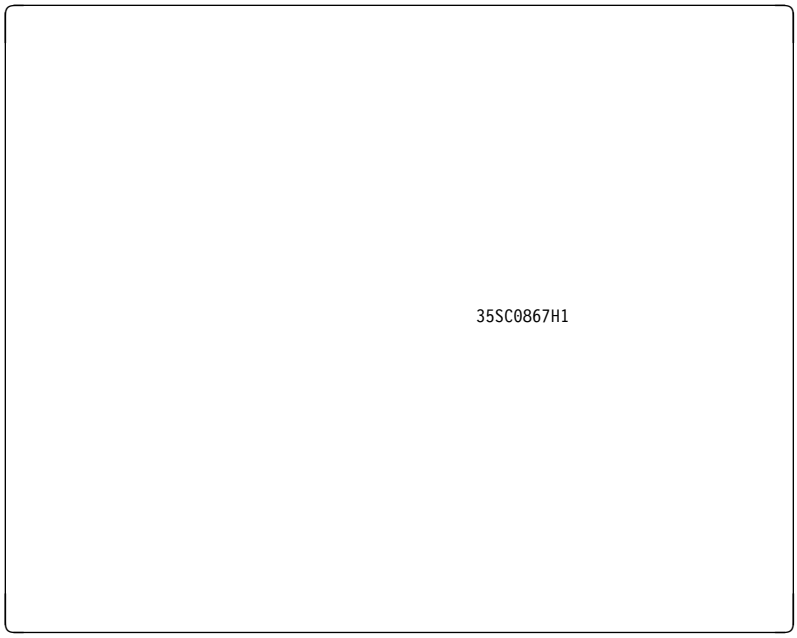


Figure 51. Building plan produced by called segments

meeting room segment, with its bottom-left-hand corner at its origin (0,0), is already in the required position.

The right-hand meeting room, however, is reflected about the y-axis by applying an x scaling factor of –1, and then requires displacement to position it on the right-hand side of the building. Scaling, shearing, and rotation transforms are carried out with reference to the origin of the **calling** segment (the one containing the GSCALL) **not** with reference to the origin of the called segment. In the example program this is not important, because all the segments have their origins in the same position of (0,0).

Following these calls in the building segment definition, the GSSATI call at **C** is needed to exclude the subsequently created segments from the drawing chain, so that they only appear when called, and not “in-line.”

A particular point to note is that, in the example, all the GSCALL statements a type parameter value of 2. This ensures that the transformations are performed in order from the bottom of the segment structure to the top. In the example, this means that the furniture is arranged in the meeting room segment, before the meeting room is reflected and displaced to the right-hand side of the building. You have already seen the effect of GSSAGA call sequences and type parameters in Figure 47 on page 157. The type parameter on GSCALL controls the way that the associated transformation combines with a preceding transformation in exactly the same way.

If the segments called by your program reside on a segment library of standard items of furniture, the program must first load the required segments before they can be called. (See Chapter 10, “Storing and retrieving graphics pictures” on page 173). This may mean you do not know the position of the origin of a loaded and called segment.

Figure 52. Table and chair segments with origin

If you try to produce a loop of called and calling segments, GDDM detects it and issues an error message when you run the program.

Graphics attribute handling with called segments

A called segment does not assume the default graphics attributes normally assumed by a newly created segment. Instead, it inherits the attributes that are current when it is called. By default, if you change an attribute (for example, line type, color, character box, current transform) to a new value within a called segment, GDDM automatically pushes the corresponding old primitive attribute onto a last-in first-out stack. When control returns to the calling segment, GDDM carries out an implicit GSPOP to recover the old attribute values of any attributes that were changed in the called segment. GDDM ensures, therefore, that no matter what changes are made to the attribute values in the called segment, the attribute values in the calling segment are preserved. If you wish, you can suppress GDDM's automatic preservation of attribute values. This section of example code illustrates the use of GSAM to control the preservation of attributes:

```
keep=5.
CALL GSSEG(1);                          /* Open segment 1    */
CALL GSCOL(2);                          /* Set color to red  */
CALL GSCALL(2,0, 1,1, 0,1, 1,0, 10,10, 0);/* Call segment 2    */
                                        /* and preserve      */
                                        /* calling attributes*/
CALL GSMOVE(10.0,20.0);
CALL GSLINE(15.0,22.5);                 /* Red line drawn    */
CALL GSCOL(2);                          /* Set color to red  */ A
CALL GSCALL(3,0, 1,1, 0,1, 1,0, 50,10, 0);/* Call segment 3    */
                                        /* and don't preserve*/
                                        /* calling attributes*/
CALL GSMOVE(30.0,15.0);
CALL GSLINE(17.0,14.0);                 /* Green line drawn  */
CALL GSSCLS;                            /* Close segment 1   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
CALL GSSATI(6,0);         /* Leave following segments off      */
                          /* the drawing chain until called.   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
CALL GSSEG(2);                          /* Open segment 2    */
CALL GSAM(0);                           /* Save attributes   */ B
                                        /* (this is default) */
CALL GSCOL(4);                          /* Set color to green*/
CALL GSMOVE(5.0,7.0);
CALL GSLINE(10.0,17.0);                 /* Green line drawn  */
CALL GSSCLS;                            /* Close segment 2   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
CALL GSSEG(3);                          /* Open segment 3    */
CALL GSAM(1);                           /* Don't save        */
                                        /*        attributes */
CALL GSCOL(4);                          /* Set color to green*/
CALL GSMOVE(15.0,5.0);
CALL GSLINE(70.0,23.0);                 /* Green line drawn  */
CALL GSSCLS;                            /* Close segment 3   */
```

The GSCOL call at **A** is not really needed, but is there for emphasis. Similarly, the GSAM call at **B** is only needed if a previous GSAM has suppressed the preservation of attributes.

Graphics not in named segments

Primitives do not have to be contained in named segments. They can be outside segments altogether, or in an unnamed segment–one created with a zero identifier.

Primitives outside segments

You can draw primitives before opening any segment on the current page, or between closing one segment and opening another, or after closing the last segment.

GDDM does not retain a record of primitives outside segments after the current page has been sent to the screen. This enables it to free the main storage occupied in recording the primitives. The saving is less with cell-based terminals, such as the 3279, than with other types. This is because GDDM must still retain a

record of the contents of the PS stores, even though it discards the record of the primitives.

The main uses of primitives outside segments, therefore, are in situations where you want to store graphics temporarily at the terminal, and not on the GDDM page in the host.

The main disadvantage of primitives outside segments is that they are deleted from the display if it is completely regenerated. The screen is not regenerated completely by GDDM at every ASREAD, GSREAD, MSREAD, or FSFRCE. When it is partially updated, the primitives outside segments are retained on the display.

A complete regeneration is required if, for instance, a segment is deleted after it has been displayed. The *GDDM Base Application Programming Reference* book describes, for different devices, the various circumstances under which the contents of the screen are completely redrawn.

Any manipulation of a segment generally results in a complete regeneration. For this reason, it is inadvisable to mix segments and primitives outside segments on the same GDDM page. If you are going to create even one segment, then you probably need to put all the graphics into segments. Conversely, if you want to exploit the advantages of primitives outside segments, you probably have to avoid creating any segments at all.

Other disadvantages of primitives outside segments are that they cannot be picked by graphics input devices (see Chapter 11, “Writing interactive graphics applications” on page 197); they are not saved by the FSSAVE call; they are not printed by a GSCOPY or FSCOPY call (see “Using a printer as an alternate device” on page 412); and they cannot be plotted.

Initially, when a page is explicitly or implicitly created, the primitives outside segments have the same attributes as the defaults that apply when a segment is opened. You can execute calls outside segments to change the current attributes. Changing the default values for attributes causes any primitives outside segments to be discarded.

If you do open a segment, GDDM retains a record of the graphics attributes previously in force, and restores them when the segment is closed. If, for instance, you create a page, set the current color to red, then open and close a segment, red again becomes the current color for any further primitives drawn outside a segment.

If you draw a primitive before opening any segments on the current page, then any items in the physical hierarchy that are not yet defined are defaulted.

Chapter 10. Storing and retrieving graphics pictures

This section tells you how to write GDDM programs that can:

- Store complete graphics pictures or individual segments on external storage
- Retrieve graphics pictures from external storage and load them onto the GDDM page
- Modify the drawing orders of loaded graphics pictures to change their appearance

The stored-graphics formats that GDDM supports

There are three different types of saved-graphics data that application programs can load onto the GDDM page.

GDF Graphics Data Format (GDF) orders are used internally by GDDM to draw graphics and they can be saved in files in one of the following ways:

- By a graphics application program issuing the GSSAVE call
- By end users of a graphics application using the User Control facility to save graphics output
- By users of the GDDM Interactive Chart Utility (ICU) saving their chart output

With the GSLOAD call, your applications can load pictures saved in GDF files onto the GDDM page and display them.

A full list of GDF orders is given in the *GDDM Base Application Programming Reference* book. The attributes of GDF files on each of the supported subsystems are listed in the appendixes of this book.

CGM Computer Graphics Metafiles (CGM) store graphics orders that are slightly different from GDF orders. Many PC graphics applications can export (save) their output in CGM format. With the CGSAVE call GDDM applications can save output in this format.

You can load pictures saved in CGM format onto the GDDM page using the CGLOAD call. GDDM converts the CGM orders into GDF orders before loading the picture onto the current GDDM page. Because there are several different types of CGM formats, applications need to specify the name of a **conversion profile** to help GDDM with this conversion. Table 2 on page 174 shows the conversion profiles supplied with GDDM to enable conversion between the ADMGDF format and the CGM format produced by each of the listed graphical applications:

© Copyright IBM Corp. 1982, 1996

Table 2. GDDM-supplied conversion profiles for conversion of data between ADMGDF and CGM formats.

Conversion profile	Graphics application
ADM	General Purpose
ADMCD	Corel Draw
ADMFP2	Freelance Plus V2
ADMFP3	Freelance Plus V3
ADMHG	Harvard Graphics
ADMMD	Micrografx Designer

Note: In GDDM Version 2 Release 3 the names of these conversion profiles began with the characters CGM.

The parts of the conversion process that are specific to applications are defined in a ***CGM Conversion Profile***. GDDM supplies a profile tailored to each of the applications listed above, although, depending on usage, further tailoring may be necessary.

In a number of instances, the general-purpose profile will produce acceptable output without further tailoring (especially with enhancements added for GDDM V3.2). You may want to use this profile as the basis for your own tailored conversion profiles for applications used by your enterprise.

To convert pictures from other applications you may need to write your own conversion profiles.

GDDM supports only binary encoded CGM data. A list of supported CGM orders is given in the *GDDM Base Application Programming Reference* book. The attributes of CGM files on each of the supported subsystems are listed in the appendixes of this book.

PIF Picture Interchange Format (PIF) files are used to store pictures on PC DOS and 3270-PC/G and /GX workstations and can be interchanged between a workstation and the host computer. The drawing orders in PIF files are similar to GDF orders.

PIF files are created at the workstation, sent to the host, converted to a GDF file and retrieved by an application program on the host computer issuing a GSLOAD call. The reverse route can also be followed: a host-created GDF file can be converted to a PIF file and sent to the workstation.

The attributes of PIF files on the CMS and TSO subsystems are listed in the relevant appendixes of this book.

GDDM guarantees picture fidelity for converted Base PIF files, as defined in *IBM 3270 PC/G or /GX Supplementary Reference Information for PIF*.

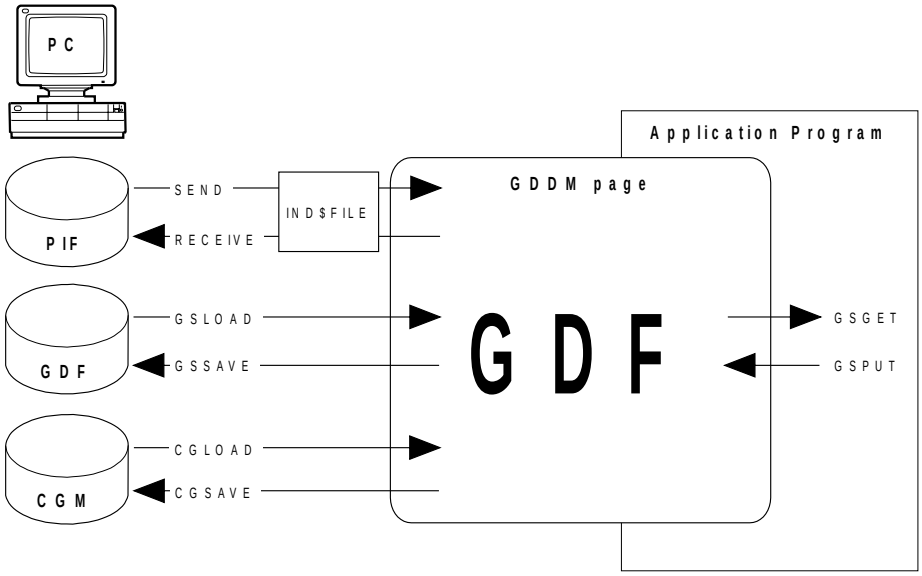


Figure 53. How application programs move saved graphics between external storage and GDDM.

Saving pictures in Graphics Data Format, using call GSSAVE

With the GSSAVE call, your program can save individual segments or all the graphics that it has placed on the current GDDM page.

There must be no open segment when the GSSAVE call is executed.

The GSSAVE call converts graphics pictures into Graphics Data Format (GDF) graphics orders and saves them in a file. GDF orders have similar meanings to GDDM graphics calls and in many cases, there is a one-to-one mapping between GDF orders and GDDM graphics calls. Here are some examples:

```
  CALL STATEMENT          FUNCTION                 GDF ORDER

CALL GSLINE(3,7);      Draw line                 X'810400030007'
CALL GSCOL(2);         Set current color         X'0A02'
CALL GSCHAP(3,'ABC');  Write character string    X'8303C1C2C3'
CALL GSSCLS;           Close segment             X'7100'
```

A full list of GDF orders is given in the *GDDM Base Application Programming Reference* book.

On the CMS subsystem, the GSSAVE call saves the GDF orders in a file with a default filetype of ‘ADMGDF’. On TSO, ‘ADMGDF’ is the default data set name of the data set in which the GDF orders are saved.

In addition to the graphics orders for primitives and their attributes, the ADMGDF file contains orders for segment attributes, identifiers, and transforms; the names of the symbol sets used by any graphics text strings; any drawing default information; the descriptor text specified in the GSSAVE; and some control information.

GSSAVE is supported regardless of the current device, with the restriction that fixed-point data cannot be generated if the device is opened for family-4 output (see Chapter 18, “Device support in application programs” on page 371).

Saving all graphics on the current page

This GSSAVE call stores all the segments on the current page in a file called PIC1.

```
DCL DUMMY(1) FIXED BIN(31);
        /* Segments   Filename  Control       Description        */
CALL GSSAVE(0,DUMMY,   'PIC1',  0,DUMMY,   16,'GSSAVE EXAMPLE 1');
```

The value 0 in the first parameter causes all segments (named and unnamed) on the page to be saved, in a file called PIC1.

Another way of saving all the graphics on the current page is to specify the parameters of GSSAVE like this:

```
DCL DUMMY(1) FIXED BIN(31);
DUMMY(1) = 0;
        /* Segments  Filename   Control       Description        */
CALL GSSAVE(1,DUMMY,   'PIC1',  0,DUMMY,   16,'GSSAVE EXAMPLE 1');
```

If you specify 0 as the first element of the array of segment identifiers, the GSSAVE call saves all the graphics on the current page. If you code 0 as any other element of the array, GDDM stops reading segment identifiers and saves only those segments whose identifiers precede the 0 in the array.

Selecting individual segments to be saved

This GSSAVE call specifies that segments 7, 8, and 20 are to be saved in a file called SEGS3 using floating-point GDF orders.

```
DCL SEG_IDS(4) FIXED BIN(31);       /* Array of segment identifiers  */
SEG_IDS(1) = 7;
SEG_IDS(2) = 8;
SEG_IDS(3) = 20;
DCL CONTROL(2) FIXED BIN(31);
CONTROL(1) = 1;                     /* Don't overwrite existing file */
CONTROL(2) = 4;                     /* Floating-point GDF data       */
        /* Segments  Filename   Control          Description         */
CALL GSSAVE(3,SEG_IDS, 'SEGS3', 2,CONT_ARR,   16,'GSSAVE EXAMPLE 2');
```

It is an error to specify a nonexistent segment in the array of segment identifiers. You can discover the identifiers of all the named segments on the current page using the GSQPRI call (see “Querying the order of all segments, using GSQPRI” on page 165).

Although the array SEG_IDS contains four segment identifiers, the value in the first parameter specifies that only 3 segments are to be saved. They are stored in a file called SEGS3 in the order in which they are specified in the array.

Naming the file or data set in which the GDF data is to be saved

The name for the file in which the graphics are stored is supplied in the third parameter of the GSSAVE call. Naming conventions vary according to the subsystem and are explained in the relevant appendix of this book.

On CMS, GDDM creates a file on the end user's A-disk with the specified value as the file name, and a file type of ADMGDF. So the example would create a file called:

```
SEGS3 ADMGDF A1
```

GDDM manages the file creation and access entirely when you issue a GSSAVE call. The program or end user needs to do nothing else.

Specifying whether GDF files of the same name should be overwritten

When a program issues a GSSAVE call to save graphics in external storage, GDDM's default action is to overwrite any GDF file with the same name as that specified on the call.

If you want to retain the graphics saved by previous invocations of the program, you need to specify this in the fifth parameter of the GSSAVE call. The fifth parameter is the name of an array of one or two elements. The first element is the one that affects the overwriting of files, so a one-element array (containing a value of 1) is enough to prevent saved graphics from being overwritten.

Choosing the type of GDF data for the graphics you want to save

For the GDF orders saved by your program, you can determine whether GDDM should store the coordinates as fixed-point or floating-point data. GDDM's default action is to use floating-point data, which produces higher quality pictures.

You do not need to code the second array element in the fourth parameter of the GSSAVE call, unless you want the GDF data saved to be shorter. Fixed-point coordinate data is saved as 2-byte integers, whereas floating-point coordinates need 4-bytes of storage. Fixed-point GDF data is less accurate however and is subject to the following problems:

Severe clipping In addition to any clipping that occurs when the graphics primitives are drawn, fixed-point GDF data is clipped at the boundary of the graphics field, when it is saved.

Distortion When fixed-point GDF pictures are retrieved from storage by programs, they may be distorted, especially if they are enlarged after retrieval.

Inter-Release compatibility

GDF files generated by earlier releases of GDDM are correctly interpreted by the current release. However, GDF data generated by the current release may not be interpreted correctly by an earlier release. There may be new orders and other changes that the earlier release cannot handle. In other words, GDF is compatible upward but not downward.

GDDM does not make any guarantees about the orders that a picture generates. For instance, mode-3 graphics may sometimes generate write-text orders, but at

other times may be broken down into line and arc orders. The GDF data for a particular picture may vary from release to release.

Saving pictures in Computer Graphics Metafile, using call CGSAVE

The CGSAVE call also enables your program to save individual segments or all the graphics on the GDDM page in external storage. The graphics orders used to store pictures saved with this call are different to the GDF orders used by GSSAVE.

The CGSAVE call stores graphics drawing orders in a **Computer Graphics Metafile** (CGM). You may need to save pictures in CGM format, if you need to use them with applications such as:

- CorelDRAW
- Freelance Plus
- Harvard Graphics
- Micrografx Designer

Because GDDM normally uses GDF orders to draw graphics, CGSAVE converts the GDF orders on the GDDM page into CGM drawing orders before storing them. The CGSAVE call is similar to the GSSAVE call but it has some special parameters that relate to this conversion process. Here is an example:

```
DCL CGM_NAME(3) CHAR(8);
CGM_NAME(1) ='PICTURE';                                                  A
DCL SEG_IDS(10) FIXED BIN(31);   /* Array of segment identifiers */      B
SEG_IDS(1) = 7;
SEG_IDS(2) = 8;
SEG_IDS(3) = 20;
DCL CON_OPTS(2) FIXED BIN(31);   /* Array of control options     */
CON_OPTS(1) = 0;                 /* Overwrite existing file      */      C
CON_OPTS(2) = 437;               /* US English ASCII code page   */      D

DCL DESC_1(1) CHAR(80) INIT(
'Metafile for use with Havard Graphics produced by PICSAVE PLI program');  E

DCL DESC_2(1) CHAR(60) INIT(
'This picture shows the floor plan of the ground floor');                  F

CALL CGSAVE(1,CGM_NAME,CGMHG,3,SEG_IDS,2,CON_OPTS,61,DESC_1,54,DESC_2);    G
```

Naming the file or data set in which the CGM data is to be saved

The CGM_NAME array holds the parts of the name given to the CGM file. The number of components used in the name depends on the number specified in the first parameter of the CGSAVE call. The number in the first array must be at least 1.

Because the CGM_NAME array at **A** contains only the first part of the name for the CGM file, the other name parts are determined by the default naming convention for CGM files on the subsystem in use. These are described in the appendixes for each subsystem at the back of this book.

Using a conversion profile to store CGM orders that suit another application

The CGMHG value specified in the third parameter of the CGSAVE call at **G** in the example is the name of the conversion profile used when converting the GDF orders on the GDDM page into CGM. This conversion profile produces a form of CGM that suits the Harvard Graphics application. If the application issues any graphics orders that are not supported by Harvard Graphics, they are mapped onto supported orders.

If the CGM data you are saving is for use with an application for which a conversion profile is not supplied, you can either create one of yourself or use GDDM's general purpose conversion profile, ADM. The conversion profiles supplied with GDDM are listed inTable 2 on page 174 .

Specifying a code page for saved CGM data

You may need to tailor the conversion to suit the ASCII code page used for creating the CGM. In most cases, the default code page (850) will be used. You can do this by specifying the required code page on the CGSAVE call, as at **D** . It is probably best to use the default code page and allow the end users to specify the code page in their conversion profiles.

Including descriptive text in the CGM data saved

The text string in the DESC_1 array at **E** is associated with the “Begin Metafile” element of the CGM file. You can use this descriptor to include the name of the program that produced the CGM data.

The text string in the DESC_2 array at **F** is also associated with the “Begin Metafile” element of the CGM file. Your application can invite end users to enter a description of the saved picture and can then place this description in the array. When end users list the CGM files they have on external storage, they can identify the files by the descriptions, if not by the names.

Retrieving graphics pictures from external storage

If graphics pictures have been saved in either the GDF or CGM formats, GDDM enables application programs to retrieve such pictures and make changes to them.

GSLOAD Loads pictures saved in GDF format into the application program

CGLOAD Converts pictures saved in CGM format into GDF format and loads them into the application program.

Note: GDDM uses a subset of CGM drawing orders to perform conversions between the CGM and GDF formats. Because there isn’t a one-for-one mapping between GDF and CGM graphics drawing orders, the converted picture is not guaranteed to be identical to the original. The restrictions governing these conversions are described in the *GDDM Base Application Programming Reference* book.

Loading a saved graphics picture is equivalent to opening a new segment on the current page, reexecuting the calls that created the contents of the first saved segment, closing the new segment, and repeating this procedure for each of the specified saved segments.

When you use either GSLOAD or CGLOAD in a program, you have the choice of loading each saved picture into a separate segment or all the segments into one segment on the current GDDM page.

When a GSLOAD or a CGLOAD call is executed, all the segments in the specified file are loaded. To save and load them individually, you must store only one segment per file. This means executing a GSSAVE or CGSAVE for every segment that you add to the library, and a GSLOAD or CGSAVE for each one you retrieve.

There must be no open segment when a GSLOAD or a CGLOAD call is issued.

Retrieving pictures stored in Graphics Data Format, using call GSLOAD

At **G** in the following programming example, the GSLOAD call retrieves all the segments stored in a file called PIC1, and adds them to the current page:

```
DCL OPT_ARRAY(1) FIXED BIN(31);
OPT_ARRAY(1) = 6;       /* Starting point for renumbering segments */  A
OPT_ARRAY(2) = 3;       /* Type-3 load - Size of picture preserved */  B
OPT_ARRAY(3) = 4;       /* Drawing defaults from saved data        */  C
OPT_ARRAY(4) = 0;       /* Orders calling unknown segments ignored */  D
OPT_ARRAY(5) = 0;       /* Use symbols sets current when saved     */  E
OPT_ARRAY(6) = 2;       /* Renumber unnamed segments. Begin with   */  F
                        /* value in OPT_ARRAY(1)                   */
OPT_ARRAY(7) = 1;       /* Tag all untagged primitives with the    */  G
                        /* value in OPT_ARRAY(7)                   */
DCL COUNT FIXED BIN(31);
DCL DESC CHARACTER(20);
         /* Filename   Control      No. of segments   Description  */
CALL GSLOAD( 'PIC1',   6,OPT_ARRAY,     COUNT,          20,DESC);     H
```

The full name of the PIC1 file for which GDDM searches, depends on the subsystem in use. The naming conventions for GDF files under each of the supported subsystem are explained in the appendixes of this book.

The manner in which segments are added to the current page is determined by the OPT_ARRAY parameter. The values that you can specify for this and the other parameters of GSLOAD are described in detail in the *GDDM Base Application Programming Reference* book. Some advice about using the elements of the OPT_ARRAY is given in “Avoiding clashes between the identifiers of new and loaded segments” on page 181 through “Loading a GDF file that contains unnamed segments” on page 182 and in “The three types of load” on page 183.

In general, segments are loaded in the order in which they were saved. The first segment saved on the file therefore has the lowest priority (see “Drawing chain and segment priority” on page 164) and the last the highest.

GSLOAD is supported no matter what device is current when it is executed, or what device was current when the segments were saved.

The segments are always loaded into the viewport that is current when the GSLOAD is executed. A graphics window, and any of the objects in the graphics hierarchy, can be set up with default values if they do not already exist when the GSLOAD call is executed.

Avoiding clashes between the identifiers of new and loaded segments

If a graphics segment has been stored with the same identifier as one in your program, you must you change its identifier when you load it onto the GDDM page.

At **A** in the example, the value 6 is specified as the starting point for renaming segments loaded from external storage. By specifying a number higher than any identifiers used in your program and renaming all the loaded segments from this point upwards, you can be sure of loading all the saved segments successfully.

If GDDM finds a call-segment order from a segment within the GDF to another segment within the GDF, it changes the identifier of the called segment to its new identifier.

Specifying whether saved segments be transformed when loaded

There are a number of ways that you can use GSLOAD to load a graphics segment from a GDF file, each of which affects the appearance of the loaded graphics differently.

These **load types** are specified using the second element of the OPT_ARRAY parameter. The positioning algorithms and major uses of the three types of load are explained further in “The three types of load” on page 183.

The effects of each type of load are illustrated by Figure 54 on page 183 and Figure 55 on page 183. Figure 54 shows three segments (a yellow circle, a white square, and a set of blue triangles) as they were saved. Figure 55 shows them after each has been loaded with a different type of GSLOAD call.

Loading the drawing-default definitions of saved graphics

When graphics pictures are saved in Graphics Data Format, the definitions of drawing defaults that are current when the GSSAVE call is issued are saved with the drawing orders. If you intend to add new graphics segments to some segments loaded from external storage, you may need to consider whether the saved drawing defaults differ from those in your program.

The default action when a program loads graphics from a GDF file, is for the saved defaults to apply to any loaded segments and for the current drawing defaults to apply to any new segments created by the program.

If you want new graphics to appear similar to the loaded graphics, you can specify that the saved drawing-default values replace the current values, as at **C** in the example in “Retrieving pictures stored in Graphics Data Format, using call GSLOAD” on page 180.

In the loaded data, segments that are called and chained do not inherit, from the caller, the attributes for which drawing default values were specified. The reloaded picture may therefore appear different from the saved picture.

Coping with GDF orders that call unsaved segments

Sometimes GDF files do not contain all the data needed to make the complete picture. This can happen if, in the program that generates the graphics, one segment calls another but only the calling segment is saved.

The fourth element of the OPT_ARRAY parameter enables you to specify the action to be taken when loading a GDF object that contains call-segment orders to segments that do not exist in the object.

You can instruct GDDM to ignore such orders or you can code your application to resolve the discrepancy. If you know what the graphics in the unknown segment should look like, you can recreate the segment in your program using the identifier of the unknown segment that the loaded GDF file calls. The loaded segment then calls the new segment.

Loading a GDF file that refers to a symbol set

After loading the segments, GDDM automatically loads any symbol sets that were loaded at the time the segments were saved, whether they were used or not. It also loads them with the same identifiers, regardless of whether any symbol sets have already been loaded in your program.

The fifth element of the OPT_ARRAY parameter allows you to specify GDDM's action, if there is a risk of a symbol set loaded by the GDF having the same identifier as one loaded by your program. You can specify whether your program's or the GDF file's symbol-set identifiers take priority.

At **E** in the example in “Retrieving pictures stored in Graphics Data Format, using call GSLOAD” on page 180, the value 0 is coded on this element of the array, allowing the symbol sets that were current when the GDF was saved to be loaded with the graphics.

Before a GDF file containing saved segments is loaded, it is not possible to query what symbol sets it uses.

Loading a GDF file that contains unnamed segments

If a GDF file has been created by a GSSAVE call had the value 0 in its second parameter, it can contain graphics stored in segments with identifier 0. Using the sixth element of the OPT_ARRAY parameter (in conjunction with the first element), you can renumber unnamed segments. This action is specified at **F** in the example in “Retrieving pictures stored in Graphics Data Format, using call GSLOAD” on page 180. Your program can then manipulate the graphics in each of these segments separately, which is not possible if they are all loaded together as unnamed segments.

Loading a GDF with untagged primitives

If you are loading a GDF to edit it, and some or all of the primitives are untagged, they will not be eligible for picking (using GSQPIK) or subsequent operations. Specifying a value greater than zero at **G** in the example in “Retrieving pictures stored in Graphics Data Format, using call GSLOAD” on page 180 means that any untagged primitives will be tagged with this value and can then be picked.

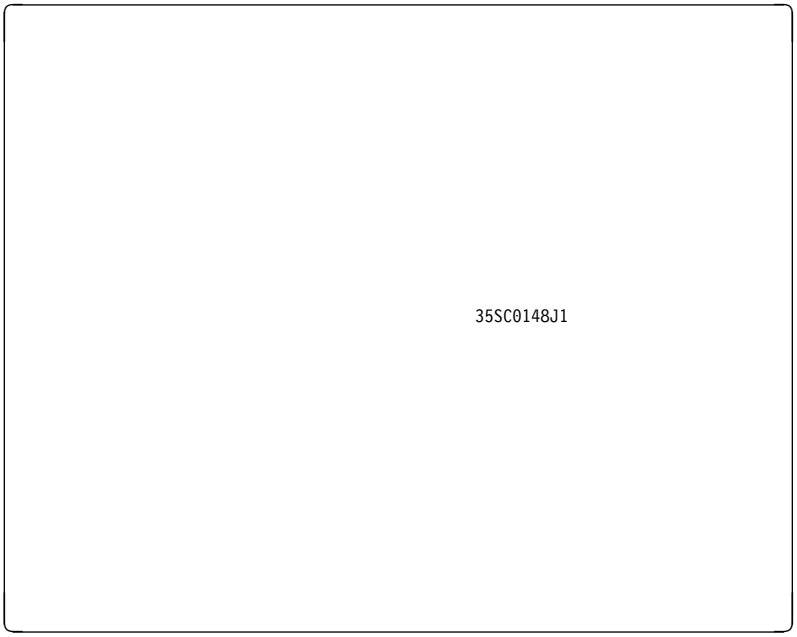


Figure 54. Segments as saved

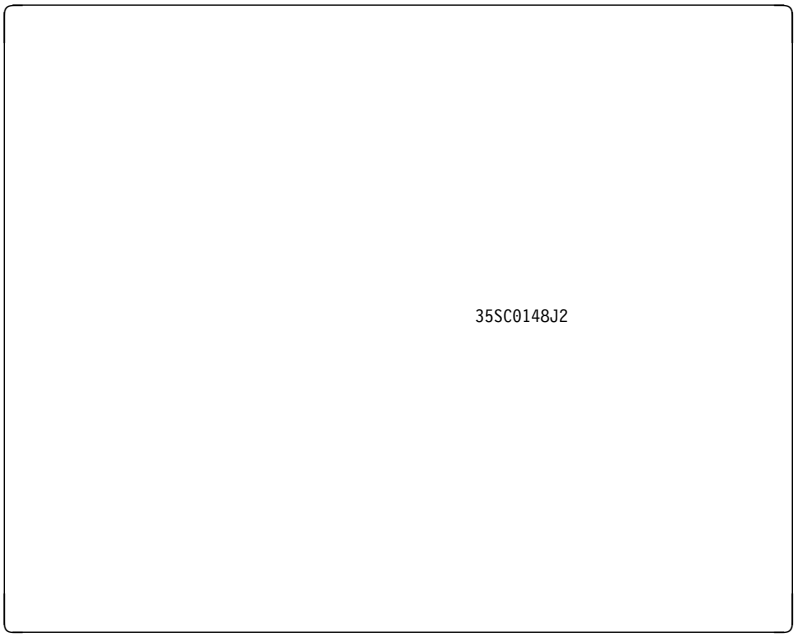


Figure 55. Segments as loaded

The three types of load

When you issue a GSLOAD call, the second element of the array in the third parameter enables you to specify what transformation, if any, you want GDDM to perform on the segment as it is loaded. The three possible types of load you can specify are described below.

Load type 1: Preserving the coordinates of the picture

If you are writing an application that composes pictures on the GDDM page by retrieving several segments from external storage, it is best to specify type-1 load on the GSLOAD call. Your application can then retrieve segments from a segment library and the primitives in the segments retain the world-coordinate values they had when they were saved (see “Maintaining a library of segments” on page 188).

If the graphics window in your program has the same size and origin as those used to draw the stored segments, then the segments appear the same when loaded. If the physical size of the world-coordinate units differ, the primitives in the loaded segment change size. If the two window origins are in different places, the positions of primitives in the loaded segment change.

The world-coordinate units in Figure 55 are physically smaller than those in Figure 54, so the yellow circle has shrunk. The origin is in the center of the screen in both figures, and the origin of the circle is at its center, so the circle remains in the center of the screen.

Applications that build up pictures by retrieving segments with type-1 load should use the same unit of measurement as the segments. This is not to say that the picture must always be displayed at the same scale. You can define the window (with a GSWIN or GSUWIN call) to contain any suitable number of world-coordinate units, and thereby display the picture at any suitable scale.

Positioning segments retrieved by type-1 load: GDDM puts the origin of the loaded segment at the origin of the current world coordinates. If the primitives are actually positioned well away from the origin of the world coordinates, or if this origin is outside the current window, the primitives may not appear on the display. The way to ensure that loaded graphics pictures appear on the display is described in “Maintaining a library of segments” on page 188.

Making sure that loaded segments appear on the display: To ensure that at least some of the primitives fall within the current window and appear on the display following a load, you must do two things:

1. Before saving the segment, ensure that its origin is at a reasonably central point with respect to the primitives. If the primitives were not drawn centrally about the origin of the world-coordinate system, the segment origin can be moved before the GSLOAD:

```
DCL SEG_ID(1) FIXED BIN(31);
DCL DUMMY(1) FIXED BIN(31);
  .
  .
CALL GSUWIN(0.0,100.0,0.0,100.0); /* Define coordinate system.  */
CALL GSSEG(4);
CALL GSMOVE(40.0,50.0);
CALL GSARC(50.0,50.0,360.0);      /* Draw circle centered       */
CALL GSSCLS;                      /* at 50,50.                  */
  .
  .
CALL GSSORG(4,50.0,50.0);         /* Make center of circle the  */
                                  /* segment origin.            */
SEG_ID(1) = 4;
```

```
         /* Segments  Filename     Control   Description */
CALL GSSAVE(1,SEG_ID,  'SEG4',     0,DUMMY,      0,'');
```

In practice, the application is likely to allow the terminal operator to choose the reference point of the segment before saving it, in a way similar to that recommended before transforming it (see “Local origin when transforming a segment” on page 223).

2. After loading, move the segment so that its origin is within the current window:

```
DCL CNTRL(2) FIXED BIN(31);
DCL COUNT FIXED BIN(31);
DCL CH1 CHARACTER(1);
DCL (X1,X2,Y1,Y2) FLOAT DEC(6);
  .
  .
CALL GSUWIN(X1,X2,Y1,Y2);
  .
  .
CNTRL(1) = 101;                   /* New segment identifier.    */
CNTRL(2) = 1;                     /* Keep original              */
                                  /* world coordinates.         */

         /* Filename   Control  No. of segments   Description   */
CALL GSLOAD( 'SEG4',   2,CNTRL,     COUNT,           0,CH1);

CALL GSSPOS(101,(X1+X2)/2,(Y1+Y2)/2);
                                  /* Move segment so that its   */
                                  /* origin is in middle of the */
                                  /* current window.            */
```

Clipping must be off (the default) at the time of the GSLOAD, otherwise any part of the segment that falls outside the window is lost.

Load type 1 can also be useful for applications that draw pictures that are larger than the screen. GDDM’s User Control facility enables end users to pan (move the display window sideways and vertically) over the graphics or zoom (change the physical size at which they are displayed). However, if you want your application to provide these functions itself you can use load type 1 to implement it. More information on this use of type-1 load is given under “Panning and zooming” on page 189 and an example is provided in Figure 58 on page 189.

Load type 2: Maximizing the size of the picture

When an application loads a segment using type-2 load, the coordinates of its primitives are transformed so that the segment is as large as possible. This is illustrated in Figure 56 on page 186, using the white square from Figure 55 on page 183.

More precisely, the transformation is such that the picture space current at the time of the GSSAVE would fill the viewport current at the time of the GSLOAD. The aspect ratio is preserved. This means that, in general, the viewport is filled in one direction only; the width in the illustration. The positioning is such that the picture space would be centered in the other direction; vertically in the illustration. The segment's position can be altered after loading with a GSSPOS call, if it was created with the transformable attribute.

GDDM maps the save-time picture space onto the load-time viewport, rather than mapping viewport to viewport. It does this because the latter mapping would not work when GSSAVE stores segments from multiple viewports.

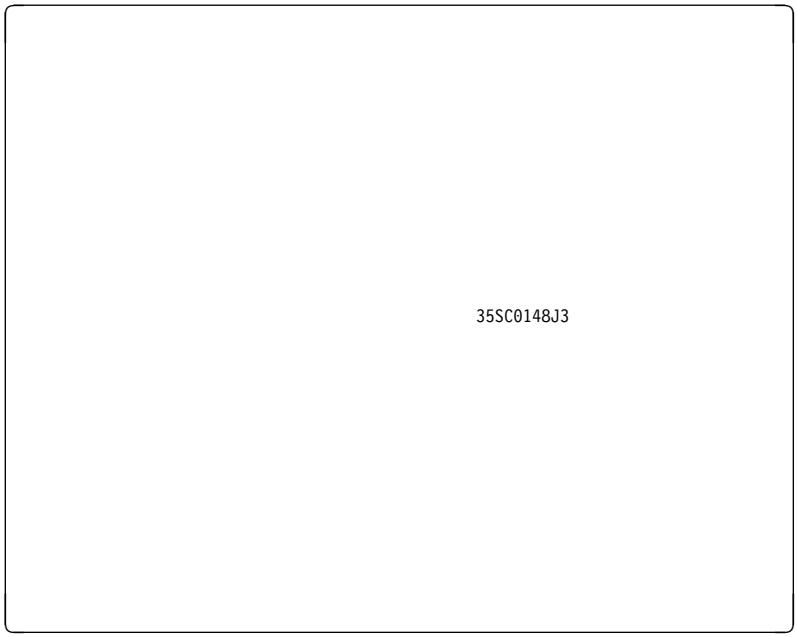


Figure 56. Type 2 load

The primary use for the type 2 load is in copying from one device to another, when the physical size of the graphics is not significant. Typically, it is used to create a hard copy of a screen display, making full use of the paper area of the printer or plotter. Here is some typical code to make a copy using a type 2 load. (It is necessary to use the DSOPEN and DSUSE calls, which are described in Chapter 18, “Device support in application programs” on page 371.)

```
DCL P_LIST(1) FIXED BIN(31);
DCL N_LIST(1) CHAR(8);
DCL CNTRL(2) FIXED BIN(31);
DCL COUNT FIXED BIN(31);
DCL DUMMY(1) FIXED BIN(31);
DCL DESC CHAR(1);
DCL (ATTYPE,ATTVAL,ATTCNT) FIXED BIN(31);


                              .
                              .
/*                   Draw the picture                          */
                              .
                              .

CALL ASREAD(ATTYPE,ATTVAL,ATTCNT); /* Send to terminal.        */
```

```
IF ATTVAL = 4                      /* PF4 key, print the picture*/
  THEN DO;
  SEG_IDS(1) = 0;                  /* Save the whole picture.   */

         /*      Segments    Name    Options   Description      */
  CALL GSSAVE(  1,SEG_IDS, 'TEMPIC', 0,DUMMY,      0,'' );

  N_LIST = 'P1';                   /* Printer name.             */
  CALL DSOPEN(1,2,'*',0,P_LIST,1,N_LIST); /* Open printer, make */
  CALL DSUSE(1,1);                 /* it the primary device.    */


  CNTRL(1) = 0;                    /* No need to change seg ids.*/
  CNTRL(2) = 2;                    /* Print as large as possible*/

         /* Obj-name Arr-cnt Array  Seg-cnt Descrip-len Descrip */
  CALL GSLOAD('TEMPIC', 2,   CNTRL,  COUNT,      0,     DESC);

  CALL FSFRCE;                     /* Send to printer           */
 END;
```

Load type 3: Preserving the size of the picture

After a segment is loaded, GDDM transforms the coordinates of its primitives to keep the size of the picture the same. All primitives have the same physical size when displayed on the current device as they had on the device that was current when the segment was saved.

The position of the segment is such that the bottom left-hand corner of the picture space current at the time of the GSSAVE would be at the origin of the viewport current at the time of the GSLOAD. The segment's position can be altered after loading using a GSSPOS call.

This positioning is illustrated in Figure 57 on page 188, using the blue triangles from Figure 55 on page 183. The origin of the viewport is at the center of the display.

The right-hand edge of the save-time picture space boundary has disappeared because it is beyond the edge of the graphics field. However, graphics are irretrievably lost only if clipping was on at the time of the GSLOAD. If clipping was not on (which is the default), then the disappeared graphics could be displayed by panning the window, as described in “Panning and zooming” on page 189.

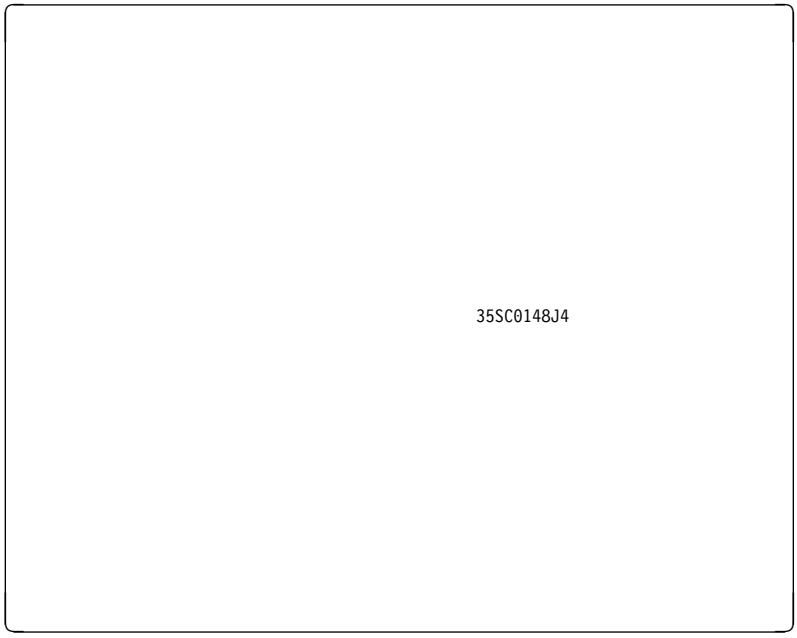


Figure 57. Type 3 load

Type 3 loads are mainly used in copying scale drawings from one device to another. Consider, for example, a geographical map that has been created with world coordinates such that, when plotted on a particular plotter, it has a scale of one centimeter to the kilometer.

The map can be saved, and subsequently retrieved using a type 3 load, on a different current device. This might be a different plotter, or a printer, or even a display unit. GSLOAD transforms the map's coordinates to ensure that when it is sent to the new device, its graphics primitives have the same physical size as on the original plotter. The new output therefore maintains the map's original scale of one centimeter to the kilometer.

The example code in “Load type 2: Maximizing the size of the picture” performs in this way if

```
CNTRL(2)=3;
```

is coded in place of

```
CNTRL(2)=2;.
```

Maintaining a library of segments

Many applications can benefit from a library of picture components. For instance, an office layout program would store drawings of all available office furniture. Each file in the library would contain one piece of furniture, in one or more segments. A general drafting application would enable the end user to create segments as required, and store and retrieve them at will.

The GSLOAD call always loads all the segments in a file. In many cases, a program needs to access segments one at a time. For instance, it might be required to let the operator select a segment from a menu and drag it around the screen. Or it might need to load different segments into different viewports. In such cases, the library must contain only one segment per file. The GSSAVE calls

that create the files must therefore explicitly specify one segment identifier. Subsequent GSLOAD calls retrieve one file each.

All the segments in a library should be drawn using the same metric unit. In other words, one world-coordinate unit should represent the same physical measurement in a real object in all cases. One world-coordinate unit could represent a micron, a millimeter, or a light-year–provided that all segments use the same unit.

Panning and zooming

If the window is altered, either to change the physical size of the picture (zooming) or to alter the portion of it that is actually displayed (panning or scrolling), then all the graphics must be redrawn. One way of doing this is to save the picture, clear the graphics field, alter the window, and load the picture. The GSLOAD is equivalent to reexecuting all the graphics primitive calls that created the picture. The following example shows how to use this technique.

```
DCL (ATTYPE,ATTVAL,ATTCNT) FIXED BIN(31) INIT(0);/*ASREAD params*/
DCL (X1,X2,Y1,Y2,XDISP,YDISP) FLOAT DEC(6);  /* Window variables*/
DCL SEG_IDS(1) FIXED BIN(31);       /* GSSAVE segment ids       */
DCL COUNT FIXED BIN(31);            /* GSSAVE parameter count   */
DCL CNTRL(2) FIXED BIN(31);         /* GSSAVE parameters        */
DCL CHAR CHAR(1);                   /* GSLOAD description dummy */
DCL DUMMY(1) FIXED BIN(31);         /* GSLOAD segment id dummy  */
DCL SAVED BIT(1) INIT('0'B);        /* Pan/zoom save flag       */

/*****************************************************************/
/*                 Draw the picture                              */
/*****************************************************************/
  .
  .
SEND:
CALL ASREAD(ATTYPE,ATTVAL,ATTCNT);  /* Send to terminal         */
/*****************************************************************/
/*                 Panning and zooming code                      */
/*****************************************************************/

IF ATTVAL=7  | ATTVAL=8  | ATTVAL=9 /* If pan or zoom button, ..*/
 | ATTVAL=10 | ATTVAL=11 | ATTVAL=12/* .. then ..               */
  THEN IF ¬SAVED                    /* ..if picture not already */
    THEN DO;                        /* saved.                   */
      CALL GSSAVE(0,DUMMY,'ZOOMTEM',0,DUMMY,0,'');/*Save picture*/
      SAVED = '1'B;                 /* Set save flag.           */
      /*       SET UP CONTROL PARAMETERS FOR GSLOAD             */
      CNTRL(1) = 0;                 /* Keep original segment ids*/
      CNTRL(2) = 1;                 /* Preserve world coords    */
    END;
```

Figure 58 (Part 1 of 3). Program using type-1 load to let users to pan and zoom a saved graphics picture

```
IF ATTVAL = 7                       /* PF7 key is pan up.       */
  THEN DO;
  CALL GSCLR;                       /* Clear graphics field.    */
  YDISP = (Y2-Y1)/2;
  Y1 = Y1 + YDISP;                  /* Move window up by half   */
  Y2 = Y2 + YDISP;                  /* its height.              */
  CALL GSUWIN(X1,X2,Y1,Y2);

      /*   Object-name Array-Count Array  Seg-Count Description */
  CALL GSLOAD( 'ZOOMTEM',    2,    CNTRL,  COUNT,    0,CHAR);
  GO TO SEND;
END;
IF ATTVAL = 8                       /* PF8 key is pan down.     */
  THEN DO;
  CALL GSCLR;                       /* Clear graphics field.    */
  YDISP = (Y2-Y1)/2;
  Y1 = Y1 - YDISP;                  /* Move window down by half */
  Y2 = Y2 - YDISP;                  /* its height.              */
  CALL GSUWIN(X1,X2,Y1,Y2);

      /*   Object-name Array-Count Array  Seg-Count Description */
  CALL GSLOAD( 'ZOOMTEM',    2,    CNTRL,  COUNT,    0,CHAR);
  GO TO SEND;
 END;

IF ATTVAL = 9                       /* PF9 key is enlarge.      */
  THEN DO;
  CALL GSCLR;                       /* Clear graphics field.    */
  XDISP = (X2-X1)/4;
  YDISP = (Y2-Y1)/4;
  X1 = X1 + XDISP;                  /* Halve size of the window */
  X2 = X2 - XDISP;                  /* without altering x,y     */
  Y1 = Y1 + YDISP;                  /* coordinates of center.   */
  Y2 = Y2 - YDISP;
  CALL GSUWIN(X1,X2,Y1,Y2);

       /*   Object-name Array-Count Array Seg-Count Description */
   CALL GSLOAD( 'ZOOMTEM',    2,    CNTRL,  COUNT,    0,CHAR);
   GO TO SEND;
 END;
```

Figure 58 (Part 2 of 3). Program using type-1 load to let users to pan and zoom a saved graphics picture

```
IF ATTVAL = 10                      /* PF10 key is pan left.    */
  THEN DO;
  CALL GSCLR;                       /* Clear graphics field.    */
  XDISP = (X2-X1)/2;
  X1 = X1 - XDISP;                  /* Move window left by half */
  X2 = X2 - XDISP;                  /* its width.               */
  CALL GSUWIN(X1,X2,Y1,Y2);

       /*   Object-name Array-Count Array Seg-Count Description */
  CALL GSLOAD( 'ZOOMTEM',    2,    CNTRL,  COUNT,    0,CHAR);
  GO TO SEND;
END;

IF ATTVAL = 11                      /* PF11 key is pan right.   */
  THEN DO;
  CALL GSCLR;                       /* Clear graphics field.    */
  XDISP = (X2-X1)/2;
  X1 = X1 + XDISP;                  /* Move window right by half*/
  X2 = X2 + XDISP;                  /* its width.               */
  CALL GSUWIN(X1,X2,Y1,Y2);

       /*   Object-name Array-Count Array Seg-Count Description */
  CALL GSLOAD( 'ZOOMTEM',    2,    CNTRL,  COUNT,    0,CHAR);
  GO TO SEND;
 END;

IF ATTVAL = 12                      /* PF12 key reduce picture  */
  THEN DO;
  CALL GSCLR;                       /* Clear graphics field.    */
  XDISP = (X2-X1)/2;
  YDISP = (Y2-Y1)/2;
  X1 = X1 - XDISP;                  /* Double the size of the   */
  X2 = X2 + XDISP;                  /* window without altering  */
  Y1 = Y1 - YDISP;                  /* the x,y coordinates      */
  Y2 = Y2 + YDISP;                  /* of the center.           */
  CALL GSUWIN(X1,X2,Y1,Y2);

       /*   Object-name Array-Count Array Seg-Count Description */
  CALL GSLOAD( 'ZOOMTEM',    2,    CNTRL,  COUNT,    0,CHAR);
  GO TO SEND;
 END;
```

Figure 58 (Part 3 of 3). Program using type-1 load to let users to pan and zoom a saved graphics picture

Notice that the picture is saved only once, and a flag is set to record the action. Repeated saving and loading distorts the picture eventually, because of the accumulation of rounding errors in the coordinates.

Retrieving pictures stored in Computer Graphics Metafiles, using call CGLOAD

You can load computer graphics metafiles (CGM) onto the current GDDM page from auxiliary storage using the CGLOAD call. If you want to modify the picture, you need to load each graphics primitive into its own unique segment on the page. Otherwise you can load all primitives into one segment.

If any primitives in the CGM file were drawn using CGM drawing default attributes, those attributes are preserved on the GDDM page for the loaded segments. These CGM defaults have no effect on any other primitives on the page.

In the following programming example, the CGLOAD call retrieves all the pictures stored in a file called CGMPIC1, converts them to GDF orders and adds them to the current page:

```
DCL CGM(1) CHAR(8);
NAME(1)='CGMPIC1';     /* Name of CGM file containing pictures      */
NAME(2)='CGM';
DCL PROF(1) CHAR(8);
PROF(1)='ADMFP3';      /* Conversion profile for Freelance V3       */

DCL OPT_ARRAY(1) FIXED BIN(31);
OPT_ARRAY(1) = -1;  /* Load all the pictures in the input file      */
OPT_ARRAY(2) = 9;   /* Starting point for renumbering segments      */
OPT_ARRAY(3) = 4;   /* Type-4 load. Picture space from CGM file     */
OPT_ARRAY(4) = 0;   /* Use value in conversion profile              */
                       /* ADMFP3 has 2; don't overwrite symbol sets */
OPT_ARRAY(5) = 0;   /* Use value in conversion profile              */
                       /* ADMFP3 has 2; use a single segment        */
OPT_ARRAY(6) = 0;   /* Use value in conversion profile              */
                       /* ADMFP3 has 850: ASCII multilingual        */
                       /* code page                                 */
DCL(RLEN_1,CNT,RLEN_2) FIXED BIN(31);
DCL DESC_1(1) CHAR(70);
DCL DESC_2(1) CHAR(70);

CALL CGLOAD(2,NAME,PROF,6,OPT_ARRAY,CNT,70,DESC_1,RLEN_1,70,DESC_2,RLEN_2) ;
```

Retrieving pictures stored in Picture Interchange Format, using call GSLOAD

Picture Interchange Format (PIF) files are used to store graphics pictures on PC DOS and 3270-PC/G and /GX workstations, and can be interchanged between a workstation and the host computer. PIF orders are similar to GDF.

In a typical case, an end user creates a PIF file at the workstation, sends it to the host where a utility converts it to a GDF file. An application program can load the picture onto the GDDM page using the GSLOAD call. The reverse route can also be followed: a host-created GDF file can be converted to PIF and sent to the workstation.

The SEND and RECEIVE utilities used to transfer PIF files between the workstation and GDDM on the host computer are described in the *GDDM Base Application Programming Reference* book.

Modifying graphics pictures that have been loaded into your program

This section explains how you can write GDDM application programs that modify the GDF drawing orders of saved pictures.

Whether a picture is saved in GDF, CGM, or PIF format in external storage, once it has been loaded onto the GDDM page it is in Graphics Data Format. An application program can use a series of calls known as a GET operation to place the GDF orders on the current GDDM page in a variable. The program can then modify the data in this variable so that the picture is changed when it is restored to the GDDM page using the GSPUT call.

You can write programs to interpret the GDF orders returned by the GET, operation or to supply new or updated GDF orders to GDDM by means of a PUT operation. Some uses for such programs are:

- Changing pictures previously defined by the application using ordinary GDDM calls (GSSEG, GSCOL, GSLINE, and so on). This is the only way that you can change primitives and attributes after they have been defined. However, note the comments made in “Inter-Release compatibility” on page 177.
- Transferring pictures to and from devices not supported by GDDM,
- Converting pictures to and from other application programming interfaces.

Placing graphics data from the GDDM page in a program variable

The GET operation that places GDF data in a program variable is initiated by a GSGETS call. This is followed by one or more GSGET calls, the number depending on the complexity of the picture, and hence the total size of the GDF orders in relation to the size of the specified program variable. GSGETE ends the retrieval of GDF data, the last GSGET must be followed by a GSGETE.

The program in Figure 59 on page 194 shows how to use GSGETS, GSGET, GSGETE, and GSPUT. It makes use of a comment order at the start of the GSGET data to set up the window before the GSPUT. The format of this, and all other GDF orders, is explained in the *GDDM Base Application Programming Reference* book.

```
GETPUT: PROC OPTIONS (MAIN);
*****************************************************************/
*  DECLARATIONS                                                 */
*****************************************************************/
DCL ADDR BUILTIN;
DCL GDFFILE RECORD SEQUENTIAL
                        OUTPUT FILE;/* File to store GDF       */
DCL GETARRAY(3) FIXED BIN(31);       /* Parameter to GSGET      */
DCL GETCNT FIXED BIN(31);            /* Length of GETARRAY      */
DCL BUFLEN FIXED BIN(31);            /* Data buffer length      */
DCL BUFDATA(10) CHAR(400);           /* Save buffers allocated  */
DCL GDFLEN(10) FIXED BIN(31);        /* Data actual lengths     */
DCL (TYPE,MODE,COUNT) FIXED BIN(31); /* Params for ASREAD       */
DCL P PTR;                           /* To address first order  */

DCL 1 COMMENT BASED(P),              /* Comment order structure */
     2 OPCODE BIT(8),                /* Order OPCODE            */
     2 LEN BIT(8),                   /* Data length in order    */
     2 FORMAT BIT(16),               /* Data format in order    */
     2 XLO FIXED BIN(15),            /* x coord  low  limit     */
     2 XHI FIXED BIN(15),            /* x coord  high limit     */
     2 YLO FIXED BIN(15),            /* y coord  low  limit     */
     2 YHI FIXED BIN(15);            /* y coord  high limit     */
DCL (XLOFL,XHIFL,YLOFL,YHIFL) FLOAT DEC(6);   /* Call parameters*/


CALL FSINIT;
/*****************************************************************/
/*  Picture creation                                             */
/*****************************************************************/
/*        .            */
/*        .            */
/*        .            */
CALL ASREAD(TYPE,MODE,COUNT);        /* Output the picture      */

/*****************************************************************/
/*   Begin data capture into GDF buffers                         */
/*****************************************************************/
GETCNT=3;                            /* 3 elements in GETARRAY  */
GETARRAY(1)=0;                       /* Capture all segments.   */
GETARRAY(2)=2;                       /* Fixed point form.       */
GETARRAY(3)=2;                       /* Full GDF                */
CALL GSGETS(GETCNT,GETARRAY);        /* Start data capture.     */
BUFLEN=400;                          /* 400-byte buffers.       */
```

Figure 59 (Part 1 of 3). Handling GDF data with GSGET and GSPUT

```
/*****************************************************************/
/*   Loop until all orders captured or no more buffers          */
/*****************************************************************/
DO J=1 BY 1 UNTIL(GDFLEN(J)=0 | J= 10);
   CALL GSGET(BUFLEN,BUFDATA(J),GDFLEN(J));
   WRITE FILE(GDFFILE) FROM(BUFDATA(J)); /* Write data to file  */
END;                                  /* until all data captured.*/
CALL GSGETE;                          /* End data capture.       */
/*   DATA CAPTURE END                                            */
JSAVE=J;

/*****************************************************************/
/*   Clear the displayed picture.                                */
/*****************************************************************/
CALL GSCLR;

/*****************************************************************/
/*   Data restore from GDF buffers.                              */
/*****************************************************************/
/*   Establish GDF-dictated graphics picture space and window.  */
P=ADDR(BUFDATA(1));                /* Start of 1st order(comment)*/
XLOFL=XLO;                         /* Convert to floating point  */
XHIFL=XHI;
YLOFL=YLO;
YHIFL=YHI;

/*   Establish GDF-dictated window coordinates                   */
CALL GSUWIN(XLOFL,XHIFL,YLOFL,YHIFL);
DO J=1 TO JSAVE-1;                        /* For all buffers used*/
  CALL GSPUT(2,GDFLEN(J),BUFDATA(J));     /* Restore the picture */
END;

/*   Output the restored picture                                 */
CALL ASREAD(TYPE,MODE,COUNT);
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END GETPUT;
```

Figure 59 (Part 2 of 3). Handling GDF data with GSGET and GSPUT

Device variations with GDF

Fixed-point GDF data cannot be retrieved by means of GSGET when the current device is a 3279, 5080, 6090, or printer enabled for family-4 output (output families are explained in Chapter 20, “Sending output from an application to a printer” on page 399).

Chapter 11. Writing interactive graphics applications

This section tells you how to make the screen of a display unit into a drawing pad. It describes GDDM calls and programming techniques that create and manipulate graphics primitives interactively, in response to the demands of the end user.

The GDDM facilities for creating graphics are basically the same in interactive graphics applications as in noninteractive ones. In other words, you define graphics attributes and create graphics primitives and segments using the calls introduced in “Example: The HOUSE program” on page 6.

Interactive graphics applications differ in the calls they use for handling input from the terminal. These calls help you by letting GDDM do a lot of the work.

The GDDM-supported terminals most suitable for interactive graphics are workstations running GDDM-OS/2 Link and the 3270-PC/G and /GX workstations. This section tells you primarily how to write programs for these. Differences on other types of terminal are described in “Device variations with interactive graphics” on page 229.

Overview of graphics input functions

The type of input a host computer may receive depends on the type of terminal that sent it. To help programs be device-independent, the GDDM interactive graphics calls present all input as if it comes from **logical input devices**, rather than physical facilities of a terminal.

There are five types of logical input device:

- **Choice devices**, which correspond to the keys on the terminal that can cause an interrupt, such as the ENTER key and PF keys. The ordinary alphanumeric data keys can also cause interrupts. Choice devices provide input as a code identifying which key was pressed.
- A **locator device**, which corresponds to the graphics cursor, and provides input as an (x,y) screen position expressed in world coordinates.
- A **pick device**, which also corresponds to the graphics cursor. It differs from a locator in providing input as the identifier of a graphics primitive that has been selected, or **picked**, by the end user. The identifier is called a **tag**. The identifier of the segment to which the picked primitive belongs is also returned. The workstation creates the input by translating the (x,y) position of the cursor into a tag and a segment identifier, a process called **correlation**.
- A **string device**, which consists of graphics text typed by the end user into an area of the graphics field defined by the application program.
- A **stroke device**, which, like the locator, corresponds to the graphics cursor. It differs in providing a set of (x,y) coordinates sampled from the trajectory of a moving cursor. The sampling is either at intervals fixed by GDDM or at points indicated by the end user with the mouse or puck buttons or the stylus tip-switch.

The 3270-PC/G and GX workstations provide several ways of controlling the graphics cursor: a mouse; a tablet with either a puck (four-button cursor) or stylus;

© Copyright IBM Corp. 1982, 1996

or, if neither a mouse nor a tablet is plugged in, the cursor keys. Any of these physical devices can provide locator, pick, and stroke input, except that the cursor keys cannot provide stroke input. The input data is completely independent of the physical device; application programs cannot, in general, determine which is used.

The end user has a separate alphanumeric cursor for typing into alphanumeric fields. It is positioned using the cursor keys. If the workstation has neither a mouse, puck, nor stylus, these keys control the graphics cursor as well. The end user switches between alphanumeric and graphics cursor control by holding down the ALT key and pressing PF24.

The graphics cursor is a type of **device echo**. In general, an echo is the immediate feedback that the workstation provides for the end user. In the case of a pick, locator, or stroke device, the echo indicates the device's position. In the case of a string device, it indicates the characters that the end user has typed in.

After positioning the graphics cursor for pick, locator, or stroke data, or after typing string data, the end user must **trigger** the logical input device (that is, start the transmission to the host) by, for instance, pressing the ENTER key.

Your program may need input from some logical devices but not others. All those from which it requires input must be **enabled**. GDDM discards input from devices that are not enabled.

There is a special call, GSREAD, for interactive graphics I/O. It sends the current page to the terminal and waits for input, just like ASREAD. It differs from ASREAD in that it presents the input as if it came from one or more of the logical input devices. It adds elements to a graphics input queue–one element for each logical input device that has provided input. Your program accesses the queue by executing query calls, of which there is one for each type of logical device.

GSREAD reads any data that the end user may have typed into alphanumeric fields, in addition to graphics data.

Simple interactive graphics program

The routine in Figure 60 on page 199 illustrates graphics input in general, and pick input in particular. It displays a menu of vector symbols and enables the end user to select one of them.

The symbols can be created with the Vector Symbol Editor. They might be, for instance, diagrammatic representations of electrical components such as transistors and resistors, or plan views of office furniture, such as desks, chairs, and filing cabinets. The end user makes the selection by first positioning the cursor over a symbol and then triggering the pick. This is done by pressing ENTER or a PF key or, if no stroke device is enabled, by pressing a button on the mouse or puck, or by using the stylus tip-switch. If the data keys are enabled as a choice device and no string device has been enabled, any data key triggers the pick too.

The MENU routine could be a subroutine of an application in which the end user picks the symbols from the menu and then positions them, for instance, on a circuit diagram or office plan.

The program introduces several important calls and concepts.

```
MENU: PROC(SYMB_TAG);

DECLARE SYMB_TAG                FIXED BINARY(31),
        SEG_NUM                 FIXED BINARY(31);
DECLARE (DEV,DEVID)             FIXED BINARY(31);

CALL GSSATI(1,1);               /* Make segment detectable.     */ A

CALL GSSEG(5);                  /* Open segment with id=5.      */ B
CALL GSCM(3);                   /* Vector symbol mode.          */
CALL GSCB(10.0,10.0);           /* Set size of symbols          */
CALL GSMOVE(1.0,1.0);

CALL GSTAG(1);                  /* Tag = 1                      */ C
CALL GSCHAP(1,'A');             /* Draw first symbol.           */
CALL GSTAG(2);                  /* Tag = 2                      */ C
CALL GSCHAP(1,'B');             /* Draw second symbol           */
CALL GSTAG(3);                  /* Tag = 3                      */ C
CALL GSCHAP(1,'C');             /* Draw third symbol            */

/*    .                            Repeat                      */
/*    .                                 for remaining          */
/*    .                                               symbols    */

CALL GSENAB(3,1,1);             /* Enable pick device           */ E

CALL GSREAD(1,DEV,DEVID);       /* Send menu to terminal        */ F
                                /* and wait for input.          */
CALL GSENAB(3,1,0);             /* Disable pick device.         */ G
CALL GSQPIK(SEG_NUM,SYMB_TAG);  /* Query selected symbol.       */ H

/*Tag of selected symbol returned to calling routine in SYMB_TAG*/

END MENU;
```

Figure 60. Graphics menu routine

Tags: If a primitive is to be picked, it must have a tag. GDDM returns the tag to your program if the primitive is picked. The tag is assigned when a primitive is created. You specify tags in the GSTAG call. It has only one parameter: a fullword integer that is to become the current tag:

```
CALL GSTAG(15);/* Assign tag of 15 to all subsequent primitives */
```

GDDM assigns this tag to all subsequently created primitives within the current segment until another GSTAG is issued. When a segment is opened, GDDM makes the drawing default tag current.

To be detectable, a primitive must have a nonzero tag. The example assigns a tag to each symbol with the statements marked **C** .

Pick aperture and echo: When a pick device is enabled, a square box is displayed, which the end user can move with the mouse, puck, stylus, or cursor keys. This is the pick echo, and it shows the size and position of the **pick aperture**. A primitive is picked only if it passes within this aperture. Setting the

size and initial position of the pick aperture is described in “Initializing logical input devices” on page 211 and succeeding sections.

If the primitive is a string of graphics text, it is picked if part of any character box in the string lies within the aperture. If two or more primitives pass within the aperture, the latest (highest-priority) one is picked. More information on priorities is given in “Drawing chain and segment priority” on page 164.

Segment and segment attributes: A primitive that is to be picked must not only have a nonzero tag, it must also belong to a segment. The segment must have a nonzero identifier, and must be detectable and visible. Detectability and visibility are segment attributes, and are described in “Segment attributes” on page 147. The GSSATI call at **A** makes detectable a current segment attribute. Visible is a default attribute.

The symbols used by the routine in Figure 60 on page 199 all belong to one segment. The segment opened at **B** has the identifier 5. It does not need to be closed.

It is important to remember that segments are collections of primitives, not areas of the screen (see Figure 42 on page 147). To return pick data, the pick aperture must be positioned over a primitive, such as a line, a symbol, or a shaded area. Putting the aperture within a closed object (in the middle of a circle for example) rather than actually over the outline, does not cause the object to be picked, unless it is an area.

Enabling logical input devices: All logical input devices from which your program requires input must be enabled using the GSENAB call. More information is given in “Enabling or disabling a logical input device, using call GSENAB” on page 206.

The routine in Figure 60 on page 199 uses only one type of input device, the pick type. It is enabled by the GSENAB call at **E**, and is disabled when no longer required, at **G**.

Input/output: To make use of GDDM's interactive graphics input facilities, you must send the current page to the terminal using the GSREAD call, as at **F**. More information is given in “Passing input to your program, using call GSREAD” on page 208. You should read that section before writing any programs that use more than one logical input device.

Querying logical input: A program accesses the data from a logical input device by issuing a query call after the GSREAD. This call queries input from a pick device:

```
CALL GSQPIK(SEGID,TAG);
```

The call returns two values: the identifier of the segment to which the picked primitive belongs, and the primitive's tag. If no primitive belonging to a detectable segment passes through the pick aperture, both parameters are set to zero.

The MENU routine queries the pick device at **H**. It is concerned only with the tag because all the primitives belong to the same segment. It returns the tag to its calling routine by means of the variable SYMB_TAG.

A similar call to GSQPIK is GSQPKS. This call returns data for the picked primitive and its segment, and the segment that called **that** segment, and so on, repeated up to and including the top segment in the hierarchy. See the *GDDM Base Application Programming Reference* book for details.

Locator input

A locator logical input device provides the program with the (x,y) screen position, in world coordinates, of the graphics cursor. It can be triggered in the same ways as a pick device.

The call that queries locator input is GSQLOC, which returns one integer and two floating-point values:

```
CALL GSQLOC(INWIN,X,Y);
```

GDDM sets the first parameter to indicate whether the locator was within the graphics window: 1 means it was inside, and 0, outside. (The graphics window, and the associated concept of the viewport, is described in Chapter 7, “Hierarchy of GDDM concepts” on page 107.) By default, the graphics window fills the screen, and so the cursor cannot be moved off the screen. In simple applications, therefore, the locator is always within the window. The second and third parameters are the locator coordinates.

The following code enables a locator, obtains input from it, and draws a symbol at the position it returns. The code includes a call to the routine shown in Figure 60 on page 199 to let the end user select the symbol.

```
DECLARE SYMB_ARRAY(10) CHAR(1)
                INITIAL('A','B','C','D','E','F','G','H','I','J');
DECLARE SYMB_NUM FIXED BINARY(31);
DECLARE (DEV,DEVID,INWIN) FIXED BINARY(31);
DECLARE (X,Y) FLOAT DEC(6);

   /*  .  */
   /*  .  */

CALL MENU(SYMB_NUM);            /* Let end user select a symbol */

   /*  .  */
   /*  .  */

CALL GSENAB(2,1,1);                     /* Enable locator.      */
CALL GSREAD(1,DEV,DEVID);               /* Transmit current page*/
                                        /* and wait for input.  */
CALL GSQLOC(INWIN,X,Y);                 /* Query locator input. */
CALL GSCHAR(X,Y,1,SYMB_ARRAY(SYMB_NUM));/* Draw char at (x,y)   */
```

Instead of the graphics cursor, the locator can be echoed by a *rubber band*, *rubber box*, or a specified segment. More information is given in “Initializing logical input devices” on page 211 and succeeding sections.

Choice input

Choice devices are associated with workstation facilities that can cause interrupts at the host, such as the PF or PA keys, or the mouse or puck buttons or stylus tip-switch.

The data keys can also be enabled as choice devices, in which case any one of them causes an interrupt when pressed (provided a string device has not been enabled).

Enabling data keys for choice input in applications

Data keys are those keys with which users enter data. (These include letters and numerals, brackets, currency signs, the space bar, the cursor keys, and ERASE EOF key.) Their usual function is simply to input a character.

Choice devices have no echo.

If you write application programs for users of GDDM-OS/2 Link or the 3270-PC/G and GX workstations, you can present the end users with a series of alternative actions, each associated with a particular data key. Using the GSENAB call, you can enable data keys on the end user's keyboard as choice devices.

You should avoid using as a choice device any key that may have a special subsystem or GDDM function. This generally means avoiding some or all of the PA keys and possibly the CLEAR key. However, you can use GDDM processing options to control the handling of these keys by the subsystem and GDDM (see the *GDDM Base Application Programming Reference* book). On the 5550, and 3270-PC/G and GX, PA3 cannot be a choice device, because it is not returned to the application. On other devices, PA3 is the default key to enter user control.

Choice input tells your program which key the end user has pressed. GDDM returns the information in two calls, GSREAD and GSQCHO:

```
CALL GSREAD(1,DEV_TYPE,DEV_ID);
CALL GSQCHO(NUMBER);
```

The second parameter of GSREAD is set by GDDM to indicate the type of device, whether choice, locator, pick, string, or stroke.

The following code shows how to use GSREAD and GSQCHO to define functions for three PF keys: PF1 enlarges a previously selected symbol, PF2 reduces it, and PF3 ends the program. There are subroutines to change the size of the symbols, called ENLARGE and REDUCE:

```
DECLARE (DEV_TYPE,DEV_ID) FIXED BINARY(31);
DECLARE PFKEY FIXED BINARY(31);
CALL GSENAB(1,1,1);               /* Enable PF keys as choice   */
                                  /* devices.                   */
CALL GSREAD(1,DEV_TYPE,DEV_ID);   /* Issue graphics read.       */
CALL GSQCHO(PFKEY);               /* Query choice input.        */
IF PFKEY=1                        /* If end user pressed PF1..  */
  THEN CALL ENLARGE;              /* ..perform enlarge function.*/
ELSE IF PFKEY=2                   /* If end user pressed PF2..  */
  THEN CALL REDUCE;               /* ..perform reduce function. */
ELSE IF PFKEY=3                   /* If end user pressed PF3..  */
  THEN GO TO FINISH;              /* go to end of the program.  */
ELSE GO TO PROCESS_ERROR;         /* Only PF1, 2, & 3 accepted. */
```

Multiple-choice devices can be enabled concurrently–the PF keys, the ENTER key, and the data keys.

Effects of stroke and string devices

If a stroke device has been enabled, then enabling the puck, mouse, or stylus as a choice device has no effect. Their use with stroke input overrides their use as a choice device, and they do not return choice data. Similarly, enabling a string device overrides the effects of enabling the data keys as a choice device–they return string, not choice, data.

Choice devices as triggers

The PF keys and the ENTER key can trigger input for all five types of logical input device: choice, locator, pick, string, and stroke. GDDM discards the choice data if the appropriate choice device is not enabled, but still passes on locator, pick, string, and stroke data to the program if these devices are enabled.

The puck, mouse, and stylus behave like the ENTER and PF keys but only when a locator or pick has been enabled and a stroke has not, as a stroke device assigns a special meaning to these keys. Button 4 on the puck and button 3 on the mouse, though, are not available–they never send input to the host.

If the data keys are enabled as a choice device, then they too trigger all enabled devices–when a string device has not been enabled.

The PA and CLEAR keys provide choice data only. They never trigger any other type of input.

Processing choice input from the data keys

When the data keys are enabled as a choice device, pressing any one of them generates an item of choice data. For instance, when the end user presses the A key, the terminal interrupts the host and transmits the letter A to it, which GDDM puts on the input queue.

For an alphanumeric key, the value that the GSQCHO call reads from the input queue derives from the EBCDIC code for the key's character. This is treated as a hexadecimal number. For instance, the EBCDIC code for uppercase A is `X'C1'`, which is equivalent to decimal 193; so the A key returns the value 193.

For nonalphanumeric keys like the cursor keys and ERASE EOF, refer to the *Graphics Control Program Workstation Programmer's Guide and Reference*. This

provides a list of all the keyboard buttons that can provide input, together with the codes they return.

You need to know when you code your program whether it should accept uppercase or lowercase characters, or both, so that you can test for the appropriate codes. If you are expecting input from the numeric data keys, you should remember that the codes are in the range 240 through 249 (corresponding to X'F0' through X'F9') not 0 through 9.

String input

A string device has a similar function to an unprotected alphanumeric field–reading alphanumeric characters typed in by the end user. The characters are displayed on the screen in the same way as for ordinary data entry: they are the string device's echo.

Here is an example of using a string device:

```
DECLARE (DEV_TYPE,DEV_ID) FIXED BIN(31);
DECLARE NAME_IN CHARACTER(8);
DECLARE CURPOS FIXED BINARY(31);

CALL GSSEG(1);
CALL GSCHAR(10,97,12,'< ENTER NAME'); /* End user prompt        */
CALL GSSCLS;

CALL GSENAB(4,1,1);                   /* Enable string device.  */
CALL GSREAD(1,DEV_TYPE,DEV_ID);       /* Send to terminal.      */
CALL GSQSTR(8,NAME_IN,CURPOS);        /* Read string data       */
                                      /* from queue.            */
```

The input is queried by a GSQSTR call:

You can have only one string input area at a time. By default it occupies 8 bytes at the top left of the graphics field. You can specify its length and position, and also any data that it is to display initially, with the GSISTR call (see “Initializing a string device, using calls GSISTR and GSIDVI” on page 213). That section also tells you how to specify the initial cursor position, using call GSIDVI.

As well as entering characters using the data keys, the end user can edit the string with the backspace and left and right cursor keys. The other editing keys, like ERASE EOF, are not available for use on string input.

Enabling end users to draw graphics with the puck, mouse, or stylus

A stroke device is like a locator, but instead of returning one (x,y) position, it returns a series. The end user has two ways of creating the input.

One way is by using the puck, mouse, or stylus to draw a line that is sampled at fixed intervals. This is called **stream** sampling. The movement of the mouse, puck, or stylus is echoed by a continuous line on the screen.

The other way is by indicating the (x,y) locations one at a time by positioning the cursor and then pressing a puck or mouse button, or the stylus tip-switch. This is called **polylocator** sampling. The end user's actions are echoed by either a

polyline or a **polymarker**. The polyline joins all the indicated (x,y) positions. The polymarker echo is a GDDM cross-marker symbol at each indicated position.

You select the sampling method and echo in your program, as explained in "Initializing a stroke device, using call GSISTK" on page 213. The default mode is polyline.

When the program has enabled a stroke device and issued a GSREAD, GDDM places a highlighted X marker on the screen coincident with the graphics cursor, to indicate that a stroke device is available. The end user must then move the cursor to the first (x,y) position that is to be recorded, and **activate** the stroke device. This is done by pressing one of the mouse or puck buttons or the stylus tip-switch. The X marker disappears when the device is activated.

In stream mode, activation initiates sampling at distance-based intervals. If the device is moved less than a minimum distance during a sampling interval, sampling is suspended until the distance moved reaches the minimum. This prevents a large number of equal (x,y) values being returned if the end user stops moving the device. The sampling interval varies with the load on workstation resources. A string device, in particular, may adversely affect the sampling interval.

Pressing a mouse or puck button or stylus switch a second time deactivates the device and suspends stream sampling. Pressing it a third time reactivates the device and restarts sampling, and so on. In this way the end user can draw a set of disconnected lines.

With polylocator sampling, the end user presses the mouse, puck, or stylus switch once for each (x,y) position.

No input can be sent to the host while a stream-mode stroke device is active.

Querying stroke input

You query the stroke data with a GSQSTK call:

```
DECLARE DFLAGS(200) FIXED BINARY(31);
DECLARE (XARRAY(200),YARRAY(200)) FLOAT DECIMAL(6);
DECLARE NUM FIXED BINARY(31);

  /* Max. no. values  Draw flags    Values    Actual no. values */
CALL GSQSTK(200,        DFLAGS,  XARRAY,YARRAY,      NUM );
```

The pairs of x and y values are returned in XARRAY and YARRAY.

The first position in the arrays is not necessarily the same as the initial position of the cursor. The workstation starts recording (x,y) data when the end user activates the device using a mouse or puck button or stylus tip-switch. The end user can move the cursor from its initial position before doing this.

Simple polyline program

The programming example in Figure 61 on page 206 uses a stroke device of the default type, namely polyline.

After reading the stroke input, the program redraws the line created by the end user. Most programs that use stroke input for line drawing need to do this,

because the echo line disappears from the screen when the next terminal I/O occurs. This example in redraws the line in red.

A second GSREAD sends the redrawn line to the workstation. Before this call is executed, stroke input is disabled, and the ENTER key enabled as a choice device. When the end user presses the ENTER key after the line changes to red, the program ends.

```
PLSTK: PROCEDURE OPTIONS(MAIN);

DCL (DEVTYPE,DEVID) FIXED BIN(31);
DCL DFLAGS(64) FIXED BIN(31);
DCL (XARRAY,YARRAY)(64) FLOAT DEC(6);
DCL NUM FIXED BIN(31);

CALL FSINIT;
CALL GSENAB(5,1,1);        /* Enable tablet or mouse for stroke */
CALL GSREAD(1,DEVTYPE,DEVID);             /* Read and wait      */
CALL GSQSTK(64,DFLAGS,XARRAY,YARRAY,NUM); /* Obtain stroke data.*/

/* Now redraw the polyline from the returned arrays of points   */

CALL GSSEG(1);                    /* Begin new segment.         */
CALL GSCOL(2);                    /* Set color to red.          */
CALL GSMOVE(XARRAY(1),YARRAY(1)); /* Make start of line the     */
                                  /* current position.          */
CALL GSPLNE(NUM,XARRAY,YARRAY);   /* Draw the polyline.         */
CALL GSSCLS;
CALL GSENAB(5,1,0);               /* Disable stroke device.     */
CALL GSENAB(1,0,1);               /* Enable enter key as        */
                                  /* choice device.             */
CALL GSREAD(1,DEVTYPE,DEVID);     /* Display polyline in red.   */
CALL FSTERM;

%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END PLSTK;
```

Figure 61. Program using polylocator stroke device

Enabling or disabling a logical input device, using call GSENAB

GSENAB enables a logical input device, thereby requesting GDDM to pass input from that device to your program. If a device is not enabled, GDDM discards all input from it.

You can have any or all of the five basic types of logical input device enabled at any one time. And you can enable as many different types of choice device as you require. However, it makes your program simpler if you enable only those that provide useful data, and let GDDM discard any input from the others.

The way in which GDDM presents input when more than one device is enabled is described in “Passing input to your program, using call GSREAD” on page 208.

A typical GSENAB call is:

```
    /* Device_type   Device_id   Control */
CALL GSENAB(1,           1,         1);    /* Enable PF keys    */
```

The parameters are as follows:

- The first is the type of logical input device being enabled. There are five types:

  1 **Choice device**. Several terminal facilities can be associated with a choice device: the ENTER key, PF keys, PA keys, the CLEAR key, the data keys, and the mouse and puck buttons and stylus tip-switch. One of them is selected using the second parameter.

  2 **Locator device**. This is associated with the mouse, puck, stylus, or cursor keys. The position of the locator is sent to the host when the end user triggers a transmission in one of the ways described in “Choice devices as triggers” on page 203. The program can discover which terminal facility acted as the trigger only if it was enabled as a choice device.

  3 **Pick device**. Physically, this device is the same as the locator, that is, the mouse, puck, stylus, or cursor keys. The difference lies in the information that GDDM passes to your program on input. Instead of x,y coordinates, GDDM identifies the primitive over which the pick device was positioned, and the segment to which that primitive belongs.

  4 **String device**. This device is represented by a string of characters typed by the end user into a program-defined area of the screen. They are mode-1 graphics text characters of default size.

  5 **Stroke device**. This device is similar to the locator: it can be the mouse, puck, or stylus, though not the cursor keys. Instead of a single pair of x,y coordinates, GDDM passes an array of coordinates that trace the path of the cursor as it was moved by the end user.

- If the first parameter does not specify a choice device, the second parameter must be set to:

  1 The only permitted value for devices other than choice devices.

- If the first parameter **does** specify a choice device, the second parameter further identifies it. The valid values are then:

  0 The ENTER key
  1 The PF keys
  4 The PA keys
  5 The CLEAR key
  8 The data keys
  10 The mouse or puck buttons or the stylus tip-switch.

- The last parameter allows you to disable logical input devices, and also to enable them. A value of 0 tells GDDM to disable the device, and 1 to enable it.

Some further advice about using GSENAB is given in “When to issue GSENAB calls” on page 214. You can query whether a logical input device is enabled, as explained in “Querying a logical input device” on page 215.

Passing input to your program, using call GSREAD

A single action by the end user can generate up to five types of input, depending on which logical devices are enabled. For instance, pressing a PF key could create:

- Choice input consisting of a code representing the key.
- Locator input consisting of the position of the cursor.
- Pick input consisting of the identities of the primitive and segment over which the cursor is positioned.
- String input consisting of a character string typed by the end user.
- Stroke input consisting of the preceding track of the cursor.

GDDM presents the input to your program as a queue, with one element, or record, for each enabled type of logical input device. Making the records on the queue available to your program is a second function of the GSREAD call, in addition to its I/O function. This is how it works:

- If the input queue is empty, then, unless you specify otherwise, a GSREAD call sends the data to the terminal, waits for input, and when the input is received, adds one or more records to the input queue. It then removes the top record from the queue, and makes it available for your program to query.
- If the input queue is not empty, a GSREAD call simply removes the next record from the queue and makes it available for querying. It does not do any I/O.

In addition, GSREAD reads any alphanumeric data that the end user may have typed in.

When GSREAD has made a record available, you may inspect it by issuing a query call, namely GSQCHO, GSQLOC, GSQPIK, GSQSTR, or GSQSTK. Your program is in error if the query is not the appropriate one for the currently available record. The order of the records is undefined, so if you have more than one logical input device enabled, it is essential to test the second parameter of GSREAD before issuing a query.

It is important to remember that GSREAD does no I/O unless the queue is empty. In other words, GSREAD does not update the screen while there are any records on the queue. To avoid problems, the recommended technique is to empty the queue immediately after it has been created, as shown in “Handling the input queue” on page 209.

GDDM ensures that a GSREAD call issued when the input queue is empty always results in at least one input record being created. If the end user causes an interrupt that does not create an input record, GDDM rejects it. No input record is created if, for instance, the end user presses the CLEAR key when this has not been enabled as a choice device. In such cases, GDDM sounds the terminal alarm and waits for another interrupt.

Checking for further graphics input records using call GSQSIM

The GSQSIM call tells you whether the queue is empty:

```
CALL GSQSIM(MORE);
```

GDDM sets the parameter to 0 if the queue is empty or 1 if there are more records. A value of 1 therefore means that the next GSREAD does not perform an I/O operation and 0 means that it does (unless the first parameter of the GSREAD is 0, in which case it never performs any I/O).

Handling the input queue

If you are using multiple logical input devices, the order of records in the input queue is undefined. Processing them as they come off the queue may therefore require complex logic. That is why you are recommended to empty the input queue, as shown in Figure 62 before attempting to process any of the data. Furthermore, this technique helps you ensure that the next GSREAD actually updates the screen. As already mentioned, GSREAD does not update the screen if there are records still on the input queue.

```
DECLARE (CHOICE,LOCATOR,PICK,STRING,STROKE,PFKEY,ENTER,MORE)
  FIXED BINARY(31);
DECLARE (DEV_TYPE,DEV_ID) FIXED BINARY(31);
DECLARE (KEY_TYPE,KEY,INWIN,SEGID,TAG,TXTCT,CURPOS,STKCT)
  FIXED BINARY(31);
DECLARE (X,Y) FLOAT DEC(6);
DECLARE (STKX(500),STKY(500)) FLOAT DEC(6);
DECLARE DRFL(500) FIXED BINARY(31);
DECLARE TXT CHAR(100);

CHOICE = 1;                  /* Initialize                    */
LOCATOR = 2;                 /*                               */
PICK = 3;                    /*                               */
STRING = 4;                  /*             mnemonic          */
STROKE = 5;                  /*                               */
PFKEY = 1;                   /*                               */
ENTER = 0;                   /*                     variables */
MORE = 1;                    /*                               */

CALL GSENAB(CHOICE,PFKEY,1);  /* Enable                       */
CALL GSENAB(CHOICE,ENTER,1);  /*                               */
CALL GSENAB(LOCATOR,1,1);     /*                               */
CALL GSENAB(PICK,1,1);        /*            required           */
CALL GSENAB(STRING,1,1);      /*                               */
CALL GSENAB(STROKE,1,1);      /*                       devices */
```

Figure 62 (Part 1 of 2). Emptying the input queue

```
KEY_TYPE,KEY,INWIN,SEGID,TAG,         /* Assign dummy values to */
STKCT,DRFL(1),STKX(1),STKY(1) = 999;  /* variables that may be  */
TXT = '999';                          /* set when input queried.*/

GET_RECORD:                           /* Create input queue and */
CALL GSREAD(1,DEV_TYPE,DEV_ID);       /* remove records from it.*/

IF DEV_TYPE=CHOICE            /* Next record is of choice type  */
  THEN DO;                    /*                                */
  KEY_TYPE = DEV_ID;          /* Store type of key code.        */
  CALL GSQCHO(KEY);           /* Which key did end user press ? */
END;                          /* Choice type.                   */

IF DEV_TYPE=LOCATOR           /* Next record is of locator type.*/
  THEN CALL GSQLOC(INWIN,X,Y);/* Query & store locator position.*/

IF DEV_TYPE=PICK              /* Next record is of pick type.   */
  THEN CALL GSQPIK(SEGID,TAG);/* Store segment id. and type.    */

IF DEV_TYPE=STRING            /* Next record is of string type. */
  THEN CALL GSQSTR(TXTCT,TXT,CURPOS);
                              /* Store length and text.         */

IF DEV_TYPE=STROKE            /* Next record is of stroke type. */
  THEN CALL GSQSTK(500,DRFL,STKX,STKY,STKCT);
                              /* Store arrays of draw flags &   */
                              /* x,y pairs, & count of x,y pairs*/

CALL GSQSIM(MORE);            /* Any ,ore elements on the queue?*/
IF MORE=1                     /* Go back to read the next record*/
  THEN GO TO GET_RECORD;      /* if the queue is not yet empty  */

/*****************************************************************/
/* Now process data in KEY_TYPE, KEY, INWIN, X, Y, SEGID, TAG,  */
/* TXTCT, TXT, STKCT, DRFL, STKX, AND STKY.                      */
/* Value of 999 means no data received from corresponding device*/
/*****************************************************************/
```

Figure 62 (Part 2 of 2). Emptying the input queue

Using ASREAD instead of GSREAD

You can use the FSENAB call to enable ASREAD for graphics input. When enabled for graphics input, ASREAD sends the current page to the terminal, waits for input, and when the input is received, adds one or more records to the input queue. Unlike GSREAD, it performs the above I/O even if there are records on the input queue, and does not remove the top record from the queue. You can use a GSREAD call with a value of 0 in the first parameter to remove records from the queue.

Initializing logical input devices

Initializing a logical input device means defining its characteristics. For a locator, for instance, you can specify its echo type and its initial position on the screen, and for a pick device, the pick aperture. There are no variable characteristics of choice devices, so these cannot be initialized.

The initialization values are taken from three sources:

- The parameters of optional initialization calls, namely GSILOC for the locator, GSIPIK for the pick device, GSISTR for the string device, and GSISTK for the stroke device.
- Other optional calls, namely GSIDVF (initial data value, float) and GSIDVI (initial data value, integer). These calls are used to specify some less frequently used initialization parameters.
- GDDM-defined defaults.

The complete set of characteristics is determined from these three sources when a device is enabled. All the required initialization and data-record calls must therefore be issued before the GSENAB call. Your program is in error if it issues any of them for an enabled device.

You can reinitialize a logical input device at any time by disabling it and then reenabling it.

You can issue as many initialization calls (GSILOC, GSIPIK, GSISTR, GSISTK, GSIDVF, and GSIDVI) as you choose while the device is not enabled. This means that you can specify a characteristic and later delete or respecify it, if the device has not yet been enabled, or has been enabled and is now disabled again. Information about undoing the effects of the initialization calls is given in their descriptions in *GDDM Base Application Programming Reference* book.

Initializing a locator device, using call GSILOC

If you specify 2 as the first parameter of the GSENAB call, you can then initialize the locator device and define its characteristics using the GSILOC call.

Specifying locator-echo type and initial position, using call GSILOC

The GSILOC call has the following form:

```
                /* Echo-type  Initial position */
CALL GSILOC( 1,      0,        20.0 , 30.0);
```

The first parameter must always be 1.

The last two parameters specify the required initial position of the locator in world coordinates. The default, applied if no GSILOC call is issued, is the center of the screen. If the locator device is a puck or stylus, the echo jumps to the puck or stylus position immediately, unless it is out of contact with the tablet. With these devices, therefore, initial positioning may be of no value.

The second parameter specifies the echo-type, the visual form of the locator device.

Initializing a rubber-band locator

This call specifies a rubber-band echo with the movable end initially positioned at the initial locator position of (20,30):

```
CALL GSILOC(1,4,20.0,30.0);
```

and these fix the other end at (50,0):

```
CALL GSIDVF(2,1,1,50.0);
CALL GSIDVF(2,1,2,0.0);
```

Initializing a rubber-box locator

This call specifies a rubber-box echo with the movable corner initially positioned at the initial locator position of (20,30):

```
CALL GSILOC(1,5,20.0,30.0);
```

and these calls fix the opposite corner at (10,20):

```
CALL GSIDVF(2,1,1,10.0);
CALL GSIDVF(2,1,2,20.0);
```

Initializing a segment locator

This call specifies that a segment is to be used as the echo, initially positioned at (20,30):

```
CALL GSILOC(1,6,20.0,30.0);
```

and this call specifies that it is to be segment 5:

```
CALL GSIDVI(2,1,1,5);
```

Initializing a segment-transform locator for applications running on GDDM-OS/2 Link

Using the GSILOC call, you can enable end users of your applications running on GDDM-OS/2 Link to perform transforms of graphics segments interactively and have their changes echoed. No other GDDM devices enable users to see the transforms as they are making them.

In addition to the logical devices described above, you can now initialize the following local segment transforms as locator echoes:

- Segment scaling
- Segment rotation
- Segment shearing

The third and fourth parameters of GSILOC, which are used to define the initial position of the locator echo, are ignored under GDDM-OS/2 Link and the initial position defaults to the center of the screen.

If you initialize a locator echo that requires a reference point, you must set the fixed point using the GSIDVF call described in the *GDDM Base Application Programming Reference* book. You can enable end users to select the reference point and pass the queried cursor position to the GSIDVF call.

Initializing a pick device, using calls GSIPIK and GSIDVF

The only initial values you can specify for a pick device are its initial position and the size of the pick aperture. The pick echo is always the aperture square.

Specifying initial position of a pick device

You can specify a primitive over which GDDM is to initially position the pick as follows:

```
CALL GSIPIK(1,0,SEGID,TAG);
```

If no GSIPIK is issued, or if the segment identifier or tag is zero, or if the segment is invisible or nondetectable, the pick is placed at the default initial position, which is the center of the screen.

Setting the pick aperture

You can set the pick aperture using the GSIDVF call:

```
CALL GSIDVF(3,1,1,1.6); /*Make aperture 1.6 times default size*/
```

The value of 3 in the first parameter indicates that the call refers to the initial data record for the pick device. The second parameter must be 1, and, when the first parameter has a value of 3, so must the third. The fourth parameter is the size of the pick aperture as a ratio to the default, which is a square equal in dimensions to the height of the default character box.

Initializing a string device, using calls GSISTR and GSIDVI

You can specify the size and position of the string input area, and supply initial data, and make the area invisible, with the GSISTR call:

```
 /* Device-id  Echo  Position  Size         Initial text      */
CALL GSISTR(1,  1,   0.0,25.0,  30, 'OVERTYPE THIS WITH YOUR INPUT');
```

If the string device is not initialized, it consists of eight characters in the top left of the graphics area, initialized to nulls, with a visible echo of the text typed by the end user.

You can use the call GSIDVI to specify the field position under which the cursor is to be placed in the string input area:

```
    /* Device-type  Device-id  Element-no  Integer-value */
CALL GSIDVI( 4,        1,          1,           4);
```

The first parameter must be 4 for a string device. The second parameter is always 1. The third parameter can be 1 or 0. A value of 1 specifies that the value in the fourth parameter is the field position of the cursor. A value of 0 in the fourth parameter is treated as a 1. A value of 0 in the third parameter specifies that any field position previously set by element-number 1 should be reset to 0.

Initializing a stroke device, using call GSISTK

If you initialize a stroke device, you can specify its mode, the maximum number of points to be returned, and the initial position of the cursor. Here is a typical call:

```
          /* Echo  Sampling  Initial position  Number of points */
CALL GSISTK(1, 1,     2,        20.0,10.0,           800 );
```

The first parameter must always be 1. The second parameter defines the echo type and the third the sampling method. Together they define the stroke device's mode of operation.

The fourth and fifth parameters are the initial position for the cursor, in world-coordinate units. If the locator device is a puck or stylus, the echo jumps to the puck or stylus position immediately, unless it is out of contact with the tablet. With these devices, therefore, the main effect of the initial position parameters is to determine where the initial X marker is placed.

Using a locator, pick, and stroke device together

You can enable a locator, a pick, and a stroke device, or any two of them, in separate GSENAB calls. However, there is no means of displaying and moving them. The box representing the pick aperture is superimposed on the locator echo, and the locator echo shows the current position of the stroke device, except when the stroke device is active in stream mode.

In stream mode, the locator echo and pick aperture box do not move while the movement of the mouse, puck, or stylus is echoed by a line being drawn on the screen. They remain stationary, at the start of the line. When the stroke device is deactivated, the locator echo and pick box jump to the end of the line, and then follow the movements of the mouse, puck, or stylus.

When a stroke device is enabled, the pick and locator data returned to your program are determined by their position when the trigger key (PF key, ENTER key, or data key) was pressed. For the locator, the input data may or may not be the same as the last pair of stroke values. It depends on whether the user moved the locator from the final stroke position before pressing the trigger key. The pick data comprises the identifiers of the highest-priority primitive and segment within the pick aperture centered on the locator position.

The specified or defaulted initial position of the stroke device overrides that of the locator device, if different, which in turn overrides that of the pick device.

To obtain the maximum sampling rate, and hence record the finest detail, it is advisable to disable all other logical input devices when a stream mode input device is in use.

When to issue GSENAB calls

You should consider carefully where in your program to issue GSENAB calls. It is often simplest to enable the required devices immediately before a GSREAD and disable them immediately after it. You should bear in mind these points:

- All initialization calls for a device must precede the GSENAB.
- The enabled devices must be associated with the graphics field that is about to be sent to the terminal (see “The graphics field and the image field” on page 112). Each page on which graphics is used has one graphics field (often created by default), and each graphics field has its own set of logical input devices. If an existing graphics field is explicitly redefined, or if a new page is created, the new graphics field has no enabled input devices.
- In some circumstances it is bad practice to disable the locator after a GSREAD. This is because on the next GSREAD the echo is displayed in its specified or

default initial position, whereas the application may be easier to use if the echo remains where the end user put it.

Querying a logical input device

You can query a logical device using the GSQLID call. GDDM indicates whether the device is enabled, what the current echo type is, and what other types of echo are valid. Here is an example:

```
DECLARE LIDLIST(3) FIXED BINARY(31);
  /* Device-type   Device-id   Count    List  */
GSQLID(  2,            1,        3,   LIDLIST );
```

The first parameter is the type of logical input device being queried. The possible values and their meanings are the same as for the first parameter of GSENAB (see “Enabling or disabling a logical input device, using call GSENAB” on page 206). The example specifies type 2, meaning a locator device.

The second parameter is the device identifier, using the same values as the second parameter of GSENAB. For all device types except choice, this must be 1.

GDDM returns the information in the last parameter, which is an array. The third parameter specifies how many elements are to be returned. The maximum is three, and their values and meanings are as follows:

- Whether the specified device is enabled:

  `1` Enabled.
  `0` Not enabled.
  `-1` The current primary device (the terminal) does not support this type of logical input device.

- The current echo type, using the same numbers as in the initialization calls.
- The highest numbered echo type that is supported, again using the same numbers as the initialization calls. All echo types with a lower or equal number are supported. If the specified logical input device is not supported, –1 is returned. If the specified logical input device is supported but has no echo (as is the case with choice devices), 0 is returned.

Segment-picking example

The program in Figure 60 on page 199 used vector symbols to draw the pictures on the screen. Each symbol was a graphics primitive, and the routine returned the tag of the picked primitive.

Many applications require the end user to pick segments rather than primitives. The program in Figure 63 on page 216 illustrates this. It draws several squares using GSLINE calls, each square being a separate segment. The end user can then select and delete any square.

To ensure that the squares really are square, the program executes a GSUWIN call at **A**. At **B** the detectable attribute is set on, to enable the end user to pick the squares. The subroutine DRAW_SQUARE called at **C** draws the squares. Each invocation draws one square.

A segment for each square is opened at **F**, and closed at **H**. Because there is one square per segment, the segment identifiers uniquely identify the squares. To be detectable, all primitives must be tagged. However, in this example there is no need to identify the individual primitives, so they are all given the same tag, 1, at **G**.

The pick device is enabled at **D**. The end user has to use a choice-type key, such as a mouse button, to send the pick input to the host. However, choice data is not required by the program, so no choice devices are enabled.

The program deletes the returned segment at **E** with a GSSDEL call, thereby removing the selected square from the current page. Control then returns to the top of the DO UNTIL loop for another GSREAD. This updates the display and waits for the next input.

The program ends when the end user causes an interrupt without positioning the pick over a primitive. GDDM sets both parameters of GSREAD to zero in this case. The first parameter, SEL, controls the DO UNTIL loop. When it is zero, looping stops and the program ends.

```
DELSQ: PROCEDURE OPTIONS (MAIN);

DCL NAMES(1) CHAR(8);                 /* Device names.          */
DCL (SEL,SEG,DEVICE_TYPE,DEVICE_ID,TAG) FIXED BIN(31);
DCL (X,Y) FLOAT DEC(6);               /* Temporary variables    */

CALL FSINIT;                          /* Initialize GDDM        */

CALL GSUWIN(0.0,100.0,0.0,100.0);     /* Ensure correct aspect  */ A
                                       /* ratio.                 */
SEG=1;                                /* Initialize segment id. */
CALL GSSATI(1,1);                     /* Make squares detectable*/ B
DO X=1 TO 51 BY 10;                   /* Draw an array of 36    */
  DO Y=1 TO 51 BY 10;                 /* squares. Each is in    */
    CALL DRAW_SQUARE;                 /* its own segment.       */ C
    SEG=SEG+1;                        /* Increment segment id   */
  END;                                /* Y loop                 */
END;                                  /* X loop                 */

CALL GSENAB(3,1,1);                   /* Enable a pick device.  */ D
DO UNTIL (SEL=0);                     /* Update the screen and  */
  CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID); /*accept selections     */
                                      /* until a null selection */
                                      /* is made. This happens  */
                                      /* when a key is pressed  */
                                      /* but the pick is not    */
                                      /* over any line in a     */
                                      /* square.                */
```

Figure 63 (Part 1 of 2). Segment-picking example

```
  CALL GSQPIK(SEL,TAG);              /* Get the segment id and */
                                     /* tag. SEL is zero for a */
                                     /* null selection.        */
                                     /* Because all parts of   */
                                     /* squares were drawn     */
                                     /* with same tag, tag     */
                                     /* can be ignored.        */
  IF SEL¬=0 THEN CALL GSSDEL(SEL);   /* Delete the selected    */ E
                                     /* segment.               */
END;

CALL FSTERM;                         /* Finished with GDDM     */

DRAW_SQUARE: PROCEDURE;
    CALL GSSEG(SEG);                 /* Create segment.        */ F
    CALL GSTAG(1);                   /* Must have non-zero     */ G
                                     /* tag to permit detect-  */
                                     /* ability. All lines     */
                                     /* have the same tag.     */
    CALL GSMOVE(X,Y);                /* Starting point.        */
    CALL GSLINE(X+8.0,Y);            /* Draw                   */
    CALL GSLINE(X+8.0,Y+8.0);        /*      sides             */
    CALL GSLINE(X,Y+8.0);            /*           of           */
    CALL GSLINE(X,Y);                /*             square.    */
    CALL GSSCLS;                     /* Close segment.         */ H
END DRAW_SQUARE;

%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

END DELSQ;
```

Figure 63 (Part 2 of 2). Segment-picking example

Simple free-hand drawing program

The program in Figure 64 on page 218 enables the end user to draw on the screen. It captures original drawings, for which the end user would typically use a tablet and stylus. It can equally well be used for digitizing existing drawings, which the end user would typically trace over using a tablet and puck.

After initializing and enabling a stream-mode stroke device, the program executes three GSREAD calls in a loop. Each read enables the end user to record the maximum number of points that GDDM allows, namely 1024. If the end user does not move the cursor between GSREADs, one continuous line can be drawn. Alternatively, one or more lines can be drawn for each GSREAD, depending on the use made of the mouse or puck buttons or stylus tip switch.

After each read, the program queries the stroke data and redraws the line or lines just created by the end user. If the program did not do this, the lines would disappear from the screen at the next GSREAD.

After the third GSREAD, the program disables the stroke device and enables the ENTER key as a choice device. A fourth GSREAD is executed to display the latest redrawn lines. When the end user then presses the ENTER key, the program ends.

```
STROKE2: PROCEDURE OPTIONS(MAIN);

DCL  DFLAGS(1024) FIXED BIN(31);
DCL  (XARRAY,YARRAY)(1024) FLOAT DEC(6);
DCL  (DEVTYPE,DEVID) FIXED BIN(31);
DCL  NUM FIXED BIN(31);

CALL FSINIT;
           /*           Initialize stroke device:-              */
           /* Stream mode   Initial position   Max. no. points  */
CALL GSISTK(1,    1,2,          0.0,0.0,             1024);

CALL GSENAB(5,1,1);                    /* Enable stroke device  */

CALL GSSEG(1);                         /* Open a segment        */

DO I= 1 TO 3;

  CALL GSREAD(1,DEVTYPE,DEVID);
  CALL GSQSTK(1024,DFLAGS,XARRAY,YARRAY,NUM);

  /* Preserve the polyline image by drawing it from the         */
  /* returned arrays of x,y pairs                               */
  DO J=1 TO NUM;
    IF DFLAGS(J)=1 THEN
      CALL GSMOVE(XARRAY(J),YARRAY(J));
    IF DFLAGS(J)=0 THEN
      CALL GSLINE(XARRAY(J),YARRAY(J));
  END;
END;

CALL GSSCLS;                           /* Close the segment.    */

CALL GSENAB(5,1,0);                    /* Disable stroke device.*/
CALL GSENAB(1,0,1);                    /* Enable enter key.     */
CALL GSREAD(1,DEVTYPE,DEVID);

CALL FSTERM;

%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

 END STROKE2;
```

Figure 64. Program for freehand drawing on the screen

Dragging segments

If a segment is to be repositioned, you can often help the end user by enabling a copy of it to be dragged around the screen before its final position is determined. You do this by making the segment the locator echo. The program in Figure 65 shows you how.

The locator is initialized at **A** with a type 6 echo, that is, a segment. The segment is a square.

The last parameter of the GSIDVI call, at **B**, specifies that the segment containing the square, namely segment 9, is the one to be used as the echo.

The GSIDVF calls at **C** and **D** offset the echo from the locator position by 0.2 window units in the x and y directions. The reason is explained in “How the 3270-PC/G and GX draw echoes” on page 220.

The GSREAD, **E**, displays the square onto the screen. The end user can then drag a copy of it around with the mouse, puck, stylus, or cursor keys.

The GSREAD waits for an interrupt from the terminal. Any locator-trigger key can be used to send this interrupt; a mouse or puck button or the stylus tip-switch is probably the most convenient. When the interrupt is received, execution of the program resumes. The position of the locator is queried and a GSSPOS issued to move the segment to that position.

This example shows a very simple case. In less straightforward cases, you may get unexpected results if you do not pay particular attention to the segment origin. More information is given in “Local origin when dragging a segment” on page 221.

```
DRAG1: PROCEDURE OPTIONS (MAIN);
DCL (DEVICE_ID,DEVICE_TYPE) FIXED BIN(31);
DCL INWIN FIXED BIN(31);
DCL (X,Y) FLOAT DEC (6);
DCL (YES,NO,STOP) FIXED BIN(15);       /* Flags                 */
YES=1; NO=0;                           /* Values for flags      */

CALL FSINIT;                           /* Initialize GDDM       */

CALL GSUWIN(0.0,100.0,0.0,100.0);      /* Uniform window coords.*/

CALL GSSATI(4,2);                      /* Make transformable a  */
                                       /* current segment attr. */
                                       /* so square can be moved*/
```

Figure 65 (Part 1 of 2). Program for dragging segments

```
CALL GSSEG(9);                          /* Open numbered segment */
CALL GSMOVE(0.0,0.0);                   /* Start square          */
CALL GSLINE(10.0,0.0);                  /* Draw                  */
CALL GSLINE(10.0,10.0);                 /*      sides            */
CALL GSLINE(0.0,10.0);                  /*           of          */
CALL GSLINE(0.0,0.0);                   /*             square    */
CALL GSSCLS;                            /* Close segment         */

CALL GSILOC(1,6,0.0,0.0);               /* Set up locator ( eg.  */ A
CALL GSIDVI(2,1,1,9);                   /* a mouse) to drag      */ B
                                        /* segment 9.            */
CALL GSIDVF(2,1,1,0.2);                 /* Offset echo from      */ C
CALL GSIDVF(2,1,2,0.2);                 /* original segment      */ D
CALL GSENAB(2,1,1);                     /* Enable the locator.   */

STOP=NO;                                /* Initialize flag       */
DO WHILE (STOP=NO);
  CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID);/* Display the square    */ E
                                        /* Square moves as the   */
                                        /* cursor is moved.      */
                                        /* When trigger key is   */
                                        /* pressed, control      */
                                        /* returns to program.   */
  CALL GSQLOC(INWIN,X,Y);               /* Get the location      */
  CALL GSSPOS(9,X,Y);                   /* Move the square       */
  IF X < 10.0 THEN STOP=YES;            /* Stop when locator near*/
                                        /* l.hand edge of screen */
END;
CALL FSTERM;                            /* Terminate GDDM        */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END DRAG1;
```

Figure 65 (Part 2 of 2). Program for dragging segments

If you want to return to the default cursor as the locator echo after dragging a segment, you must disable and then reinitialize the locator before reenabling it:

```
CALL GSENAB(2,1,0);         /* Disable the locator             */
CALL GSILOC(1,2,X,Y);       /* Reinitialize with cursor as echo */
CALL GSENAB(2,1,1);         /* Reenable                        */
```

How the 3270-PC/G and GX draw echoes

Many echoes are drawn by the workstation in **exclusive-OR mode**. The effect of this is that if a primitive in the echo overlaps another primitive on the screen, both may become invisible or change color where they intersect. This is also true for segments.

There are several ways of preventing the segment and echo initially being invisible. The simplest is shown in the program in Figure 65.

The GSIDVF calls, **C** and **D**, slightly offset the echo from the original segment. They set the position of the segment echo to 0.2 world coordinates from the current locator position in both directions–just enough to ensure that it uses adjacent pixels to the original segment. This prevents the echo from exactly coinciding with the original segment, both initially and following a segment move.

Another method is to make the original segment invisible using a GSSATS call. The echo does not inherit the invisible attribute. There is then only one copy of the segment on the screen – the echo. Changing a segment's visibility attribute from visible to invisible may cause some or all of the screen to be redrawn, which may be a disadvantage.

Local origin when dragging a segment

In Figure 65, the segment's origin is at the origin of the current world coordinate system. It is located at an obvious place within the segment, namely, the bottom left-hand corner.

Such simple conditions do not usually apply in a real application program. For a more typical situation, consider the following code, which amends the code in Figure 65 from the point where the segment is opened.

```
CALL GSSEG(9);                      /* Open numbered segment */  J
CALL GSMOVE(20.0,20.0);             /* Start square          */
CALL GSLINE(30.0,20.0);             /* Draw                  */
CALL GSLINE(30.0,30.0);             /*      sides            */
CALL GSLINE(20.0,30.0);             /*           of          */
CALL GSLINE(20.0,20.0);             /*             square    */
CALL GSSCLS;                        /* Close segment         */  K

CALL GSSPOS(9,40.0,40.0);           /* Move segment          */  L

CALL GSENAB(2,1,1);                 /* Enable default locator*/  M
CALL GSSEG(10);                     /* Display instructions  */
CALL GSCHAR(0.0,0.0,24,'INDICATE REFERENCE POINT');
CALL GSSCLS;
CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID);/* Display the square   */
CALL GSSDEL(10);                    /* Delete instructions   */
CALL GSENAB(2,1,0);                 /* Disable default loc'r */
CALL GSQLOC(INWIN,X,Y);             /* Get indicated ref PT  */  N

CALL GSSORG(9,X,Y);                 /* Move segment origin   */  O
CALL GSILOC(1,6,X,Y);               /* Set up locator ..     */  P
CALL GSIDVI(2,1,1,9);               /*  .. to drag segment 9 */
CALL GSIDVF(2,1,1,0.2);             /* Offset ..             */
CALL GSIDVF(2,1,2,0.2);             /*  .. echo              */
CALL GSENAB(2,1,1);                 /* Enable the locator.   */
```

Figure 66. Defining a local origin for dragging

The following changes have been made to the way the square is drawn:

1. The statements between **J** and **K** now draw the square with its bottom left-hand corner at (20,20) instead of (0,0).
2. The statement **L** moves the segment so that its origin is at (40,40).

The most obvious result of these changes is that the original segment is displayed with its bottom left-hand corner at (60,60) instead of (0,0), as illustrated in Figure 67.

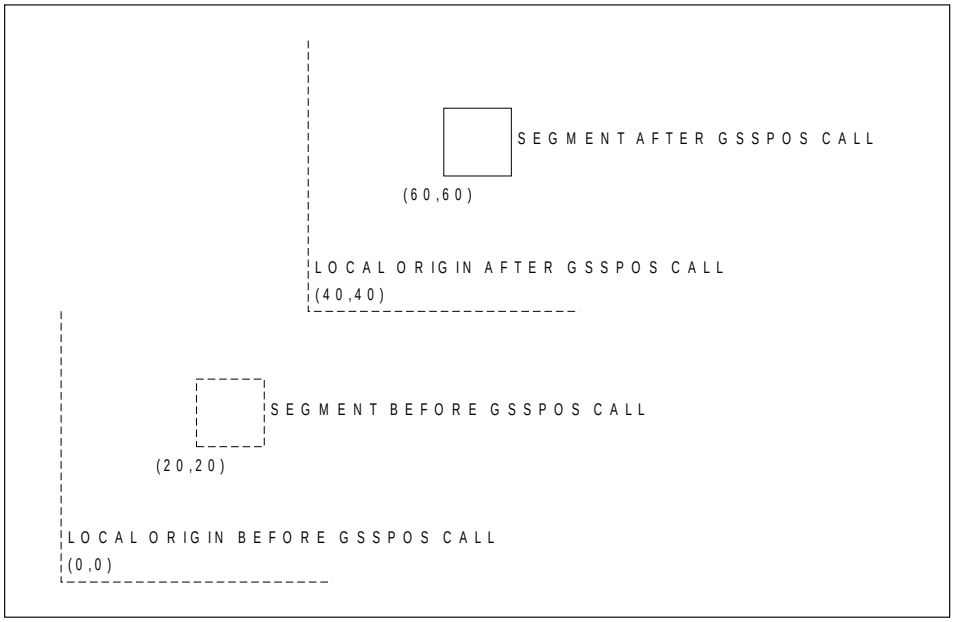


Figure 67. Local origin of echo segment

Less obviously, the first change would prevent the operator dragging the square any nearer the bottom or left-hand edge of the screen than 20 world-coordinate units. This is because the origin cannot be dragged off the screen, and the origin is 20 units leftward and downward from the bottom left-hand corner of the square. The operator can still end the program because the GSQLOC returns the position of the origin, not the bottom left-hand corner of the square.

The second change would cause the echo to initially appear 40 units leftward and downward from the original segment. This is because the GSSPOS call puts the segment's origin at (40,40), whereas the echo's origin is initially placed at (0,0). This is because the GSILOC call in the example specifies (0,0) as the initial position for the echo. When the echo is a segment, it is the segment origin that is put at the specified initial position.

These pitfalls can be avoided by defining a **reference point** within the segment. This is, conceptually, the point at which the dragging mechanism is attached to the segment. Often it is best to allow the end user to select a reference point before dragging or transforming a segment. The statements between the points marked **M** and **N** do this.

Then, to avoid the first pitfall, you should make the reference point into the segment origin using a GSSORG call. This is done at **O**. And to avoid the second, you should specify the reference point as the initial position of the locator. This is done at **P**.

If the operator has to pick the segment before it is dragged, it may help to enable a locator as well as the pick. The (x,y) position of the combined pick/locator when the operator makes the selection can then be used as the reference point for dragging. Here is an example:

```
DCL (INWIN,DEVICE_ID,DEVICE_TYPE,MORE,SEG,TAG) FIXED BIN(31);
DCL (X,Y) FLOAT DEC (6);

/*                      CREATE THE SEGMENTS                     */
/*                               .                              */
/*                               .                              */

CALL GSENAB(2,1,1);                    /* Enable default locator*/
CALL GSENAB(3,1,1);                    /* Enable pick           */

REREAD:
CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID);  /* Read pick & locator   */

IF DEVICE_TYPE = 2                     /* If next input is loc'r*/
  THEN CALL GSQLOC(INWIN,X,Y);         /* get indicated ref PT  */
IF DEVICE_TYPE = 3                     /* If next input is pick */
  THEN CALL GSQPIK(SEG,TAG);           /* get segment id & tag  */
CALL GSQSIM(MORE);                     /* Another input record? */
IF MORE=1 THEN GO TO REREAD;           /* If yes, read it       */
IF SEG=0 THEN GO TO REREAD;            /* No segment picked     */

CALL GSENAB(2,1,0);                    /* Disable default loc'r */
CALL GSENAB(3,1,0);                    /* Disable pick          */

CALL GSSORG(SEG,X,Y);                  /* Move segment origin   */
CALL GSILOC(1,6,X,Y);                  /* Set up locator ..     */
CALL GSIDVI(2,1,1,SEG);                /* .. to drag picked seg */
CALL GSIDVF(2,1,1,0.2);                /* Offset ..             */
CALL GSIDVF(2,1,2,0.2);                /* .. echo               */
CALL GSENAB(2,1,1);                    /* Enable the locator.   */
CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID);  /* Op. can now drag seg  */
CALL GSQLOC(INWIN,X,Y);                /* Get the new location  */
```

Figure 68. Defining a reference point for segment dragging

Local origin when transforming a segment

Any transformation involves a reference point. It is the point about which rotation, scaling, or shearing takes place, or the one that is displaced to a specified new position.

In a simple shape there may be an obvious location for it–the center of a circle or a corner of a polygon, for instance. But in general, there is no obvious point definable by a program. So to ensure that the results on the screen are as required, an application can ask the operator to indicate the reference point. The method would be similar to the one described in “Local origin when dragging a segment” on page 221.

The transformation calls (GSSAGA, GSSTFM, and GSSPOS) treat the origin of the segment as the reference point. Before executing a transformation call, therefore, the program can execute a GSSORG call to move the segment origin to the point indicated by the operator.

The next example shows how to perform the technique for a rotation.

```
DECLARE (X1,X2,Y1,Y2) FLOAT DEC(6);
DECLARE (INWIN,DEVICE_TYPE,DEVICE_ID) FIXED BINARY(31);

/*                CREATE SEGMENT 99                            */
/*                        .                                    */
/*                        .                                    */


CALL GSENAB(2,1,1);                   /* Enable default cursor  */

CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID); /* Read reference point   */
CALL GSQLOC(INWIN,X1,Y1);             /* Get location           */

CALL GSSORG(99,X1,Y1);                /* Move local origin      */

CALL GSENAB(2,1,0);                   /* Disable default cursor */
CALL GSILOC(1,4,X1,Y1);               /* Initialize rubber band */
CALL GSENAB(2,1,1);                   /* Enable rubber band     */

CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID); /* Read location for angle*/
CALL GSQLOC(INWIN,X2,Y2);             /* Get location           */

CALL GSSAGA(99,1.0,1.0,0.0,1.0,X2-X1,Y2-Y1,0.0,0.0,0);
                                      /* Rotate segment         */
CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID); /* Redisplay segment      */
```

Panning and zooming

Panning applies to pictures that are bigger than the screen. It means changing the section of the picture that is displayed–in effect, treating the screen like a window and moving it up and down and from side to side. Scrolling is another term for it.

Zooming means displaying more or less of the picture by shrinking or enlarging the graphics–in effect moving the window closer to or further from the picture.

This guide describes three methods of panning and zooming. You should refer to the indicated sections for more information.

- Setting new window coordinates with a GSWIN or GSUWIN call, and redrawing the picture (see “Uniform world coordinates” on page 118).
- Saving the picture with a GSSAVE, altering the window with a GSWIN or GSUWIN, and restoring the picture with a GSLOAD (see “Panning and zooming” on page 189).
- Allowing the operator to use the user-control facility for panning and zooming.

Retained and nonretained modes on the 3270-PC/G and GX

In the normal mode of operation, GDDM sends graphic orders to the 3270-PC/G and GX workstation with the request that they be retained in its segment storage. GDDM then sends further instructions to execute these orders and thereby display the picture. This is called the retained mode of operation.

When storage in the workstation is limited or the picture is complicated, the graphics orders for the whole picture cannot be retained. In these cases, GDDM sends the graphics orders to the workstation with a request for immediate execution, and hence immediate display of the picture. The workstation retains no graphics orders. This is called nonretained mode.

If the program amends nonretained graphics, GDDM may have to retransmit the whole picture, whereas in retained mode it could transmit just the updates. Using nonretained mode, therefore, results in longer data streams. Furthermore, some workstation functions that require retained graphics are wholly or partially unavailable in nonretained mode.

Retained mode is the default. GDDM degrades from retained to nonretained mode if necessary, with no action by you. Or, instead, you can specify nonretained mode in a processing option. The terminal can also be configured as “output only”, which has the same effect as the nonretained-mode processing option.

Switching modes may cause a redraw of the screen. If the pictures are such that GDDM would switch frequently, you may improve usability by specifying nonretained mode. And applications that create complex pictures and have little graphics-input function may be most efficient in nonretained mode.

You can specify nonretained mode with a processing option, either on a DSOPEN call:

```
DECLARE NAME(1) CHARACTER(8);
DECLARE PROCOPT_LIST(2) FIXED BINARY(31);

PROCOPT_LIST(1) = 17;    /* RETAINED/NONRETAINED MODE PROC. OPTION    */
PROCOPT_LIST(2) = 1;     /* 0 = RETAINED (DEFAULT); 1 = NONRETAINED   */

CALL DSOPEN(1,1,'*',   2,PROCOPT_LIST,   0,NAME);
```

or in a nickname statement:

```
         ADMMNICK FAM=1,PROCOPT=((SEGSTORE,NO))
```

The nickname method is usually preferable, because it enables the end user to select the mode to suit the terminal being used.

Query primitives and segments in specified area using call GSCORR

A pick logical input device returns the tag and segment identifier of a primitive selected by the end user. The GSCORR call performs a similar correlation function without using a pick device. Your program specifies a rectangular area, and GDDM returns the tags of all the primitives completely or partly contained within it, together with the identifiers of their segments.

Here is a typical call:

```
DECLARE SIZE(1) FLOAT DEC(6);
DECLARE SEGS(10) FLOAT DEC(6);
DECLARE TAGS(10) FLOAT DEC(6);
DECLARE NUMHITS FIXED BINARY (31);

SIZE(1) = 5;

   /* CORR-TYPE  POSITION  SIZE-TYPE  SIZE     HITS       NUMBER */
CALL GSCORR(1,  30.0,40.0,    1,    1,SIZE, 10,SEGS,TAGS, NUMHITS);
```

The parameters of this call are explained in the *GDDM Base Application Programming Reference* book.

Correlation with GSCORR differs from selection with a pick device in several ways:

- GSCORR does not require action by the end user. It is usually used in an interactive context, but it need not be.
- GSCORR returns all the hits within the specified area. A pick device returns only the one with the highest priority.
- A pick device correlates only visible and detectable segments. If the first parameter of GSCORR is 0, it does the same, but if the first parameter is 1, it correlates all types of segment.
- Correlation can be done without altering the pick device. If, for instance, the application uses the pick for menu selection, this function can be retained while correlation with GSCORR is being done.

The program in Figure 69 on page 227 shows how to use GSCORR in an interactive context.

It displays an array of crosses. The end user indicates the size and position of the correlation area using two pointings with the locator. The first pointing is with the default cursor. For the second pointing a rubber box is provided. After the second pointing, all crosses within the rubber box are made invisible.

Further pairs of pointings can be made at the operator's choice. The program ends when the operator indicates an area of zero width or depth.

The crosses are drawn in the loop at **A**. Each has its own segment, opened at **B**. The default cursor is enabled for the first time at **C**.

The position of the fixed corner of the rubber box is read at **D**. At **E**, the default cursor is disabled so that the rubber box can be initialized, at **F**, and enabled, at **G**. One corner of the box is fixed at the position indicated by the locator input (X1,Y1). The movable corner is attached to the locator. When the second locator input (X2,Y2) is obtained at **H**, the area enclosed by the rubber box is made the correlation area.

The rubber box is disabled at **I**. The default cursor is reenabled at **K** ready for the next pair of pointings, after being initialized at **J** to the last location indicated by the operator.

At **L**, the size and position parameters for the GSCORR at **N** are calculated, in world-coordinate units.

The code at **M** checks for the end condition.

```
CORR1: PROCEDURE OPTIONS (MAIN);
DCL (DEVICE_ID,DEVICE_TYPE)  FIXED BIN(31);
DCL INWIN                    FIXED BIN(31);
DCL (X1,Y1,X2,Y2)            FLOAT DEC (6);
DCL (XPOS,YPOS)              FLOAT DEC(6);
DCL SIZE(2)                  FLOAT DEC(6);
DCL SEGNUMS(100)             FIXED BIN(31);
DCL TAGS(100)                FIXED BIN(31);
DCL HITS                     FIXED BIN(31);

CALL FSINIT;                      /* Initialize GDDM            */
CALL GSUWIN(0.0,100.0,0.0,100.0); /* Uniform window coordinates */

N=1;
DO I=2.5 TO 92.5 BY 10;           /* Draw array of crosses      */  A
  DO J=2.5 TO 92.5 BY 10;
     CALL GSSEG(N);               /* Open segment for each cross*/  B
     CALL GSTAG(1);               /* Tags must be nonzero       */
     CALL GSMOVE(I,J);            /* Draw cross                 */
     CALL GSLINE(I+1,J+1);
     CALL GSMOVE(I,J+1);
     CALL GSLINE(I+1,J);
     CALL GSSCLS;                 /* Close segment              */
     N = N+1;
  END;
END;

CALL GSENAB(2,1,1);               /* Enable default cursor      */  C

DO WHILE(1>0);
  CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID); /*Read first corner     */  D
  CALL GSQLOC(INWIN,X1,Y1);

  CALL GSENAB(2,1,0);            /* Disable default cursor      */  E
  CALL GSILOC(1,5,X1,Y1);        /* Initialize ..               */  F
  CALL GSIDVF(2,1,1,X1);         /*            .. rubber ..     */
  CALL GSIDVF(2,1,2,Y1);         /*                      .. box */
  CALL GSENAB(2,1,1);            /* Enable rubber box cursor    */  G

  CALL GSREAD(1,DEVICE_TYPE,DEVICE_ID);  /* Read second corner  */  H
  CALL GSQLOC(INWIN,X2,Y2);
  CALL GSENAB(2,1,0);            /* Disable locator             */  I
  CALL GSILOC(1,0,X2,Y2);        /* Leave position unchanged &  */  J
  CALL GSENAB(2,1,1);            /*  reenable as default cursor */  K
```

Figure 69 (Part 1 of 2). Correlation with rubber box

```
  XPOS = (X2+X1)/2;              /* Position of ..              */ L
  YPOS = (Y2+Y1)/2;              /*         .. center of area   */
  SIZE(1) = ABS(X2-X1);          /* Absolute size ..            */
  SIZE(2) = ABS(Y2-Y1);          /*            .. of area      */

  IF SIZE(1) = 0 | SIZE(2) = 0   /* End when no area defined    */ M
    THEN GO TO FIN;

         /* TYPE  POSITION    SIZE    PRIMITIVES&SEGS   HIT NO.*/
  CALL GSCORR(1,  XPOS,YPOS, 2,2,SIZE, 100,SEGNUMS,TAGS,    HITS);   N

  DO I=1 TO HITS;
    CALL GSSATS(SEGNUMS(I),2,0); /* Make segment invisible      */
  END;
END;

FIN:
CALL FSTERM;                     /* Terminate GDDM              */
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END CORR1;
```

Figure 69 (Part 2 of 2). Correlation with rubber box

Querying segment structure in specified area using call GSCORS

GSCORS is specifically used to correlate segments structured by using GSCALL, covered in “Calling segments from other segments, using GSCALL” on page 165. You need to understand that call before you use GSCORS.

GSCORS is a more sophisticated version of GSCORR. Your program specifies a rectangular aperture,
and GDDM returns tag and segment identifier pairs for each segment completely or partly contained within it, together with one tag and segment identifier pair for any segment calling **that** segment, then any segment calling **that** segment, and so on, repeated until the root segment is reached. The returned data for each calling segment is the segment identifier, and the tag of the primitive that immediately precedes the GSCALL to the called segment. The **calling** segments do not have to be completely or partly contained within the specified area.

Here is a typical call:

```
DECLARE SZ(1) FLOAT DEC(6);
DECLARE SGS(50) FLOAT DEC(6);
DECLARE TGS(50) FLOAT DEC(6);
DECLARE NUMHITS FIXED BINARY (31);

SZ(1) = 5;

   /* CORR-TYPE  POSITION  TYPE  SIZE  HITS  DEPTH       NUMBER */
CALL GSCORS( 1,  20.0,30.0,  1,  1,SZ, 10,    5, SGS,TGS, NUMHITS);
```

As an example of a use of GSCORS, let's imagine, as in “Calling segments from other segments, using GSCALL” on page 165, a structure consisting of a building segment, containing office segments, that themselves contain furniture segments.

If you have an application that enables the user to interactively reposition items within the building plan, using a pick device, then the user would be able to pick all or part of a desk, and then proceed to drag not just the desk, but also the office that contains it, around the building plan. This would be possible because the program using GSCORS would have not only the segment identifier of the picked desk segment, but also the segment identifier of the office segment that called it.

For a description of GSCORS, see the *GDDM Base Application Programming Reference* book.

Interactive graphics with multiple partitions

The calls for defining logical input devices described in this section apply to the current page. When multiple partitions are used, each page can have its own set of logical input devices. The user can interact with all of the partitions that have logical input devices enabled. So if you want end users to use interactive graphics in all of the partitions that your application program creates, you need to enable logical input devices on the current page of each partition.

For example, if two partitions exist, with enabled locator and pick devices, the user can select objects from either partition. The partition from which the selection is made becomes the current partition.

On a workstation that supports a number of different echoes (such as the 3270-PC/G) the echoes shown on the screen are those defined for the current partition. The user can still move the locator to a different partition, but the locator echo changes to the default when its position goes outside the current partition's graphics field.

Device variations with interactive graphics

The preceding sections of this section refer primarily to the 3270-PC/G and /GX. The following sections describe functional variations on other types of device.

On other terminals with vector graphics capabilities

The main differences affecting interactive graphics on devices such as the 3179-G, 3472-G, and 3192-G terminals, and on workstations supported by GDDM-PCLK and GDDM-OS/2 Link, are that the following are not supported:

- Tablets
- String and stroke devices
- Locator echoes
- Rubber-band locators
- Rubber-box locators.

On terminals that use PS stores for graphics

Members of the IBM 3270 family that use programmed symbols for graphics, such as the 3279, have fewer graphics capabilities than the 3270-PC/G and /GX family of workstations. The main differences affecting interactive graphics are:

- The ordinary 3279 terminal has fewer processing capabilities than the 3270-PC/G and /GX family. Many operations, such as vector-to-raster

conversion, have to be done in the host instead of in the terminal. Others, such as segment dragging, are not supported at all.
- The 3279 has no purely graphics input device (mouse, puck, or stylus). The alphanumerics cursor serves as the graphics cursor, under the control of the cursor keys. Its accuracy is restricted to character cells.
- On the 3279 terminal, the keys that can trigger input or act as choice devices are different from those on a 3270-PC/G or /GX.

Enabling logical input devices: A typical GSENAB call is:

```
    /* DEVICE_TYPE   DEVICE_ID   CONTROL */
CALL GSENAB(1,           1,         1);    /* Enable PF keys    */
                                           /* as choice devices */
```

The valid parameter values for an ordinary 3270 terminal, such as the 3279, are as follows.

- For the first parameter, which specifies the type of logical input device being enabled, there are three valid values:

  1 Choice
  2 Locator
  3 Pick.

  String and stroke devices are not supported.

- The second parameter, which further describes choice devices, can have one of these values:

  0 ENTER key
  1 PF keys
  2 Alphanumeric light pen
  4 PA keys
  5 CLEAR key.

  The data keys cannot be choice devices.

- As with other types of terminal, the last parameter allows you to disable logical input devices, as well as enable them. A value of 0 tells GDDM to disable the device, and 1 to enable it.

Choice devices: The choice data returned by GDDM is shown in Table 3.

Table 3. Choice data returned by nonPC 3270 terminals

Terminal facility	**Parameter values**	
	GSREAD(1,D_T,DEV_ID)	GSQCHO(NUMBER)
ENTER key	0	0
PF key	1	Number of key (1-24)
Alphanumeric light pen[1]	2	0
PA key	4	Number of key (1-2)
CLEAR key	5	0

[1] Or CURSR SEL key

Locator devices: The locator echo is always the alphanumeric cursor. No other type of echo can be enabled. You can set its initial position with a GSILOC call. The ENTER key, a PF key, or the light pen triggers the locator, whether or not they are enabled as choices.

Pick devices: The alphanumeric cursor echoes the pick device. No indication of the size of the pick aperture is given on the screen. The default aperture size is the height of a hardware cell. You can set the initial position with the GSIPIK call, and change the size with a GSIDVF call.

On the IBM 5080 and 6090 graphics systems

The IBM 5080 and 6090 Graphics Systems are designed for polyline CAD/CAM applications.

GDDM/MVS, GDDM/VM, and GDDM-PGF communicate with the 5080 and 6090 through GDDM/graPHIGS, a separate IBM licensed program. This support allows graphics applications written for other devices to be run on a 5080 or 6090. Interactive graphics applications written for other types of display can run on these graphics systems but do not take advantage of their full capabilities.

The 5080 or 6090, with or without the 3270 feature, has interactive graphics capabilities that can be programmed like those of a 3270-PC/G or /GX.

The main differences are:

- The 5080 or 6090 must be explicitly opened by a DSOPEN call.
- Neither the 5080 nor the 6090 has a mouse input device.
- Valid choice devices are:
  Enter
  PF key
  PA key or CLEAR, by switching to 3270 during read operation
  Puck/stylus
- Valid locator echoes are:
  `0` Small cross
  `1` Small cross
  `2` Crosshair
  `3` Tracking cross
  `4` Rubber band
  `5` Rubber box
  `6` Draggable segment. The whole of the segment appears white.
- When using rubber band and rubber box echo types, if the position of the fixed end or corner is not visible at the time of a GSREAD call, GDDM does not ensure that the initial position and type of the locator echo are correct.

5550-family Multistation

Support is the same as for 3270-PC/G, described in this section, with these exceptions:

- Segment dragging is not supported
- String and stroke devices are not supported
- A mouse is supported as the choice, locator, and pick devices. Neither a puck nor a stylus are supported.
- For a locator device, GSILOC echo types 3 through 5 are not supported.

Chapter 12. Using symbol sets

Chapter 2, “Drawing graphics pictures” on page 25, describes the use of symbols sets for graphics markers and shading patterns. Symbol sets are even more useful in GDDM applications that perform alphanumeric and graphics text tasks.

You do not need to specify a symbol set for either graphics or alphanumeric text: GDDM always supplies a default. If you want to use a symbol set other than the default set, there are two operations that your program must perform.

The first is to load the required symbol set into main storage. This is best done before the program opens any graphics segments. Several symbol sets can be loaded and stored concurrently, so the second operation is to specify which one is to be used for a given piece of text.

Table 4 on page 235 shows these operations, as steps 1 and 3, in the context of the other major text output calls.

Most of this section applies to devices with programmed symbols. For device variations, see “Device variations with symbol sets” on page 254.

General information about symbol sets

GDDM supports the use of two distinct types of symbols or characters: image symbols and vector symbols. Printer fonts, which are not supplied by GDDM but which GDDM programs can use, are described in Chapter 20, “Sending output from an application to a printer” on page 399.

Image symbols are patterns of dots, each dot corresponding to one screen position or pixel. These symbols are therefore of fixed size. GDDM supplies an interactive Image Symbol Editor to enable users to create their own image symbol sets. This editor is described in the *GDDM Using the Image Symbol Editor* book. When a symbol set has been created, it is stored on disk and is available for use by any GDDM program.

The other type, **vector symbols**, are defined as a sequence of straight and curved graphics lines. When you write a program that displays vector symbols, you can use GDDM to manipulate the lines that make up the symbols, and therefore display the symbols at any required size, angle, shear (italicization) or aspect ratio. GDDM supplies an interactive Vector Symbol Editor to enable the user to create his own vector symbol sets. This editor is described in the *GDDM-PGF Vector Symbol Editor* book. Once created, both image and vector symbol sets are saved on disk for subsequent use by GDDM programs.

Figure 70 on page 234 illustrates the difference between image symbols and vector symbols.

A symbol set consists of a number of symbols (up to 256 in a vector symbol set or 190 in an image symbol set), and each symbol is associated with a position in the symbol set known as a **character code**. A character code may be expressed either as a hexadecimal number (in the range `X'00'` through `X'FF'` for vector symbols or `X'41'` through `X'FE'` for image symbols), or as the EBCDIC character normally occupying that position. Most symbol sets contain representations of a

© Copyright IBM Corp. 1982, 1996

font, that is, the alphabet, numerals, and special characters all in a single style such as italic or Gothic. When the program sends the string ABC to the terminal using such a symbol set, the letters A, B, and C appear in the particular style of that symbol set.

The symbol set need not represent a font, however. The user may create an image symbol set (using the Image Symbol Editor) that has, for example, a multicolored company logo at position “A” (X'C1'). When the program issues a

```
CALL GSCHAR(X,Y,1,'A');
```

using this symbol set, the company logo is added to the graphics that appear on the device.

GDDM supplies some font symbol sets of both image and vector types for use with the product. They are described in the *GDDM Base Application Programming Reference* book. and illustrated in the user’s guides for their respective symbol editors.

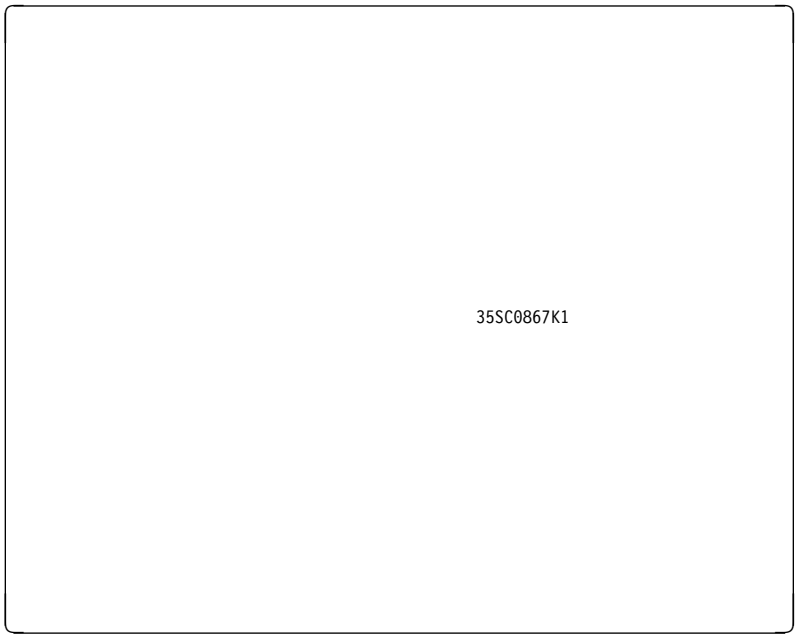


Figure 70. Comparison of image and vector symbols

Table 4. Examples using symbol sets for alphanumerics and graphics text

ALPHANUMERICS	GRAPHICS TEXT

1. Load the symbol set
`CALL PSLSS(0,'ADMITALC',199);`
	1. Load the symbol set
	`CALL GSLSS(1,'ADMITALC',199);`
	or
	`CALL PSLSS(0,'ADMITALC',199);`

2. Define the field `CALL ASDFLD(33,10,20,2,8,0);`	2. Set the character mode `CALL GSCM(2);`
3. Specify which loaded symbol set to use `CALL ASFPSS(33,199);`	3. Specify which loaded symbol set to use `CALL GSCS(199);`
4. Write the characters onto the page `CALL ASCPUT(33,4,'TEXT');`	4. Write the characters onto the page `CALL GSCHAR(40.0,5.0,4,'TEXT');`
5. Send the page to the terminal `CALL ASREAD(TYPE,NUM,COUNT);`	5. Send the page to the terminal `CALL ASREAD(TYPE,NUM,COUNT);`

Each symbol set has a name of up to eight characters. On most subsystems this is a member name in a library devoted to symbol sets. Under CMS, the scheme is slightly different. The symbol set “SCRIPT55,” for example, might exist on any disk in the current search order. If it was on the user's A-disk, its full name would be “SCRIPT55 ADMSYMBL A.”

A GDDM application program refers to a symbol set by name only when it is loading the symbol set. After that, the program refers to the symbol set by its identifier, which is the number you allocate to it at load time.

Loading symbol sets for graphics text

For mode-2 graphics text, any image symbol set can be used, and for mode-3, any vector symbol set. This call loads a symbol set from auxiliary storage into main storage for use in mode-2 or -3 graphics text:

```
/*        TYPE  NAME      ID                                  */
CALL GSLSS(1, 'ADMITALC',194);/* Load image symbol set ADMITALC */
                               /* from auxiliary storage, and   */
                               /* give it an identifier of 194  */
```

In this example, an image symbol set (type 1) is loaded. Information about other types of symbol set that you can load with GSLSS can be found in the *GDDM Base Application Programming Reference* book.

The GSLSS call loads the symbol-set definitions into main storage for use by GDDM. It does **not** load the symbol set into the device as PSLSS would (except on the terminals described in “IBM 3270-PC/G and GX workstations” on page 255).

Imagine, for example, that a subsequent request is made to send the characters XYZ to the screen of an IBM 3472-G terminal using this mode-2 italic symbol set. Then GDDM retrieves from the symbol set the dot patterns at positions X, Y, and Z. It then merges these pixels (in the current color) with the pixels representing the rest of the specified graphics. All of this processing takes place in the host, not at the device.

For mode-1 graphics text, you can only use image symbol sets. The character size must exactly match that of the device on which the text is to be displayed. Such a symbol set can be loaded into one of a device's programmed symbol buffers (also known as PS stores). You must load the symbol set with a PSLSS call. If you intend to use a symbol set that matches the hardware cell size for mode-2 graphics text, you could load it using either PSLSS or GSLSS.

You can query a loaded symbol set with a GSQSSD call. Briefly, you specify a symbol set type and identifier, and GDDM returns its size (for image symbol sets) or aspect ratio (for vector symbol sets). For details of the parameter values of GSQSSD, see the description of the call in the *GDDM Base Application Programming Reference* book.

GSQSSD offers a way of ensuring that the aspect ratio of the character box matches that of a vector symbol set:

```
DCL ARRAY(1) FLOAT DEC(6);
DCL (WIDTH,HEIGHT) FLOAT DEC(6);

   /*TYPE:2=VECTOR S-SET  S-SET ID  ARRAY ELEMENTS  ASPECT RATIO*/
CALL GSQSSD(2,               65,           1,           ARRAY);

HEIGHT = 10;               /* Set height of character box in    */
                           /* world coordinate units ....       */
WIDTH = ARRAY(1) * HEIGHT; /* ... and then its width            */
CALL GSCB(WIDTH,HEIGHT);   /* Aspect ratio of character box now */
                           /* matches that of vector symbol set */
```

GDDM sets the first (and, in this case, only) element of ARRAY to the width of the symbols as a proportion of their height. The proportion was defined when the symbol set was created. The GSCB call ensures that the character box has the same proportions.

Information about using the symbol set you have loaded is given in “Specifying a symbol set for graphics text.”

Specifying a symbol set for graphics text

This GSCS call specifies which symbol set should be used for graphics text characters:

```
CALL GSCS(194);              /* Set symbol set attribute to 194 */
```

The actual symbol set used depends both on this parameter and on the character mode. It is possible to have three symbol sets current in an application (a hardware set, an image set, and a vector set), each with a symbol-set identifier of 194. On most types of device, the chosen character mode then determines which of these sets is used. However, this is not always the case, (see “IBM 3270-PC/G

and GX workstations" on page 255). To ensure device-independence, duplicate identifiers should therefore be avoided in all programs.

If no GSCS call is made, GDDM uses the drawing default to select a symbol set.

The programming example in Figure 71 uses three symbol sets. Two of them are vector symbol sets: one for the heading (part of which is displayed larger than the rest, highlighting the word MAZE), and the other for the subheading and annotations.

The third symbol set is an image symbol set. Only one symbol from this set is used. It is a large and complex symbol, comprising a multicolored maze. It has a character code of X'C1', which corresponds to the letter A in EBCDIC. This symbol set is not one supplied with GDDM. If you want to run this program you will need to created a symbol set using the GDDM Image Symbol Editor and save it with the name 'MAZE'. As explained in "Multicolored image symbols" on page 242, to display a multicolored symbol, the current color must be set to 7 (neutral).

```
MAZE: PROC OPTIONS(MAIN);

DCL (TYPE,NUM,COUNT) FIXED BIN(31);

CALL FSINIT;
CALL GSWIN(0.0,130.0,0.0,130.0);/* Set up the coordinate system */


/*****************************************************************/
/*                   WRITE THE HEADING                           */
/*****************************************************************/

CALL GSLSS(2,'ADMUWCRP',65);  /* Load symbol set for heading    */
CALL GSLSS(2,'ADMUWCSP',66);  /* Load symbol set for annotation */
CALL GSLSS(1,'GGMAZE',67);    /* Load the maze symbol           */

CALL GSCM(3);                 /* Set text mode to vector symbol */
CALL GSCS(65);                /* Make heading symbol set current*/
CALL GSCB(5.0,7.0);           /* Set size and ...               */
CALL GSCOL(6);                /*  ... color of heading          */
CALL GSCHAR(18.0,115.0,5,'THE A'); /* First part of heading     */
CALL GSCB(9.0,16.0);          /* Make character size larger     */
CALL GSCOL(3);                /* Change color                   */
CALL GSCHAP(4,'MAZE');        /* Next part of heading           */
CALL GSCB(5.0,7.0);           /* Reset size and ...             */
CALL GSCOL(6);                /*  ... color                     */
CALL GSCHAP(17,'ING COMPUTER GAME'); /* Last part of heading    */
```

Figure 71 (Part 1 of 2). Program using symbol sets for graphics text

```
/*****************************************************************/
/*                   WRITE THE SUBHEADING                        */
/*****************************************************************/

CALL GSCS(66);                /* Make symbol set current        */
CALL GSCB(4.0,5.0);           /* Set size and ...               */
CALL GSCOL(4);                /*  ... color                     */
CALL GSCHAR(20.0,105.0,39,'CAN YOU GET THE CURSOR OUT OF THE MAZE?');


/*****************************************************************/
/*                   WRITE THE ANNOTATIONS                       */
/*****************************************************************/

CALL GSCOL(2);                /* Set the color                  */
CALL GSCHAR(58.0,45.0,10,'START HERE');
CALL GSCHAR(46.0,85.0,8,'END HERE');


/*****************************************************************/
/*                   DRAW THE MAZE                               */
/*****************************************************************/

CALL GSMIX(3);                /* Maze to underpaint annotation  */
CALL GSCM(2);                 /* Set text mode to image symbol  */
CALL GSCS(67);                /* Make maze symbol set current   */
CALL GSCOL(7);                /* Set color to neutral           */
CALL GSCHAR(42.2,0.0,1,'A');  /* Write the maze symbol          */

CALL ASREAD(TYPE,NUM,COUNT);  /* Send to display                */

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

 END MAZE;
```

Figure 71 (Part 2 of 2). Program using symbol sets for graphics text

Loading symbol sets for alphanumeric text

Only image symbols can be used for alphanumeric applications. If you want to use an image symbol set for alphanumerics in a program, make sure that the symbols are the same size as the hardware cells of the current device. If the alphanumerics are to appear on a 3472-G, for example, the symbols in the symbol set must be 9 pixels by 16.

This is a typical statement to load a symbol set from auxiliary storage. When the current page is sent to the display (typically, when the next ASREAD is executed), GDDM loads the symbol set into a PS store at the terminal.

```
CALL PSLSS(3,'SCRIPT55',193);/* Load symbol set into PS store 3 */
```

This call specifies that the image symbol set called “SCRIPT55” is to be loaded into PS store 3 of the device and given it an identifier “193.” Any subsequent calls in the program, that refer to the symbol set, such as ASFPSS, must use this identifier. The number you specify as the **symbol-set identifier** must lie in the range 65 through 223.

If you want your application to use more than one symbol set for alphanumerics, you just issue more PSLSS calls in the program, using a different symbol set identifier for each image symbol set loaded.

Specifying a symbol set for use in an alphanumeric field

To use a symbol set for alphanumerics after you have loaded it, you specify it as an alphanumeric attribute. The ASFPSS call sets the symbol-set attribute for an alphanumeric field:

```
CALL ASFPSS(8,193);           /* Set field symbol-set attribute */
```

This call specifies that all subsequent output in alphanumeric field 8 should use the symbol set that was loaded with an identifier of 193. If the field has been defined to accept input, any characters entered into the field appear on the screen as symbols from the same loaded symbol set.

Most commonly the symbol set loaded into the PS store is a font of some sort. If it is, for example, a Gothic font, the effect of the ASFPSS and a CALL ASCPUT(8,6,'ABC123') is to send a Gothic version of ABC123 to the screen. Any input typed in the field also appears in Gothic characters.

Setting the symbol set attribute to 0 requests the hardware, nonloadable symbol set (in other words, the standard character set of the device). This symbol set is used if no ASFPSS call is executed for a field.

Specifying a symbol set for individual characters in a field

If you need to use different symbol-set attributes within a single alphanumeric field, character symbol-set attributes must be used:

```
CALL ASCSS(8,6,'AAAA  ');/* Set character symbol-set attributes */
```

This call must be issued after the data is put into the field by an ASCPUT.

To specify the symbol set for the field attribute, a fullword parameter was used (set to 193 in the example given on page 239). This is not a suitable method for character attributes. The symbol-set identifiers are therefore expressed as 1-byte hexadecimal numbers. For coding purposes it is most convenient to use numbers that correspond to an EBCDIC letter. The letter A, for example, corresponds to X'C1' which is 193 in decimal.

The above ASCSS statement therefore requests that the first 4 characters of field 8 should be displayed using the symbol set with identifier 193. The 5th and 6th characters use whichever symbol set was specified in the field attribute (the ASFPSS call, if any); this is the meaning of the blanks here. If the field has more than 6 characters in it, the remaining characters also take their attributes from the field-attribute specification.

ASCPUT sets all character attributes to their default settings. If you use an ASCSS call to set character attributes for the contents of a field, you must place it after the ASCPUT statement for that field in the program. Character symbol-set attributes act on the character data in the field rather than on the field itself.

The effect of typical ASFPSS and ASCSS calls may be seen in Figure 18 on page 76.

Figure 72 on page 241 contains an example of how to specify symbol sets for alphanumeric fields:

```
CALL FSPCRT(1,32,80,2);  /* Create page that allows char attrs. */
CALL PSLSS(3,'GOTHIC2',194); /* Load Gothic s-set with id = 194 */
CALL PSLSS(0,'ADMITALC',195);/* Load italic s-set with id = 195 */
CALL ASDFLD(1,14,56,1,7,0);  /*Define field 1, 7 characters long*/
CALL ASDFLD(2,18,40,1,5,0);  /*Define field 2, 5 characters long*/

CALL ASFPSS(1,195);   /* Set field 1's s-set attribute to italic*/
CALL ASCPUT(1,7,'ABCDEFG');       /* Assign data to field 1     */
CALL ASCPUT(2,5,'PQRST');         /* Assign data to field 2     */
CALL ASREAD(TYPE,MOD,COUNT);      /* Send 1st output to screen  */
 /*************************************************/
 /*                                               */
 /* FIELD 1: ABCDEFG all appear in italic         */
 /* FIELD 2: PQRST all appear in the default      */
 /*          (hardware) symbol set                */
 /*                                               */
 /*************************************************/

CALL ASFPSS(2,194);   /* Set field 2's s-set attribute to Gothic */
CALL ASCSS(1,4,' BBB');           /* Chars 2-4 use               */
                                  /* s-set 'B' (X'C2', 194)      */
CALL ASREAD(TYPE,MOD,COUNT);      /* Send 2nd output to screen   */
 /*************************************************/
 /*                                               */
 /* FIELD 1: A...EFG  appear in italic            */
 /*          .BCD...  appear in Gothic            */
 /* FIELD 2: PQRST all appear in Gothic.          */
 /*                                               */
 /*************************************************/

CALL ASCPUT(1,7,'HIJKJLM');       /* Assign new data to field 1,*/
                                  /* thereby canceling the      */
                                  /* character s-set attributes.*/
CALL ASCSS(2,3,'CCC');            /* Chars 1-3 use              */
                                  /* s-set 'C' ( X'C3', 195)    */
CALL ASFPSS(2,0);                 /* Reset field 2 to the       */
                                  /* hardware nonloadable set   */
CALL ASREAD(TYPE,MOD,COUNT);      /* Send 3rd output to screen  */
 /*************************************************/
 /*                                               */
 /* FIELD 1: HIJKLMN  all appear in italic        */
 /* FIELD 2: PQR.. appear in italic               */
 /*          ...ST appear in the default          */
 /*          (hardware) symbol set                */
 /*                                               */
 /*************************************************/
```

Figure 72. Routine to specify symbol sets for alphanumeric fields

Changing a field attribute alters the appearance of the data in that field the next time a screen output is performed. This applies even if the data of the field was already sent out on a previous screen output. The same is true of character attributes. The new attributes are applied to the current field contents - even if new data has been typed into the field by the terminal operator.

Input of character symbol-set attributes

If the display device has a keyboard that permits input of character attributes, there are buttons on it marked PSA, PSB ... PSF. These correspond to PS stores 2 through 7.

In the preceding example, the PSLSS for the Gothic symbol set explicitly requested PS store 3. If the terminal operator presses the PSB button, all subsequent typed characters appear on the screen in Gothic. A character attribute of PS store 3 is returned to GDDM for all such characters. You may query the character attributes of the input symbol-set by issuing an ASQSS call:

```
CALL ASQSS(1,7,CHAR7);   /* Place s-set character attrs in 'char7' */
```

This call returns the first seven symbol-set character attributes of field 1 into the variable CHAR7. By the time the attributes arrive in the variable, they are in the same form as in a corresponding ASCSS call. In other words, there is a B (X'C2', decimal 194) for all positions where the character attribute was set to PSB (the location of the Gothic font).

Multicolored image symbols

When image symbol sets are created, it is possible to make them multicolored. The dots making up the symbols may each be any of the 7 colors available. When you use multicolored image symbols, whether for alphanumerics, or for mode-1 or mode-2 graphics text, the color must be set to 7 (neutral). If any other color is specified (or defaulted), the dots in each symbol are all in that color.

Symbols for pounds, dollars, and cents

Some EBCDIC character codes are reserved for “national use” characters. Because device controllers are configured differently in different countries, the appearance of these characters on the screen varies from country to country. When such character codes are displayed using GDDM-supplied symbol sets, apparent conflicts may occur. This problem may be solved in one of two ways:

- By using GDDM’s code-page conversion facilities (see “Using GDDM to convert character code pages for international applications” on page 247)
- By using one of the GDDM symbol editors to create a symbol set with national-use characters set to the desired values

Device-dependent symbol-set suffixes

To enable programs to run against different devices, symbol-set names may be specified that end in a **substitution character**. This is coded as a period, for example:

```
CALL GSLSS(4,'SCRIPT5.',193);
```

When GDDM comes to load the symbol set, it replaces the substitution character with one from a set of device-dependent one-character suffixes.

A list of suffixes is given in the *GDDM Base Application Programming Reference* book.

Manipulating symbol sets in a program

You may need to manipulate GDDM symbol sets in your application programs. This section summarizes some useful calls. For a description of each, see the *GDDM Base Application Programming Reference* book.

Symbol sets and program variables

This call reads a symbol set from auxiliary storage into a program variable:

```
CALL SSREAD('ADMITALC',1200,CHAR1200); /* Read the symbol set   */
                                       /* from auxiliary storage*/
                                       /* into the user's       */
                                       /* program storage       */
```

The symbol set is called ADMITALC. The variable is 1200 bytes long and is called CHAR1200. If the length is insufficient, an error message is issued. The symbol set may be of any mode. After reading in the symbol set, the program might, for example, set a special character into one of the character codes.

This call would write the same symbol set back to auxiliary storage:

```
CALL SSWRT('ADMITALC',1200,CHAR1200);  /* Write the symbol set  */
                                       /* to auxiliary storage  */
                                       /* from the user's       */
                                       /* program storage       */
```

Loading a symbol set from an application program

The GSDSS call is similar to GSLSS in that it loads a symbol set into GDDM's storage ready for transmission to the terminal, but it loads from a program variable rather than auxiliary storage:

```
/*    IMAGE SET    NAME    S-SET ID  VARIABLE LENGTH   VARIABLE */
CALL GSDSS(1,   'ADMITALC',  194,         1200,        CHAR1200);
```

The name serves only to help identify the symbol set within the program. You can choose any helpful name or leave it blank. It does not refer to any symbol set on auxiliary storage.

The PSDSS call is the equivalent of the PSLSS call. It loads a hardware image symbol set from a program variable into a specified PS store at the terminal. The load takes place at the next ASREAD call.

```
CALL PSDSS(3,'SCRIPT55',194,1200,CHAR1200);
```

This call loads into PS store 3 the symbol set held in the 1200-byte-long variable CHAR1200. The name of the symbol set is 'SCRIPT55' and its identifier is 194. The name can be left blank.

The PSLSSC call is similar to PSLSS, but is a conditional load:

```
CALL PSLSSC(3,'SCRIPT55',194);
```

If PS store 3 already contains a symbol set with identifier 194, the load is not performed. This scheme may be used even when the PS store was loaded by a different instance of GDDM.

Using double-byte characters for graphics text

You can use all three modes of graphics text to display the double-byte character set (DBCS) characters used in some Asian countries. (On the IBM PS/55 workstation or 5550 Multistation, you can use the device's hardware symbol set to display DBCS characters in alphanumeric fields, and receive DBCS alphanumeric input, as explained in “Using procedural alphanumerics for double-byte characters” on page 265.) The same hardware symbol set can be used for mode-1 graphics text. With the GSLSS call, you can load other DBCS symbol sets for use with mode-2 and mode-3 graphics text.

If you use mode-3 graphics text in your program, and load the appropriate vector symbol set into storage, you can display double-byte characters on any device.

Each DBCS character is represented by a two-byte code instead of the single-byte EBCDIC-type code used for Latin characters. You must specify the hexadecimal codes in a GSCHAR or GSCHAP call. The length you specify in these calls must be the number of bytes – twice the number of DBCS characters.

GDDM supplies three special multipage, double-byte symbol sets for Kanji graphic text – an image symbol set for mode-2 text, a vector symbol set for mode-3 text, and a special, high-quality vector symbol set, also for use with mode-3 text.

GDDM also supplies three single-byte character sets for Kanji graphics text – two image symbol sets for use with mode-2 text, and one vector symbol set for mode-3 text.

GDDM supplies a similar multipage, mode-3 text vector symbol set for Simplified Chinese graphic text.

If you require the default double-byte symbol set, you can specify a special symbol-set identifier of 8 on a GSCS call. An example is given in the program below, which was written for a device, such as the IBM PS/55 workstation, that supports DBCS characters.

```
/*****************************************************************/
/*          FIRST SET UP HEX CODES IN AN ARRAY                   */
/*****************************************************************/
DCL KC(65:254) CHAR(1);    /* Array to hold hexadecimal numbers */
DCL INDEX FIXED BIN(15);                      /* Local variable */
DCL BIT16 BIT(16);                            /* Local variable */

DO INDEX=65 TO 254;/* Initialize array with hex'41' through 'FE'*/
  BIT16=UNSPEC(INDEX);                    /* Convert to bit      */
  UNSPEC(KC(INDEX))=SUBSTR(BIT16,9,8);   /* Extract last 8 bits */
END;

/*****************************************************************/
/*          NOW WRITE THE KANJI CHARACTERS                       */
/*****************************************************************/
CALL GSCM(3);                        /* Vector symbol mode       */
CALL GSCS(8);                        /* Default DBCS symbol set  */
                                     /* ( Kanji on the PS/55 )   */

DECLARE KANJI_DATA5 CHARACTER(10);   /* String for 5             */
                                     /* Kanji characters         */
KANJI_DATA5=KC(65)  || KC(192) ||    /* Assign                   */
            KC(...) || KC(...) ||    /*     five                 */
            KC(...) || KC(...) ||    /*       two-byte           */
            KC(...) || KC(...) ||    /*           Kanji          */
            KC(...) || KC(...);      /*               characters */

CALL GSCHAR(8,8,10,KANJI_DATA5);     /* Write the Kanji          */
```

Figure 73. Routine to add graphics text to the page using double-byte characters.

Another method, which enables ordinary single-byte characters to be mixed in a single string with double-byte characters, is to use the special shift-out (SO) and shift-in (SI) characters. The data between these two special characters is interpreted by GDDM as double-byte. Other characters are interpreted as single-byte. With this method, you do not need a GSCS(8) call.

The SO code is `X'0E'` and the SI is `X'0F'`. You must allow one byte for each of these and two bytes for each Kanji character. Within any string, only SO/SI pairs are allowed, in that order.

Here is an example:

```
/*****************************************************************/
/*          SET UP SO/SI CHARACTERS                              */
/*****************************************************************/
DCL (SO,SI) CHAR(1);                /* Shift-out & shift-in     */
SO='0E'X;                          /* Set shift-out codepoint   */
SI='0F'X;                          /* Set shift-in codepoint    */

/*****************************************************************/
/*          WRITE MIXED KANJI AND LATIN DATA                     */
/*****************************************************************/
CALL GSCM(2);                               /* Image symbol mode */

DECLARE MIXED_DATA28 CHARACTER(28);/*String for mixed characters*/
                                   /*                    bytes*/
MIXED_DATA28='LATIN'||             /* 5 Latin characters   5  */
             SO ||                 /* Shift-out            1  */
             KC(65)  || KC(192) || /* 3 Kanji characters   6  */
             KC(...) || KC(...) ||
             KC(...) || KC(...) ||
             SI ||                 /* Shift-in             1  */
             'LATIN AGAIN' ||      /* 11 Latin characters 11  */
             SO ||                 /* Shift-out            1  */
             KC(...) || KC(...) || /* 1 Kanji character    2  */
             SI;                   /* Shift-in             1  */
                                   /*        Total bytes  28  */

CALL GSSEN(2);
CALL GSCHAR(8,1,28,MIXED_DATA28);
```

Figure 74. Routine to add graphics text to the page mixing single- and double-byte characters.

If you left out the GSSEN call from the above code, the character positions used by the SO and SI codes would be output to the display as blanks. GSSEN applies to the current page, and its one parameter determines whether the SO and SI characters are to be represented by blanks between the single-byte and double-byte characters or whether the single-byte and double-byte characters are to be adjacent.

Before executing such a program, you need to specify a GDDM external default.

GDDM default required for Kanji and Simplified Chinese

If you use the SO/SI method, you must tell GDDM to check for the shift codes in graphics text strings. You do this using the GDDM external defaults mechanism, which is similar to the nicknames mechanism described in “How GDDM compounds device-definition information for a conceptual device” on page 375. This statement specifies the required default:

```
ADMMDFT MIXSOSI=YES
```

It can be passed to GDDM in any of the ways described in “How GDDM compounds device-definition information for a conceptual device” on page 375. If an ESSUDS or ESEUDS call is used, it should be executed immediately after the FSINIT. Full information about the defaults mechanism is given in the *GDDM Base Application Programming Reference* book.

Using GDDM to convert character code pages for international applications

If you write an application program that:

- Sends text to an output device, using alphanumeric or graphics text functions
- Receives text input, either from the end user's keyboard or from text files that are passed to it
- Loads GDDM object files that may contain text

you may need to make some special considerations for users of your program in other countries.

If your program is used in another country to send text output to a display or printer, most letters and numbers appear exactly the same as they did when you coded them in the program. With some characters, however, this may not always be the case.

Problems may also occur when a program receives user input through a keyboard from a foreign country.

General information on code pages and national characters

IBM devices attached to host computers use the Extended Binary Coded Decimal Interchange Code (EBCDIC) as the standard way of representing single-byte characters. For every device, the way in which characters are mapped onto EBCDIC codes is determined by a **code page**.

The codes for the Latin letters (A through Z), in uppercase and lowercase, and for Arabic numerals (0 through 9) are consistent across all **device code pages**. But other codes, designated as being for national use, vary in the characters assigned to them, especially between devices made for different countries. Such codes are entirely device-dependent.

The code for a character generated by a device in one country may produce an entirely different character on a device from another country or it may not have any corresponding character on the second device and may not be displayable or printable.

On U.S. devices, for example, character codes `X'5B'` and `X'4A'` represent the dollar and cent signs respectively. A British keyboard (and terminal) uses these codes to represent pound and dollar signs, respectively.

Suppose a multinational company based in the U.K. wants to monitor its share price in major stock exchanges around the world. At the corporate HQ every day, they receive files from the Tokyo, Frankfurt, London, and New York stock exchanges, containing the price of their shares at each exchange when it closed and an indication of the change in price from the previous day. The file from New York might typically read;

```
Share Price: $33.75     Change: down 5¢
```

However, when the file is displayed or printed in the U.K., the output appears like this

```
Share Price: £33.75     Change: down 5$
```

It would be very difficult to make sound business decisions based on this information.

To help overcome such problems, GDDM supports special extensions to the standard EBCDIC code pages, called **Country Extended Code Pages** (CECPs).

Country-extended code pages

All CECPs contain the same set of characters mapped onto 190 code points. The difference between one CECP and another is in the assignment of characters to code points, that is, the order in which the 190 characters are mapped onto the 190 code points.

The CECP for a particular country or group of countries maps national-use characters onto the same code points as the traditional EBCDIC code page for that country. However, it also includes the national-use characters of all other CECPs. This means that any character entered into a file on a device using a particular CECP can be represented by some code-point of each other CECP.

CECPs supported by GDDM

These are defined in translation tables contained in the alphanumeric defaults module, ADMDATRN. Those defined in the standard module are listed in the *GDDM Base Application Programming Reference* book. For information on how to modify this module, see the *&sca.* manual or consult your system-support personnel.

Most of the GDDM sample symbol sets, that contain single-byte Latin characters, contain the full set of 190 CECP characters.

If an application contains some mode-3 graphics text calls, and a CECP has been specified as the application code page, ADMDVECP is used as the default vector symbol set. If the GDDM default EBCDIC code page, 00351, is the application code page, the vector symbol set ADMDVSS is used instead.

The three GDDM code pages, default EBCDIC (00351) and Katakana (00290 and 01027), are special and are illustrated in the description of the ASTYPE call in the *GDDM Base Application Programming Reference* book.

Code-page conversion

You can use GDDM to ensure that the code for any character that your program sends as output is converted to the code that outputs the same character on a device with a different code page to yours.

Suppose you write a GDDM application program that reads output data from files and sends it to displays, printers or plotters and reads input data from terminals and stores it in files. If the data was put in the files by a device with the same code page as the output device, the presentation of the output data on the screen or paper contains the correct characters. Otherwise the code page of the data must be converted before any output is sent to the display or printer.

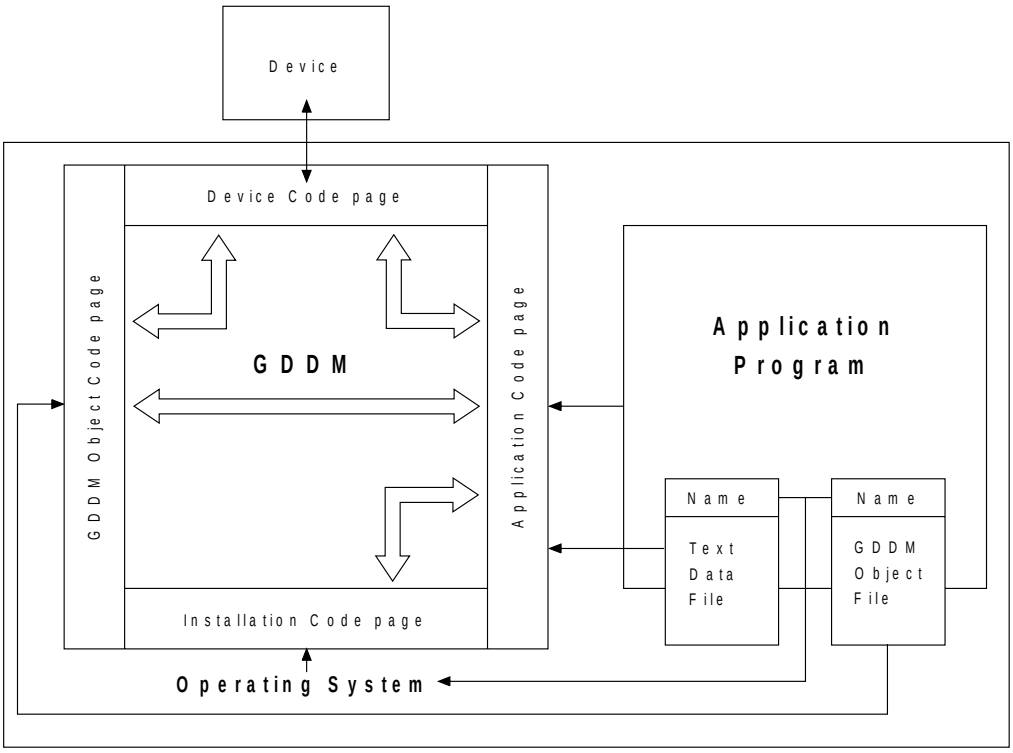


Figure 75. Code-page conversion

Similarly, input data may need to be converted after receipt from the terminal to be sure that all the data in the files uses the same code page.

GDDM can perform these conversions on behalf of an application program if a CECP is specified for use as the **application code page**. The various conversions that GDDM performs for an application program are illustrated in Figure 75.

If you include API calls in the application that pass character data to the device as output, GDDM converts the code points that represent each character on the application code page to code points that represent the same characters on the device code page.

Similarly, input data is converted from code points on the device code page to the code points on the application code page before being returned by API calls to the application.

The application code page is used to convert **all** data flowing between a program and GDDM, whether passed to, or returned by, GDDM, and whether passed as a parameter on a call or in a data structure. It applies to any character data that an application program reads from user files and passes to GDDM, in addition to characters coded literally in the program source and GDDM messages.

If either the source or the target for any of the possible conversions shown in Figure 75 is itself a GDDM code page, **no conversion takes place**.

There are a number of ways in which you can specify an application code page for a program to pass data to, and receive data from, GDDM.

Implicit conversion of code pages by GDDM

If an installation needs to use your application to process some data files it has received from another country, it can get GDDM to perform the necessary code-page conversion by setting two particular GDDM external defaults. In the ADMMDFT macro the **installation code page** is usually set to the national CECP, using the INSCPG default. This ensures that data, such as a filename, passing between GDDM and the operating system is preserved intact when it contains national-use characters from a different code page. The APPCPG default specifies the application code page to be used for conversions when data passes between GDDM and the application. This is also usually set to the national CECP and can be specified in one or more of the following locations:

- The ADMMDFT macro
- The external-defaults module

or

- The user-defaults file (PROFILE ADMDEFS).

An APPCPG default specified in one of these files overrides any other values specified for it in the files above it in this list.

Whenever a device is opened, GDDM queries it to determine the device code page (although you can specify it explicitly also). With the query reply from the device as the device code page and the default setting for the application code page, GDDM can then perform the conversion implicitly.

Some devices do not return their code-page support information to GDDM when they are opened. Users of such devices can specify a device code-page for them by including the DEVCPG processing option on a NICKNAME statement for the device in their user-defaults file.

Even if a device does return code-page support information to GDDM, a device code page specified by the DEVCPG default overrides the query reply. If the device does not return the information, and no processing option has been specified, the installation code page is taken as the device code page.

In most cases, you don't need to specify any code pages in an application program. It is probably best, if you leave this to the users or installers of the application. This allows them to specify an application code page that best matches their input devices and the files in their data bases. However, there are some circumstances in which you may need to program code-page conversion into your program, see "Converting code pages using API calls in the program."

For detailed information on setting GDDM default statements, see the *GDDM Base Application Programming Reference* book.

Converting code pages using API calls in the program

If you wish to ensure that your application uses the correct characters, regardless of the installation and device on which it runs, you can specify code-page conversion within the program by issuing one of these API calls:

ESEUDS or ESSUDS With these calls, you can set GDDM defaults, such as APPCPG.

FSTRAN This call can be used to specify **explicitly** code-page conversion for a character string in the program's memory. Using FSTRAN, one application can deal with data from **several** different code pages.

SPINIT If your application uses the System Programmer Interface to initialize GDDM, you can specify the APPCPG default in the application anchor block specified in this call.

If you specify an application code page within your program, it overrides those set in the end user's defaults file, external defaults module or ADMDFTS macro.

Suppose the multinational company described at 247 writes a GDDM application program to produce a list of the daily share-price data, sent from the various stock exchanges. Because the application needs to process data from a number of different, (but known), national code pages, it can use an FSTRAN call to convert the data from each file to the code points of the U.K. CECP.

If the application reads in the data from the New York file and places it in a character string or an array called NY, this call can perform the conversion:

```
/*          TYPE  FROM-PAGE  TO-PAGE  LENGTH  IN-STRING  OUT-STRING  */
CALL FSTRAN( 0,     00037,    00285,    38,      NY,       NYUK);
```

When the call passes the code point X'F5' to GDDM, this is transmitted to the display unchanged: it represents the character “5” in both the U.S.A. and U.K. CECPs. But when the application passes the code point X'4A' to GDDM, it transmits the code point X'B0' to the screen. This is the code point that represents the “¢” character on the U.K. CECP.

Translating user input on Japanese extended code pages using FSTRAN

You can use the FSTRAN call to translate the character codes of strings entered by users from one of the extended Japanese code pages (290 or 1027) to the other.

You only use FSTRAN to translate code points between code pages that are based on the same character set.

Translating Latin text input on Japanese (Katakana) Extended code page 290 to uppercase with ASFTRN

With the ASFTRN call you can assign a translation table to an alphanumeric field in which you expect user input.

If your application runs on a device that uses Japanese (Katakana) code page 290 with character set 332, any Latin characters entered will be in uppercase.

However, devices configured with Japanese (Katakana) Extended code page 290 do support lowercase Latin input. You can use the ASFTRN call to specify the uppercase translation table to convert end user input to uppercase.

Converting code pages for GDDM objects

Once an application code page has been specified, GDDM normally performs the code-page conversion for an object file, without any instructions from the application program loading the object. With some objects, however, this isn't possible and you may need to include special routines in your program for these.

When an application program loads a GDDM object file, such as an ADMGDF file, GDDM recognizes from a tag on the object file what code page was current when it was originally saved. It then uses that code-page as the **object code page** and displays or prints the object with the correct graphics text characters.

There are two types of GDDM object for which GDDM cannot perform an implicit code-page conversion:

- Object files that were created by a release of GDDM before Version 2 Release 2. If you want an application to be able to load GDDM objects that were created before this release, you can use the ESQCPG call to query the object's code page. If the value returned by GDDM is zero, you can then issue an ESSCPG call to copy the object and tag the copy with a code-page identifier. The ADMUOT end-user utility, described in the *GDDM Base Application Programming Reference* book. can also be used for this purpose.
- ADMGDF files that were created by converting PIF files. These are tagged with the code page that was current when they were converted but the graphics text content is unchanged.

Code-page conversion for symbol sets: Symbol sets are only converted if they contain a character in every CECP code point; that is, every code point in the range `X'41'` through `X'FE'` for image symbol sets, and `X'42'` through `X'FE'` for vector symbol sets.

Code-page conversion for generated mapgroups: The code page of alphanumeric data in ADMGGMAP-type files is converted.

Code-page conversion for ADMIMG, and ADMPROJ files: These contain character data in the description only; this data is converted.

Code-page conversion for chart-format, -data, and -definition files:
GDDM-PGF does not have any extended code-page functions, so no conversion of data in ADMCFORM, ADMCDATA, and ADMCDEF files is carried out. However, when these files are saved, they are tagged with the current application code page.

The character data in chart format and data files can be converted explicitly using the FSTRAN call. The PL/I sample program, ADMUSP7, which is supplied with the GDDM PGF licensed program, converts the data from the object code page to the current application code page.

GDDM object files are described in the *GDDM Base Application Programming Reference* book.

GDDM can perform automatic code-page conversion for ADMPRINT files and ADMGDF files that were created by converting computer graphics metafiles.

Compatibility with releases of GDDM before Version 2 Release 2

GDDM code pages

In some circumstances, an installation may require programs to continue operating without code-page conversion. For this purpose, GDDM provides a special code page called GDDM default EBCDIC (00351). If this is specified as the application code page, it prevents all CECP code-page conversion, apart from explicit conversions using the FSTRAN call.

For GDDM Version 2 Release 2 or later, if no installation or application code page is specified, and no tagged GDDM objects are used, GDDM default EBCDIC is used by GDDM as the application and object code pages, which prevents implicit conversion. Thus, if no action is taken to specify code pages, programs run as they did under releases of GDDM before Version 2 Release 2.

Inhibiting input of extended code points

Some terminals, such as the 3179-G, enable the host computer to specify whether keyboard input of all 190 CECP code points is to be allowed. If disallowed, only a base set (of, typically, 94 code points) can be entered. Attempting to enter one of the new CECP code points puts the terminal into the input-inhibit state.

An external default, CECPINP, controls this function. It enables existing applications to be protected from new code points. It does not affect the use of the new code points in output data, nor the display and printing of the full range of CECP characters.

Code-page conversion for 4250 printers

If your application uses an IBM 4250 printer to print output that may contain national characters from a different code page, you must specify code-page support for the printer's device-code page in a different way.

GDDM performs the conversion for the printer, if either the CPN4250 default for the installation or a GSCPG call within a program specifies one of the code pages supplied by GDDM for the 4250 printer. This code page is only used when the 4250 printer is the current device.

If you want to provide support for these printers from within the program, you should issue the GSCPG call first and then load a 4250 (type 5) symbol set, using the GSLSS call. The specified symbol set is loaded using the code page defined in the GSCPG call or in the external default.

This support is provided independently of CECP support and so the current GDDM device and application code pages do not affect it.

APL characters

The CECP character set does not include APL characters. If you want to write an application that uses these characters, you can either use the GDDM default EBCDIC set (00351) as the application code page or specify the special APL code page (00293) on the ASTYPE call, which overrides the alphanumeric character-code assignments of the device. Alternatively, if a nonloadable APL

symbol set is available, you can use the ASCSS or GSCS calls to specify it for alphanumerics or graphics text.

If you choose code page 00351 and also require the national use characters for your country, you need to replace the GDDM default symbol sets. The process is described in the *GDDM System Customization and Administration* book.

Device variations with symbol sets

The preceding sections of this section refer primarily to the IBM 3472-G terminal. The functions described are the same for most other display devices but some functions are different on particular devices, as described in the following sections.

Transferring programs between different types of device

If you run a program that uses an image symbol set on two terminals with different default cell sizes, the aspect ratio of image symbols changes. This is the case whether the symbol sets are loaded by a PSLSS or a GSLSS call.

Displays that use programmed symbols for graphics

On the IBM 3279 and some other types of 3270 device, the PS stores used for holding symbol sets are the same as those used by GDDM for its graphics. (Variations on other devices are described at the end of the section.)

It would therefore not be sound practice to try to load a symbol set into PS store 4 if some graphics had previously been output. GDDM might currently be using PS store 4 to hold some of the dot patterns making up the graphics. There are several ways round this problem:

1. PS store 4 can be reserved for this usage by issuing a CALL PSRSV(1,4) statement before any graphics output is performed
2. The PSLSS statement itself can be issued before any graphics output is performed
3. The first PSLSS parameter may be set to 0 to ask GDDM to choose a PS store not currently in use.

Querying PS stores

The following PSQSS call queries the first 5 PS stores and returns information into the four arrays specified as parameters.

```
CALL PSQSS(5,TYPES,STATES,SYMBOL_SET_NAMES,SYMBOL_SET_IDS);
```

Releasing a symbol set from a PS store

The following call releases the symbol set with identifier 194 from the PS store containing it:

```
CALL PSRSS(194);
```

Reserving or freeing a PS store

The following call reserves PS store 5 for later use by the application program. The first parameter may also be set to 0 to indicate that the store is to be freed.

```
CALL PSRSV(1,5);
```

IBM 3270-PC/G and GX workstations

The PSLSS call

As with a 3279, PSLSS loads image-symbol definitions into the device's PS storage. The symbols should be the same size, in pixels, as the alphanumerics hardware cells.

The number of PS stores available to the PSLSS call depends on what features the workstation has and how it has been set up. The maximum is two. You can discover the actual number by executing an FSQURY call:

```
/* Query number of available PS stores */
/* And save number in num_PS_stores    */

DCL ARRAY(10) FIXED BIN(31);
DCL NUM_PS_STORES FIXED BIN(31);

CALL FSQURY(0,10,ARRAY);
NUM_PS_STORES = ARRAY(10);
```

The PS stores are monochrome, so multicolored image symbols are displayed in monochrome, using the current color.

The GSLSS call

The GSLSS call uses the same workstation storage as graphics orders. As on the 3472-G, the GSLSS call generally loads the symbol sets into the workstation. They are held in the same storage as is used for graphics orders. It is therefore advisable to release symbol sets when they are no longer required, using GSRSS calls, so that the storage can be reused.

The cell size is different from that of symbols loaded with a PSLSS. GSLSS uses the graphics cell size, whereas PSLSS uses the alphanumerics cell size. The graphics cell size is the default character-box size.

Graphics text:

Mode-1: The symbols do not have to match the hardware cells: they can be of any size. Their horizontal and vertical spacing are equal to their width and depth.

The symbol sets can be loaded by a GSLSS call, or, if the symbols are the same size as the graphics cell, by a PSLSS. Unlike on other 3270 display devices, you should not use the same symbol-set identifier in a PSLSS as in a GSLSS because you cannot predict which is to be made current when the identifier is specified in a GSCS call. It might also cause the PSLSS-loaded set to be erroneously selected for mode-2 graphics text.

Default symbol set: The workstations have built-in image and vector symbol sets. For mode-1 and -2, the workstation image symbols are used by default. Their size equals that of the hardware graphics cell. For mode-3, hardware vector symbols scaled to fit the current character box are used by default. The default character box is the same size as the graphics cell.

For mode-2 and -3, you can specify that a GDDM set be used as the default instead of the hardware set. You do so with a processing option (see “Using DSOPEN to tell GDDM about a device you intend to use” on page 371). In this case, the default symbol sets are the same as on a 3472-G (see “Specifying a symbol set for graphics text” on page 236), except for device-dependent suffixes.

Printers managed by PSF and CDPF

This section describes the use of symbol sets for mode-1 and -2 graphics text on printers such as the IBM 3825 and 4250. In particular, it describes the differences between these devices and ordinary members of the IBM 3270 family, such as the 3472-G. There are no differences with mode-3 text.

Graphics text:

Mode-1: Mode-1 symbols are taken from the default symbol set. The symbols are scaled to fit within the default character box (see “On advanced function printers and the IBM 4250” on page 70).

Mode-2: You can specify an image symbol set, but this is not recommended. Each dot in each symbol is printed as one pixel. On a high-resolution device such as the 4250, the pixels are very close together. To be distinguishable, symbols therefore need to be very large. Vector symbol sets can be specified for mode-2 instead of image symbol sets.

Default symbol set: The default for all modes, if none is specified using a GSCS call, is the vector symbol set ADMUWARP for the 4250 or ADMUVSRP for the 3800.

Plotters

Graphics text support for plotters is similar to that for 3270 terminals such as the 3472-G. The default symbol set for all text modes is the vector set ADMDVSS. Some further information is given in “Symbol sets” on page 450.

Plotters do not support GDDM alphanumerics.

Chapter 13. Advanced procedural alphanumerics

Several uses of GDDM's simpler alphanumeric functions were discussed in Chapter 5, "Basic procedural alphanumerics" on page 71. This section provides guidance on some of the more complex uses the alphanumerics API, such as:

- Defining multiple fields
- Querying modified fields
- Specifying default field attributes
- Using light-pen fields.

The information in this section does not apply to graphics-only devices such as plotters.

Example: Alphanumeric menu program

The MENU program in Figure 76 uses multiple definition of alphanumeric fields to:

1. Present a dinner menu to the user
2. Query which alphanumeric fields have been modified to record the user's food order
3. Inform the user of the cost of the order
4. Recommend a bottle of wine to suit the user's meal.

An example of output from the program is shown in Figure 77 on page 262.

```
MENU: PROC OPTIONS(MAIN);
DCL (TYPE,MOD,COUNT) FIXED BIN(31);    /* Parameters for ASREAD */
DCL (FIELD_IDS(3),LENG(3),I_LENG(3)) FIXED BIN(31);
                                        /* ASQMOD params         */
DCL COSTS(3,3) FIXED BIN(15) INIT(180,230,220,980,1050,750,175,
                                                   240,175);
                                    /* Costs per dish (in cents) */
DCL BILLPIC PIC'$99.99';   /* PL/I picture variable for editing */
DCL CHAR1 CHAR(1);                      /* Temporary variable    */
DCL (BILL,WINE) FIXED BIN(15);          /* Temporary variable    */
DCL BOTTLE(4) CHAR(30) INIT('CHATEAU TALBOT 1977  AT $11.80',
                            'MEURSAULT 1980 AC    AT $15.75',
                            'COTE DE BEAUNE 1979  AT $12.20',
                            'BOLLINGER CHAMPAGNE  AT $23.60' );
                                    /***************************/
CALL FSINIT;                        /* Initialize GDDM         */
                                    /*                         */
CALL GSFLD(1,1,31,80);              /* Define graphics field   */
CALL GSSEG(0);                      /* Open segment            */
CALL GSCOL(6);                      /* Set color to yellow     */
CALL GSMOVE(0.0,0.0);               /* Move to bottom left     */
CALL GSLINE(0.0,100.0);             /***************************/
CALL GSLINE(100.0,100.0);           /*  Draw yellow frame      */
CALL GSLINE(100.0,0.0);             /*  around the screen      */
CALL GSLINE(0.0,0.0);               /***************************/
```

Figure 76 (Part 1 of 4). MENU programming example

© Copyright IBM Corp. 1982, 1996

```
                                  /****************************/
CALL GSLSS(2,'GEP',194);          /* Load Gothic vector set  */
CALL GSCS(194);                   /* Set symbol set attribute*/
CALL GSCM(3);                     /* Set char mode to vector */
CALL GSCB(3.5,8.0);               /* Set character box (size)*/
CALL GSCOL(5);                    /* Set color to turquoise  */
CALL GSCHAR(15.0,90.0,21,'RESTAURANT LA CORNICE');/*Main heading*/
CALL ASDFLD(1,6,15,1,14,2);       /* Protected alpha field   */
CALL ASCPUT(1,14,'FIRST  COURSE:');  /* Assign prompt data      */
CALL ASDFLD(2,12,15,1,14,2);      /* Protected alpha field   */
CALL ASCPUT(2,14,'SECOND COURSE:');  /* Assign prompt data      */
CALL ASDFLD(3,18,15,1,14,2);      /* Protected alpha field   */
CALL ASCPUT(3,14,'THIRD  COURSE:');  /* Assign prompt data      */
CALL PSLSS(0,'ADMITALC',193);     /* Load Italic symbol set  */   A
                                  /* into hardware PS store  */
DO I=1 TO 3;                      /****************************/
CALL ASFCOL(I,2);                 /* First 3 fields are to be*/
CALL ASFPSS(I,193);               /* red and in Italic style */
END;                              /****************************/
CALL ASDFLD(11,6,30,1,1,0);       /****************************/
CALL ASDFLD(12,12,30,1,1,0);      /* Define 3 input fields   */
CALL ASDFLD(13,18,30,1,1,0);      /****************************/

DCL ASR_ATTS(81) FIXED BIN(31) INIT(                              B
/* Fld_id row col dpt wdt typ int colr sym_set
     101,  6, 35,  1, 25,  2,  1,  0,    193, /******************/
     102,  7, 35,  1, 25,  2,  1,  0,    193, /* Attributes for */
     103,  8, 35,  1, 25,  2,  1,  0,    193, /* multiple       */
     104, 12, 35,  1, 25,  2,  1,  0,    193, /* definition of  */
     105, 13, 35,  1, 25,  2,  1,  0,    193, /* 9 fields       */
     106, 14, 35,  1, 25,  2,  1,  0,    193, /******************/
     107, 18, 35,  1, 25,  2,  1,  0,    193,
     108, 19, 35,  1, 25,  2,  1,  0,    193,
     109, 20, 35,  1, 25,  2,  1,  0,    193);
CALL ASRFMT(9,9,ASR_ATTS);          /*Define 9 protected fields*/   C
CALL ASCPUT(101,25,'(1) PRAWN COCKTAIL  $1.80'); /***************/
CALL ASCPUT(102,25,'(2) FISH SOUP       $2.30'); /*             */
CALL ASCPUT(103,25,'(3) GAME PATE       $2.20'); /* Assign data */
CALL ASCPUT(104,25,'(1) T-BONE STEAK    $9.80'); /*             */
CALL ASCPUT(105,25,'(2) SOLE MEUNIERE  $10.50'); /*             */
CALL ASCPUT(106,25,'(3) JUGGED HARE     $7.50'); /*             */
CALL ASCPUT(107,25,'(1) FRESH PINEAPPLE $1.75'); /*             */
CALL ASCPUT(108,25,'(2) PROFITEROLES    $2.40'); /*             */
CALL ASCPUT(109,25,'(3) DESSERT TROLLEY $1.75'); /***************/
```

Figure 76 (Part 2 of 4). MENU programming example

```
CALL ASDFLD(50,24,14,1,42,2);
CALL ASFPSS(50,193);                /* Italic symbol set       */   D
CALL ASCPUT(50,42,'THE BILL FOR YOUR SELECTED MENU WOULD BE:-');
CALL ASDFLD(51,24,58,1,6,2);
CALL ASFCOL(51,6);                  /* Bill total in yellow    */
CALL ASFPSS(51,193);                /* Italic symbol set       */
CALL ASDFLD(52,30,17,1,42,2);
CALL ASCPUT(52,42,'SELECT ANOTHER MENU OR PRESS PFKEY TO EXIT');
CALL ASDFLD(53,26,10,1,60,2);
CALL ASFCOL(53,5);          /* Wine recommendation in turquoise */
CALL ASFPSS(53,193);                /* Italic symbol set       */

     /*****************************************/
     /* TOP OF LOOP TO PROCESS MENU REQUESTS    */
     /*****************************************/
OUTPUT:;
CALL ASFCUR(11,1,1);   /* Position cursor in first-course field */
DO I=11 TO 13;
CALL ASCPUT(I,1,' ');         /* Reset menu selections to blank */
END;
CALL ASREAD(TYPE,MOD,COUNT);  /* Output to screen & await reply */   E
IF TYPE¬=0 THEN GOTO ENDIT;   /* End run if interrupt not ENTER */
IF COUNT=0 THEN GOTO OUTPUT;               /* No fields entered */
DO I=101 TO 109;                  /******************************/
CALL ASFCOL(I,4);                 /* Reset all dishes to green */
END;                              /******************************/
BILL=0;                       /* Initialize amount of bill to 0 */
WINE=4;        /* Select champagne unless a main dish is chosen */

/*************************/
/* QUERY MODIFIED FIELDS */
/*************************/
CALL ASQMOD(3,FIELD_IDS,LENG,I_LENG);                              F
DO I=1 TO 3;                              /* Process the order  */   G
                                          /* a course at a time */

IF FIELD_IDS(I)=0                         /* < 3 dishes ordered */
  THEN GOTO ORDER_COMPLETE;
CALL ASCGET(FIELD_IDS(I),1,CHAR1);   /* Retrieve dish selection */
IF (CHAR1='1')|(CHAR1='2')|(CHAR1='3')           /* Valid entry */
  THEN DO;
BILL=BILL+COSTS(FIELD_IDS(I)-10,CHAR1); /* Add dish cost to bill*/
CALL ASFCOL(100+CHAR1+3*(FIELD_IDS(I)-11),6);                      H
                                       /*Chosen item to yellow */
IF FIELD_IDS(I)=12
  THEN WINE=CHAR1;                /* Wine to match main course */
END; /* VALID ENTRY */
END; /* I-LOOP */
```

Figure 76 (Part 3 of 4). MENU programming example

```
ORDER_COMPLETE::;
CALL ASCPUT(53,60,'MAY WE RECOMMEND A BOTTLE OF '||BOTTLE(WINE)||'?');
BILLPIC=BILL;       /* Convert amount of bill to character form */
CALL ASCPUT(51,6,BILLPIC);  /* Assign total bill to alpha field */
GOTO OUTPUT;                      /* Branch back to ASREAD call */
ENDIT: CALL FSTERM ;                         /* Terminate GDDM */
%INCLUDE ADMUPINA;                     /* Include declarations */
%INCLUDE ADMUPINF;                     /* of GDDM entry points */
%INCLUDE ADMUPING;
%INCLUDE ADMUPINP;
END MENU;
```

Figure 76 (Part 4 of 4). MENU programming example

Note: A version of the MENU program written in the REXX language, (ERXMENU EXEC), is supplied as a sample program with GDDM.

Concepts introduced by the MENU program

Defining multiple alphanumeric fields

The ASRFMT call enables you to define several alphanumeric fields at the same time. Rather than use 9 different ASDFLD calls and any of the twelve other calls that set the attributes of alphanumeric fields, the ASRFMT call at **C** defines 9 fields and all their attributes in one call. Any attributes that are not defined here or are given the value, 0 take default settings.

When fields are logically connected, it is useful to group their definitions together in this way.

If any of the field identifiers match those of existing fields, the existing fields are replaced by the new ones.

The ASDFMT call can also be used to make multiple field definitions but it deletes all existing alphanumeric fields on the current page before creating any new ones. For more information on either of these calls, see the *GDDM Base Application Programming Reference* book.

Setting the field attributes as you define the field

The advantages of using ASFRMT rather than ASDFLD to define fields are demonstrated in the program. The italic symbol set known to GDDM as 193 is loaded using the PSLSS call at **A**. This symbol set is chosen in the multiple declaration of attributes at **B** for the nine fields of the menu defined at **C**. Any text entered into these fields is displayed or printed as italic letters.

The same symbol set is used in the other fields of the menu but because they are defined using the ASDFLD call, attributes such as the symbol set must be set explicitly for each field. For example, at **D** the ASFPSS call specifies that the text in field 50 uses the italic symbol set. For more detailed information on the use of symbol sets, see Chapter 12, “Using symbol sets” on page 233.

Discovering how many fields on the current page were modified

The logic of the MENU program allows for the end user to enter data in up to three fields or in no field at all. There are two techniques used to discover how many fields have been modified. The simpler way is to inspect the COUNT variable returned by ASREAD. COUNT holds the number of procedural alphanumeric fields that have been modified. Immediately after the ASREAD call at **E**, the MENU program tests to see if COUNT is zero. If it is, the program reissues the ASREAD and awaits input.

The other method involves using the ASQNMF call like this:

```
DCL COUNT FIXED BIN(31);
CALL ASQNMF(COUNT);
```

The number of modified fields is then returned in COUNT.

Identifying which fields have been modified

Even when you know how many fields have been modified since the last ASREAD was issued, you still may not know exactly which fields have changed. At **F** in the program, an ASQMOD call returns information on up to three fields modified since the previous ASQMOD or ASREAD. One of the parameters returned by the call contains the actual field identifiers of the modified fields.

If there are fewer modified fields than the number requested on the ASQMOD call, the remaining entries in the passed arrays are set to zero by GDDM. The program loop that processes the meal order makes use of this fact. If it finds that a returned field identifier is zero, it knows that the meal order has been completed. For more information on ASQMOD, see the *GDDM Base Application Programming Reference* book.

Choosing advantageous field identifiers

As demonstrated at **B** in the example, it helps to choose sequential identifiers for related alphanumeric fields. Your program can process them in a loop, as at **G**.

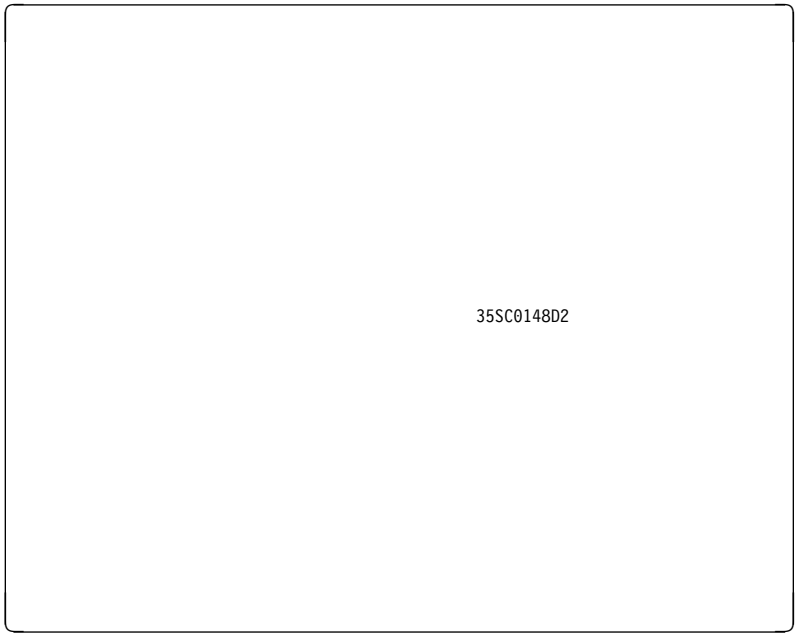


Figure 77. Output from "Menu" program

Redefining the attributes of existing fields

At **H** in the example, the color attribute for the fields containing the selected dishes is changed by means of an ASFCOL call. This is the best way of making that change given that the program needs to be dynamic. However, if you wanted to redefine several attributes of one or more alphanumeric fields in your program all at once, you could then use the ASRATT call. This call takes the exact same form as ASFRMT and ASDRMT but the field identifiers supplied with ASRATT must all belong to already existing fields. It would be an error to try to use the ASRATT call to define new alphanumeric fields.

If you do not specify an attribute, such as highlighting, then a default attribute value is taken.

Resetting the default value of an alphanumeric field attribute

You may want to change the default value for some of the attributes. If most of your fields are to appear in reverse video, then making this the default highlight would save you several calls to ASFHLT. This is how to do it:

```
DCL DEFAULT_ATTRS(5) FIXED BIN(31) INIT(
     2,              /* Type       = protected      */
     1,              /* Brightness = normal         */
     1,              /* Color      = blue           */
     0,              /* Symbol set = default        */
     2 );            /* Highlight  = reverse  video */

CALL ASDFLT(5,DEFAULT_ATTRS);  /* Define new default values for */
                               /* the first five attributes     */
```

All fields subsequently defined are subject to the new defaults.

Note: The new defaults apply only to the current page. If you only want to reset the fifth attribute, you must also set the first four (they can be set to –1 if the existing default is satisfactory). The remainder of the attributes (6 through 12, in this case), revert to the normal default.

Processing an alphanumeric field with changed status

The status of an alphanumeric field denotes whether it is modified or unmodified. The status of a field is set to modified whenever the operator of the display screen types data into the field or selects it with the light-pen. The field may also be arbitrarily set to modified status by issuing an ASFMOD call, for example:

```
CALL ASFMOD(23,1);                /* Mark field 23 as modified */
```

Fields return to unmodified status in one of two ways. An ASQMOD call can return information on them, leaving them marked as unmodified, or an ASFMOD call can be issued with the second parameter set to 0.

ASQMOD and ASQNMF do not return information just on the fields that were modified as a result of the most recent ASREAD. They return all fields (in the current page) that are marked as modified, whether they came in on the most recent ASREAD or some previous one.

This is a typical sequence:

```
CALL ASREAD(TYPE,MOD,COUNT);        /* Issue read to the device */

 /*The operator types into, six fields                          */

CALL ASQMOD(4,F_IDS,LENGS,ILENGS);  /* Query first four          */
                                    /* modified fields          */

 /* The four queried fields are now marked unmodified,          */
 /* leaving just two fields marked as modified.                 */

CALL ASREAD(TYPE,MOD,COUNT);        /* Issue second read        */

 /* The operator now types into N fields, one of which was      */
 /* already marked as modified. There are now N+1 modified      */
 /* fields                                                      */

CALL ASFMOD(13,0);                  /* Program requests field   */
                                    /* 13 be marked unmodified  */

 /* Field 13 was one of the fields into which the operator      */
 /* typed. It is now marked unmodified, leaving N modified      */
 /* fields.                                                     */
```

```
CALL ASQMOD(4,F_IDS,LENGS,ILENGS);  /* Query first four        */
                                    /* modified fields          */

 /* The four queried fields are now marked unmodified,          */
 /* leaving N-4 fields marked as modified.                      */

CALL ASQNMF(NUM);

 /* The variable NUM is set to the number of modified fields    */
 /* that remain, namely N-4                                     */

DO I=1 TO (NUM+3)/4;

  CALL ASQMOD(4,F_IDS,LENGS,ILENGS); /* Query next four         */
                                    /* modified fields          */

 /* Details of the rest of the modified fields are returned,    */
 /* four at a time                                              */

END;
```

Processing light-pen fields

Light-pen fields are alphanumeric fields that may be selected by the operator with the selector pen feature. In this case, “select” means “mark as modified.”

Descriptive data may be assigned to light-pen fields (using ASCPUT), but the fields are always protected on the screen so that no data may be entered. The first data position of each row of a light-pen field contains a **designator character**. This is a visible indication of whether a field has been selected.

There are four different types of light-pen field:

- **Light-pen select fields** have initially a ? in the first data position of every row. The ? designator characters are inserted by GDDM and replace the first data byte. So, if you want a prompt of `TOTAL PROFITS`, you must issue:

  ```
  CALL ASCPUT(8,14,' TOTAL PROFITS');
  ```

  The field then appears on the screen as `?TOTAL PROFITS`. When the operator selects such a field with the light-pen, the ? changes into a > but no interrupt is caused.

  Several such fields may be selected (and data may be typed into fields other than light-pen fields) before the operator causes an interrupt, for instance, by pressing the ENTER key. All modified and selected fields are now returned to GDDM. See “Processing an alphanumeric field with changed status” on page 263 for information on how to process the returned fields.

  A selection of this type of field does not cause an interrupt (thereby completing a screen read), they are known as **deferred light-pen fields**.

- **Light-pen enter fields** have initially an & in the first data position of every row (again set by GDDM). When one such field is selected, an interrupt is caused immediately. The ASREAD that is satisfied by this interrupt returns with its first parameter (the type of interrupt) set to 0. In other words, the same type of

interrupt as when the ENTER key is pressed. Such fields are known as **pen-enterable fields**.

- **Light-pen attention fields** have initially a blank character in the first data position of each row (set by GDDM). They are similar to pen-enterable fields except that a different type of interrupt is caused when they are selected.

  Note: Selection of a light-pen attention field destroys the data of all unprotected fields on the screen.

  Light-pen attention fields should therefore not be mixed with alphanumeric data-entry fields.

- **General light-pen fields** may be set to any one of the previous three types by setting the designator character appropriately (for each row of the field). In other words, the program sets the designator character as part of the field's data (using ASCPUT), rather than defining the type of light-pen field explicitly and letting GDDM insert the designator characters. For example:

  ```
  CALL ASCPUT(1,14,'?TOTAL PROFITS').
  ```

Alphanumeric fields may be specified as being any of the above four types by setting the last parameter of the ASDFLD call:

```
/*    FIELD_ID, ROW, COLUMN, DEPTH, WIDTH, TYPE                    */
CALL ASDFLD(1,   3,    4,      1,     7,    3);  /* Define lightpen */
                                                  /* attention field */
```

The same parameter settings may be used to change the type of a field, using the ASFTYP call. (See “Setting the attributes of alphanumeric fields” on page 74.)

There are a few points to note on light-pen fields in general:

- Where a field has more than one row, the whole field becomes selected whichever row is addressed by the light-pen.
- The hardware imposes several restrictions on the positioning of light-pen fields:

  All light-pen fields must be at least 3 characters long.

  No light-pen field may begin in column 1.

  If there is another field to the left of the light-pen field, there must be a separation of at least four columns.

Following a screen read, the processing of light-pen fields is similar to that for other types of alphanumeric field as shown in Figure 76 on page 257. Selected fields are marked as modified and may be determined by a call to ASQMOD.

Using procedural alphanumerics for double-byte characters

Some Asian languages have so many written characters that they cannot all be stored in a conventional single-byte character set. Each character, therefore, must be represented in the 3270 data stream by a two-byte code. GDDM supports input and output of these double-byte characters but there are two important points to bear in mind if you are using them in alphanumeric applications.

Hardware Although you can program the input and output of double-byte character alphanumerics on any GDDM supported terminal or workstation, you (and the end user of the application) can only display such alphanumerics on devices such as the IBM PS/55 workstation

and the 5550 multistation. The hardware symbol set of these devices is a Kanji double-byte character set (DBCS). Alphanumeric applications on these devices use that set by default, but applications can load other Kanji sets and Hangeul sets instead.

Field width The code for a DBCS character uses twice as many bytes as the EBCDIC code for a character of the Latin alphabet. Alphanumeric fields for Japanese or Korean characters must be twice as wide as the number of characters they contain.

Note: For output, you can use DBCS graphics text (see "Using double-byte characters for graphics text" on page 244) either as an alternative to the alphanumeric output functions described here or as a way of improving the usability of your application on terminals that don't support DBCS. On such a device, the application could take the hexadecimal codes entered as an alphanumeric string by the end user and display them as DBCS graphic text characters.

Example: Routine to fill an alphanumeric field with Kanji data

This code shows how, without having a PS/55 or 5550, you can still program the output of alphanumeric Kanji characters in an application that is to be used on those devices.

```
/*****************************************************************/
/*          CREATE FIELD CONTAINING KANJI DATA                   */
/*****************************************************************/

DCL KANJI_DATA5 CHAR(10);      /* String for 5 Kanji characters */

    /* FIELD-ID  ROW  COLUMN  DEPTH  WIDTH  TYPE */
CALL ASDFLD(77,   7,    7,      1,     10,   0);/* 10-byte field*/   A

CALL ASFPSS(77,248);            /* Specify Kanji character set */   B

KANJI_DATA5='45EE46CC48F243CD4391'X;       /* Assign 5 two-byte */   C
                                           /* Kanji characters  */

CALL ASCPUT(77,10,KANJI_DATA5);    /* Put characters into field */   D
```

Figure 78. Routine to place double-byte characters in an alphanumeric field.

Points illustrated by the example

You must express the width of the field in terms of bytes not characters. At **A** the alphanumeric field defined is 10 bytes wide. This is wide enough to hold 5 double-byte characters.

The ASFPSS call at **B** specifies the Kanji character set, the hardware character set of the device on which the program is to run, as the primary symbol set for the field. The character set has the special identifier 248 (X'F8').

Because the keyboard of the programming device cannot type Kanji characters, the hexadecimal code of each character must be placed in the string parameter of the ASCPUT call. At **C** in the example, the hexadecimal codes are placed in the

KANJI_DATA5 variable which is then passed as the string parameter of ASCPUT at **D** .

Any double-byte characters input from the keyboard by the end user of such an application can be returned to the program by an ASCGET call.

Performing output of strings mixing single- and double-byte characters

Some devices, such as the IBM PS/55 workstation and 5550 multistation, support mixed alphanumeric fields. On these devices applications can use the ASCPUT call to present strings containing both single- and double-byte characters in **any** field.

On other devices, you must define a **mixed string** attribute in your program for each field into which you put a mixed string. You can do this using the ASFSEN call (see “Example: Routine to mix SBCS and DBCS data in an alphanumeric field” on page 268). You also need to specify the MIXSOSI GDDM default parameter (see “GDDM default required for Kanji and Simplified Chinese” on page 246). The terminal displays the double-byte codes in hexadecimal.

On devices that support DBCS, when mixed data is entered into an alphanumeric field, either as input or output, the DBCS substrings are delimited by special control characters. The shift-out (SO) control character marks the beginning of a DBCS substring and the shift-in (SI) control character marks the end. (If no DBCS characters are entered, no SO/SI codes are inserted.) Within any field, only SO/SI pairs are allowed, in that order.

If you are writing a program on an ordinary GDDM-supported terminal to generate mixed-string alphanumeric output on a device that can display or print DBCS characters, you must code these special control characters in the ASCPUT call. This is shown by the example in Figure 79 on page 268.

Example: Routine to mix SBCS and DBCS data in an alphanumeric field

```
MIX: PROC OPTIONS(MAIN);

DCL (I,J,K) FIXED BIN(31);         /* Parameters for ASREAD     */

DCL (SO,SI) CHAR(1);               /* Shift-out & shift-in      */
SO='0E'X;                          /* Set shift-out codepoint   */  A
SI='0F'X;                          /* Set shift-in codepoint    */  A

DCL MIXED_DATA50 CHARACTER(52);      /* String for mixed chars  */

CALL FSINIT;                       /* Initialize GDDM           */

DCL (Kanji1, Kanji2, Kanji3, Kanji4, Kanji5, Kanji6) CHAR(2);

Kanji1='45EE'X;                    /* Ward 68     Position 238  */
Kanji2='46CC'X;                    /* Ward 70     Position 204  */
Kanji3='48F2'X;                    /* Ward 72     Position 242  */
Kanji4='43CD'X;                    /* Ward 67     Position 205  */
Kanji5='4358'X;                    /* Ward 67     Position 88   */
Kanji6='4391'X;                    /* Ward 67     Position 145  */

/*****************************************************************/
/*           CREATE FIELD FOR MIXED KANJI AND LATIN DATA         */
/*****************************************************************/
    /* FIELD-ID  ROW  COLUMN  DEPTH  WIDTH  TYPE */
CALL ASDFLD(81,   8,    1,      1,     52,   0);/* 52-byte field*/  B
CALL ASFSEN(81,2); /* String attribute of mixed without position*/  C

                                   /*                    bytes*/
MIXED_DATA52='In Japanese' ||      /*12 Latin characters  12  */
             SO ||                 /* Shift-out            1  */
             Kanji1 ||             /* 6 Kanji characters  12  */
             Kanji2 ||
             Kanji3 ||
             Kanji4 ||
             Kanji5 ||
             Kanji6 ||
             SI ||                        /* Shift-in      1  */
             ' is pronounced eisuji deta'; /* Latin Chars  26  */
                                          /*  total bytes 52  */

CALL ASCPUT(81,52,MIXED_DATA52);   /* Put mixed string into   */
                                   /* alphanumeric field      */
CALL ASREAD(I,J,K);
CALL FSTERM;                /* Terminate GDDM                     */

%INCLUDE ADMUPINA;          /* GDDM  Entry declarations           */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END MIX;
```

Figure 79. Routine mixing single- and double-byte characters in an alphanumeric field.

Points illustrated by the example

At the points marked **A**, the hexadecimal codes of the DBCS delimiter characters are assigned into variables. The SO code is `X'0E'` and the SI is `X'0F'`.

The definition of the alphanumeric field at **B** takes account of the SO and SI control characters, inserting an extra byte for each one. It also allows two bytes for each Kanji character and one for each Latin character so it is wide enough to hold the MIXED_DATA50 variable.

At **C**, the ASFSEN call defines field 81 as a mixed field in which GDDM inserts a blank space between the SBCS and DBCS substrings when the field is displayed or printed. This is the “mixed with position” option. It is one of the two ways a mixed string can treat the SO/SI codes, when you use ASFSEN to set the mixed string attribute for a field.

The other mixed-field option, “mixed without position”, allows the field to be displayed or printed without inserting any blank spaces between the SBCS and DBCS substrings.

The non-null length of the string as displayed on the device may be less than the length of the field as defined in your program. The difference is equal to the number of SO/SI codes that you have in the field declaration. The end of the string on the display is padded with nulls.

If you use the mixed-string attribute of “mixed without position,” then for calls ASCCOL, ASCHLT, ASCSS, the character attributes corresponding to SO/SI control codes have no effect.

Note: Once a field has been defined as mixed, you cannot use the ASFPSS call to define it as a DBCS field.

Returning the mixed-string contents of a user-input field to the application

When the terminal operator changes input mode from SBCS to DBCS, enters one or more DBCS characters and then changes back to SBCS, the SO and SI control characters are inserted on either side of each DBCS substring.

A program that asks the user to type their surname into a field in both Kanji and Latin might make use of these control characters. It might be useful to include both versions in one “NAME” input field but, for presentation purposes, not to have them merging on the display. In such a case, a mixed-field attribute setting of “mixed with position” would display the SI character following the Kanji name as a blank space. This would separate it from the Latin version following it.

If you define the mixed-string attribute of a field as “mixed with position” then on input, ASCGET returns data of a similar structure to the output. It contains SO characters wherever the operator shifted out of single-byte mode and SI characters where there was a shift back in. There are DBCS codes between these shifts, and ordinary single-byte codes elsewhere.

If you use the mixed-string attribute of “mixed without position,” then on input, the non-null length of the string as displayed on the device may be less than the length of the field as returned to your program. The difference is equal to the number of SO and SI codes that GDDM inserts around the DBCS portions of the string.

When retrieving data from an input field using calls ASCGET, ASQCOL, ASQHLT, or ASQSS, you must therefore allow for the generated SO/SI codes.

For this reason, it is good practice to issue an ASQLEN call before issuing an ASCGET call, so that your program can ascertain how much storage is needed to hold the returned data. Here is a typical call:

```
CALL ASQLEN(81,FIELD_LENGTH,INPUT_LENGTH,SCREEN_LENGTH);
```

For mixed alphanumeric input on a device which does not have a DBCS keyboard, the operator can shift into and out of double-byte mode by entering a special emulation character; by default, the double quote character. While in double-byte mode, the operator must enter a string of double-byte hexadecimal codes. GDDM returns these codes preceded by an SO code and followed by an SI in place of the emulation character. You can change the emulation character with the SOSIEMC GDDM default parameter (see the *GDDM Base Application Programming Reference* book).

Cursor position with mixed-without-position fields

For mixed fields that are mixed-without-position, you may want to position the cursor in terms of the byte position in the field contents in your program, as opposed to the column position in the field on the screen. Here is an example of ASFCUR that places the cursor at byte 14, the start of the SBCS string “LATIN AGAIN” in the example code on page 268:

```
CALL ASFCUR(81,-1,14);  /* SET CURSOR AT BYTE 14 IN FIELD 81/*
```

On the screen, the cursor would appear under column 12 of the mixed field, because the two bytes containing the SO/SI codes do not appear.

Similarly, you can use ASQCUR to query the position of the cursor in terms of the byte position in the field in your program:

```
CALL ASQCUR(2,81,ROW,COLUMN);  /* QUERY POSITION OF CURSOR */
```

For detailed information on these and other calls used here, see the *GDDM Base Application Programming Reference* book.

Field outlining on the IBM 5550 Multistation

In applications intended for use on the 5550 multistation you can use the ASFBDY call to draw a complete or partial outline around a field. You specify outlining with the ASFBDY call:

```
CALL ASFBDY(33,15);  /*Type 15 outline (complete box) for field 33*/
```

If you want to group several adjacent fields in one box, you can specify the appropriate partial outline for each of the fields to make up a larger perimeter.

Improving the performance of procedural alphanumerics applications

There are two basic rules to follow for minimizing processor usage when using GDDM procedural alphanumerics:

1. Perform operations on the alphanumeric fields in screen order.
2. The field identifier should be related to the screen position of the field.

In other words, if you want to define fields, for instance, do it sequentially. Start at the top row, define the fields from left to right, and then do the same for each successive row until you reach the right-hand end of the bottom row. The field identifier should increase as you work down the screen. Experiments have shown that using this technique can reduce processor usage by up to 40% compared with assigning unrelated field identifiers and accessing fields in random order.

Other miscellaneous points to bear in mind when programming with procedural calls:

- ASDFMT uses slightly less processing time than ASDFLD when defining fields, although this gain is probably offset by use of ADMUFO to bypass parameter checking.
- Using unnecessary parameters on ASDFMT and ASRFMT calls requires additional processor resource.
- Deep fields are slightly cheaper than having a field on every line of the display. If the data on each line is of varying length, the data stream of the deep field will be longer as padding is inserted on each line.
- Avoid defining long fields in which only the first few bytes contain data. Although the rest of the field will get padded with nulls or blanks, there is a processing cost involved in compressing the padding bytes when the data stream is transmitted to the terminal.
- Use of attributes such as highlighting, color, or reverse video, on a character rather than a field basis, involves extra work in the processor. These should be used sparingly from the human factors point of view also, if they are to be effective.

It is possible to avoid recreating an alphanumeric screen before redisplaying it within an application. This can be achieved by defining multiple pages using FSPCRT; alphanumeric panels that are going to be displayed several times within the application should each be defined on a separate page. To redisplay a particular panel, reselect that page using FSPSEL and issue an ASREAD call to display it. You must use a unique page for every alphanumeric panel that you want to keep in this way. The following example shows how to do it:

```
     .
     .
CALL FSPCRT(1000,0,0,2);    /* Define a page with an id of 1000 */
CALL FSPSEL(1000);          /* Make it the current page         */
CALL ASDFLD(1,2,5,1,16,2);  /* Define an alphanumeric field     */
                            /*              on the current page */
CALL ASCPUT(1,16,'THIS IS SCREEN 1');  /* Initialize it         */
CALL ASREAD(ATT,ATTM,COUNT);/* Display the current page         */
     .
CALL FSPCRT(1001,0,0,2);    /* Define a page with an id of 1001 */
CALL FSPSEL(1001);          /* Make it the current page         */
CALL ASDFLD(1,2,5,1,16,2);  /* Define an alphanumeric field
                                         on the current page */
CALL ASCPUT(1,16,'THIS IS SCREEN 2');  /* Initialize it         */
CALL ASREAD(ATT,ATTM,COUNT);/* Display the current page         */
     .
CALL FSPSEL(1000)           /* Reselect page 1000               */
CALL ASREAD(ATT,ATTM,COUNT);/* Display it                       */
```

Because several pages are kept, this technique increases the amount of dynamic storage that the application uses, but it is unlikely to add much to application overheads. Processor usage required for the second and subsequent displays of a panel are likely to be 30 to 40% of the original creation and display cost.

Chapter 14. GDDM high-performance alphanumerics

High-performance alphanumerics (HPA) is another way of handling alphanumerics using GDDM and is intended for complex applications that require instruction paths of minimum length within GDDM.

HPA provides greater

- Flexibility in screen building
- Ease in varying the number of lines per screen
- Ease in varying the length of fields

The application programming interface of HPA differs from that of other parts of GDDM used for creating alphanumerics. HPA provides the dynamic field definition capabilities of procedural alphanumerics combined with a “buffer” style interface and even shorter instruction-path length than mapped alphanumerics.

With procedural alphanumerics, application programs use several API calls to describe the data GDDM is to output and to determine the data entered by the device operator. In contrast, an application using HPA builds a data structure which describes all the data, and passes it to GDDM for output. The data entered by the device operator is returned to the HPA application in the same data structure. Changes to the data are shown by indicators which are part of the structure.

The relative advantages of high-performance alphanumerics and the other methods in various situations are discussed in “Comparison of the three methods of implementing alphanumeric functions” on page 56.

Note: You may not mix mapped or procedural alphanumeric field definitions with HPA field definitions on the same GDDM page.

How to use high-performance alphanumerics

To create alphanumeric fields in your application using high-performance alphanumerics you need to first set up the data structure describing the fields. In a PL/I program, this means declaring and initializing the **field list** and its associated **bundle list** and **data buffer**. You then issue an APDEF call to pass the field list, bundle list, and data buffer to GDDM and this puts the fields on the GDDM page. For an example of how to set up the HPA data structure and use the APDEF call, see “Example: Program displaying high-performance alphanumeric output” on page 274.

Declaring and initializing the field list

The field list connects all information governing the appearance of alphanumeric fields on the GDDM page. In the field list you specify most of the information on the fields and refer to the other two objects in the data structure to specify the attributes of fields and the data contained in them.

You can declare the field list in a program either as a structure or as a two-dimensional array stored in row-major order. If you use a programming language which requires two-dimensional arrays to be in column-major order, you

© Copyright IBM Corp. 1982, 1996

must exchange the rows for columns as they are shown in “Example: Program displaying high-performance alphanumeric output” on page 274.

The first row of the field list is called the header row. Here you specify general information about the list itself and about the page. On each other line you define an alphanumeric field, with elements specifying the status, size, and position of the field. One of the elements in each field definition is called the “bundle row”. This is an index to a row in the bundle list that contains a “bundle” of attributes for the field.

Declaring and initializing the bundle list

The bundle list is similar in structure to the field list. You can also declare it either as a structure or as a two-dimensional array in row-major order. In the header row, you specify information about the bundle list itself and the application can also use it to record its own data. In each of the other rows, you define a bundle of settings for attributes of the particular field with an index to that row of the bundle list in its definition.

Declaring and initializing the data buffer

You can declare the data buffer as a data area to hold the character data that you want to display in the alphanumeric fields on the GDDM page and any attributes for these characters. If you are programming in PL/I, you declare the data buffer as a character string.

The portion of the data area that contains the character data for a particular alphanumeric field is defined in that field’s definition in the field list. In each field definition in the field list, you can include an index to the start of the data area that contains the data and a measure of its length. If you wish you can also include similar indexes to parts of the data area that contain definitions of the character attributes such as color, highlighting, and symbol-set.

Mixing single-byte and double-byte character fields

You can mix single- and double-byte characters in HPA fields in the same way as with procedural alphanumerics. You must place the special shift-out control character (`X'0E'`) at the beginning of a DBCS substring and the shift in control character (`X'0F'`) at the end.

The display method, either mixed-with-position or mixed-without-position, can be specified in the bundle definition for the mixed field. For more information, see “Performing output of strings mixing single- and double-byte characters” on page 267.

Example: Program displaying high-performance alphanumeric output

The programming example in Figure 80 on page 275 demonstrates the various tasks involved in creating a page of high-performance alphanumeric output. It displays a page with four fields, one of which uses character attributes. When the ENTER key is pressed the color of the first field is changed. When PF3 is pressed the program terminates.

```
/* EXHPA     - SAMPLE CHARACTER ATTRIBUTES    */
EXHPA:    PROC;

DCL TYP      FIXED BIN(31) STATIC INIT(1);
DCL VAL      FIXED BIN(31) STATIC INIT(3);
DCL CNT      FIXED BIN(31) STATIC INIT(0);
DCL ENDKEY   BIT(1) INIT('0'B);

DCL FL( 5,10)FIXED BIN(15) STATIC INIT                              A
             /*STA  DEP  WID  CSR  CSC                          */
              ( 1,   5,  10,   2,   5,   0,   0,   0,   0,   0,    B
             /*STA  ROW  COL  WID  BLR  DAI  ACT  COI  HII  SSI*/
                1,   2,   5,   4,   2,   1,   4,   0,   0,   0,    C
                1,   4,  10,  11,   2,   5,  11,   0,   0,   0,    D
                1,   6,  15,  13,   3,  16,  13,   0,   0,   0,
                1,   8,  20,   3,   4,  29,   3,  32,  35,  38);   E
DCL BL( 4,10) FIXED BIN(15) STATIC INIT
             /*STA  DEP  WID                            */
              ( 0,   4,  10,   0,   0,   0,   0,   0,   0,   0,
             /*STA  PRS  TYP  VAL  COL  VAL  BDY  VAL  PSS  VAL*/
                0,   3,   8,   0,  24,   3,  72,   1,   0,   0,
                0,   4,   8,   0,  24,   6,  72,  15,  32,  80,
                0,   4,   8,   0,  24,   3,  72,   7,  88,  7);    F
DCL DB    CHAR(40) STATIC INIT
          ('HighPerformanceAlphanumericsAPI356124 &&');            G

CALL FSINIT;

CALL PSLSS(0,'ADMITALC',80);    /*   load a symbol set         */

/*   Define a field list for the panel                           */
CALL APDEF(1,DIM(FL,1),DIM(FL,2),FL,LENGTH(DB),DB,                  H
             DIM(BL,1),DIM(BL,2),BL,6);
     /* This uses the built in DIM feature of PL/I. */
     /* where DIM is the dimension of the array.    */
     /* It could have been coded as:                */
     /* CALL APDEF(1,5,10,FL,40,DB,4,10,BL,6);      */             H
```

Figure 80 (Part 1 of 2). Program to display high-performance alphanumeric output

```
/*  Display panel and process selection until END key pressed  */
DO   UNTIL(ENDKEY);
     CALL ASREAD(TYP,VAL,CNT);           /* Display panel      */
     SELECT;                             /* Process selection  */   I
       WHEN(TYP=1 & (VAL=3 | VAL=15))    /* END key            */
            ENDKEY = '1'B;

       WHEN(TYP=0) DO;  /* ENTER key alone - change field color*/   J
            BL(1,1) = 1;             /* Set bundle list status */   K
            BL(2,1) = 1;       /* Set bundle definition status */   L
            BL(2,6) = MOD(BL(2,6)+1,8);  /* change color value */   M
            END;

       OTHERWISE CALL FSALRM;          /* Error condition      */
       END;
     END;
CALL PSRSS(80);                        /* Release a symbol set */

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINP;

END EXHPA;
```

Figure 80 (Part 2 of 2). Program to display high-performance alphanumeric output

Points illustrated by the EXHPA program

Setting the status of field list

At **A**, the field list is declared as a two-dimensional array called FL with a depth of 5 and a width of 10. It is then initialized and the header line at **B** gives the field list a status of 1, which means that only those fields in the field list that also have a status of 1 are to be processed. The status of the field list is the value you get by ORing together all the individual field-status values from the field definitions. There are different status values to indicate other actions such as, creation, redefinition, deletion, output, and input of field lists.

Setting the depth, width, and cursor position of the field list

The header line also states the depth and width of the field list and specifies that the cursor be placed in the fifth column of the second row of the GDDM page. This is at the beginning of the first alphanumeric field as can be seen from the ROW and COL elements of the field definition at **C**

Defining fields

In addition to the field status and location elements, each of the fields defined is given:

- A width setting
- An index to a particular row of the bundle list
  (Several fields can have indexes to the same row)

- An index to that part of the data buffer where the data for the field begins
- A measure of the actual length of the data.

Setting the optional character attributes

Only in the fourth field definition at **E** are any of the character attributes set. Here, indexes are given to parts of the data buffer that contain attribute settings for the color, highlighting, and symbol set of the data in the fourth field. GDDM finds the color attribute setting for the field beginning at byte 32 of the data buffer, the highlight attribute setting beginning at byte 35, and the symbol set to be used for data in the field beginning at byte 38.

```
DCL DB    CHAR(40) STATIC INIT
          ('HighPerformanceAlphanumericsAPI356124 &&');
                                          ▲  ▲  ▲  ▲
                                         29 32 35 38
```

At these bytes, the color code ‘356’, the highlight code ‘124’, and the symbol-set code ‘&&’ are specified. (The blank character at byte 38 specifies inheritance of the field symbol set and the two “&” characters (X'50' or decimal 80) request the use of a symbol set with identifier 80.)

Specifying attributes for the alphanumeric fields

The elements in the header row of the bundle list specify the status of the bundle list as unchanged, that the number of rows in it be 4 (including the header), and that 10 elements in each row are to be used by GDDM.

Because some of the fields share the same attributes, there is no need for each individual row of the field list to have a corresponding row in the bundle list. For this reason, the lines marked **C** and **D** in the example both have an index to line 2 of the bundle row. The bundle definition at **F** specifies an unchanged (0) status for the bundle. It specifies that four attributes are to be defined for the bundle. These attributes are then specified by pairs of values; one identifying the attribute (for instance, 8–field type), the other specifying which value is to be set for that attribute (in this case 0–unprotected alphanumeric).

The attributes 24 (color), 72 (field outlining), and 88 (character reply mode) are defined in this way for the fourth bundle.

Setting up the data buffer

At **G** in the program in Figure 80 on page 275, the data buffer is declared as a string of forty characters. In some other languages, such as C/370, where character strings are not a recognized data type, you may need to declare the data buffer as an array of type character.

Defining a field list for the GDDM page

Either of the APDEF calls marked **H** in the program can be used to define a field list to GDDM and combine with it the information specified in the bundle list and data buffer.

Choosing modes of data transfer for HPA applications

The APDEF calls marked **H** specify the value 6 in the “mode” parameter. This value is the sum of two values, 2 and 4, which means that a combination of those modes will be used for the application. The value 2 specifies that **locate** mode will be used to transfer data between GDDM and the application program. The value 4 specifies that the cursor-row and cursor-column fields of the field list header are to contain the position of the alphanumeric cursor.

Data can be transferred between GDDM and the application program, by means of either the **move** or the **locate** mode.

If you do not specify locate mode in your application, move mode is used. The choice of move mode or locate mode affects any application data embedded in the field list, data buffer, or bundle list. If move mode is used, this application data is copied by GDDM on APDEF and subsequent calls to APMOD. The value copied on the most recent APDEF or APMOD call is returned by GDDM on APQRY. This means that any changes made after APDEF or APMOD are lost on the next call to APQRY.

Move mode: With move mode, the field list, data buffer, and bundle list are copied by GDDM when the APDEF call is issued. Subsequent processing of input and output by GDDM uses the GDDM copies. When the application needs to retrieve updates made by the device operator, or to modify the fields, it must query the field list, data buffer, and bundle list by calling APQRY. This returns copies of the field list, data buffer, and bundle list held by GDDM. When the application has modified the field list, data buffer, and bundle list, it must pass the modified versions back to GDDM by calling APMOD.

Locate mode: In the example in Figure 80 on page 275 locate mode is used, so the application data is not copied by GDDM.

GDDM does not copy the field list, data buffer, or bundle list. Subsequent processing of output and input by GDDM uses the copies in application storage. The application must not release the storage that these objects occupy until the field list has been deleted. The contents of the field list, data buffer, and bundle list must be valid whenever GDDM is called. When using locate mode, it is not necessary to call APQRY to determine device operator updates, nor to call APMOD to inform GDDM of changes made by the application. Using locate mode, an application can define a screen of alphanumerics, display it, and get the input with just one GDDM call (ASREAD).

Changing the attributes of the alphanumeric field

At **I**, the program tests what kind of user interrupt has been made. If the interrupt matches the case at **J**, the ENTER key has been pressed and the color attribute value for the alphanumeric field is changed. When any attributes of the field are to be changed the status of the bundle-list (at **K**) and the status of each bundle definition with values to be changed (at **L**) must be set to 1. This indicates to the application that they need to be processed. The actual value of the color attribute is altered at **M**. After the next ASREAD or other I/O call, the status values are reset to 0.

Returning HPA user input to the application

To retrieve end-user updates to the page of alphanumeric fields following an I/O operation:

- If using move mode, retrieve the field list, data buffer, and bundle list from GDDM by calling APQRY.
- Test the field list status input indicator to determine if any fields have been updated by the device operator. If they have, test field definition input indicators to determine which fields have been changed, and process the input found in the data buffer.
- If the alphanumerics are not to be reshown, they should be cleared by calling APDEL.

Displaying alphanumeric fields again

Once an application has read in a page of alphanumeric fields as input, it may still require the end user to see that page. The application can reshow the fields if you follow these steps:

1. Reset the field list status input indicator and the field definition input indicators.
2. Change the data or character attributes in the data buffer as required, and set the corresponding output indicators in the field definition and header status.
3. Change the bundle definitions in the bundle list as required, and set the corresponding bundle definition and header status indicators.
4. Change the field definitions in the field list as required, and set the corresponding status indicators to specify what has changed.
5. If using move mode, return the modified field list, data buffer, and bundle list to GDDM by calling APMOD.
6. Call ASREAD, or another GDDM I/O call as required.

Field-list update rules

The rules for altering a field list are:

- After each I/O, the application must reset the input indicators, which indicate end user updates to the alphanumeric fields. Otherwise, the application cannot detect further updates on a subsequent I/O operation.
- Field row, field column, and field width may not be changed, except when using a previously unused field definition entry to define a new field. Fields may be defined in any order, but must not overlap. They may wrap from row to row, but must not extend beyond the end of the page.
- Bundle row may be changed by the application, in which case the application must also set the output indicator to indicate to GDDM that this is changed. It is not necessary to set this indicator if only the bundle definition has changed and the field definition has not changed.
- If the character index, color index, highlight index, symbol-set index or actual length are changed, the application must set the Output indicator to show GDDM that the field has changed and is therefore to be output on the next I/O.

- When a previously unused field definition is activated, the process indicator and the create indicator must be set by the application. These indicators should never be reset by the application, only by GDDM.
- If an existing field is to be deleted, the field delete indicator should be set by the application. This indicator should never be reset by the application, only by GDDM, and the field definition entry may only be reused to define a new field after GDDM has reset the entire field status element.
- Changes to any field definition status indicator may also require changes to the corresponding header status indicator. The header status must always be set to the value obtained by ORing together all the field status elements.

Data buffer update rule

If a character data area, or a character attribute data area is modified, then the output indicators in the corresponding field definition status and field list status must be set.

Bundle list update rule

If a bundle definition is modified, the bundle changed indicator in the bundle definition status and bundle list status must be set.

Dynamic fields

You can use HPA to create dynamic alphanumeric fields by reserving space in the field list, data buffer, and bundle list for the fields to be added later. If you want to reserve field definitions in the field list, do not set the field-status indicator in the field-definition row. Leave the process indicator off. Reserved space is left in the data buffer if you do not refer to it in any existing field definitions. You can reserve bundle definitions in the bundle list by setting the number of type-and-value pairs to zero, or by using the dummy attribute type.

You may, at some stage, need to enlarge the structures. When this is the case, use the APMOD call to change the size of the field list, data buffer, or bundle list and also their location if using locate mode. Your application must allocate new, larger data structures to replace the old ones, initialize them from the old ones (or by calling APQRY), call APMOD to define the enlarged versions to GDDM, and throw the old ones away.

Note: If APMOD is used in this way, any differences between the contents of the old and new structures must be indicated by change indicators as defined in the rules above.

Programming HPA with interpreted languages

In general, locate mode cannot be used by applications written in interpreted languages such as APL or REXX. When using these languages move mode must be used.

It is recommended that you only use REXX to prototype HPA applications because the instruction-path length for HPA is significant in the REXX interpreter interface to GDDM. See also the restrictions on shared storage below.

Read-only storage

In certain circumstances, you may want to use HPA with the field list, data buffer, or bundle list in read-only storage, for instance, if your application is to be used by many users at the same time. In such cases, it is more efficient if you place fixed panel layouts in shared storage. To use HPA from read-only storage, ensure that GDDM does not write to it by adhering to the rules below:

- Neither APDEF nor APMOD alters the storage of the field list, data buffer, or bundle list.
- In move mode, ASREAD does not alter the objects in user storage.
- In locate mode, ASREAD only alters:

The field list	If any of the create, delete, or output indicators are set, or if any field is unprotected or has the MDT attribute
The data buffer	If any field is unprotected or has the MDT attribute
The bundle list	If any status indicators are set.

Shared storage

When using locate mode, it is possible for an application to define more than one field list using the same storage. Field lists, data buffers, and bundle lists could all share storage. The rules for sharing storage are:

- Field lists may not share storage unless they are read only. See the section “Read-only storage.”
- Bundle lists may be shared between more than one field list on the same device. They may not be shared between field lists on different devices unless they are read only.
- Data buffers may be shared between more than one field list only if unprotected data areas (that is, data areas corresponding to fields that are unprotected or have the MDT attribute) are not shared.

Note: Violations of these rules are not detected, and the results of such a violation are undefined.

Choosing between validation and improved performance

To enable GDDM to be used as the device driver for fully tested program products, it is necessary to be able to run HPA programs without validation. (Validation is not necessary for tested applications and switching it off again improves performance significantly.)

Validation checks the API parameters such as identifiers and lengths, as well as the field list, data buffer, and bundle list. The field list, data buffer, and bundle list are not validated during the API call processing as other parameters are, instead they are validated during processing for each I/O call involving the GDDM page.

Validation is controlled by the FRCEVAL external default.

If you suspect a tested application (such as a program product) of containing a bug, you can turn validation on to determine whether the application or GDDM is at fault

by specifying `ADMMDFT FRCEVAL=YES` in your external defaults file. This default may not be specified in the external defaults module, on SPINIT calls, or by API call.

The default setting of FRCEVAL is NO. When FRCEVAL=YES is specified, the validation indicator in the mode parameter of APDEF is overridden so that validation is always performed. The other indicators in the mode parameter are not affected.

You can also control validation using the APDEF and APMOD API calls, but once the application has been developed and fully tested, you should recode these calls to turn validation off.

If you choose to write an application using HPA without validation, you do so at your own risk. Incorrect use may result in device checks.

Chapter 15. Mapped alphanumerics

This section describes another method by which programmers who also have the GDDM Interactive Map Definition (GDDM-IMD) licensed program can include alphanumeric-text functions in their applications. The information here does not apply to graphics-only devices such as plotters.

Using predefined screen formats for alphanumeric applications

If you create a display that includes procedural alphanumeric data, you must format it. In other words, you must define the positions and attributes of all the alphanumeric fields on the screen or printer page. Mapping is an alternative technique for doing this. Essentially, it means you define the format of a display before its execution, instead of doing it dynamically in your application program.

The predefined format is called a **map**. It is most convenient to create maps interactively. GDDM provides a product for this, called Interactive Map Definition (GDDM-IMD). Using GDDM-IMD, you can indicate on a screen where all the alphanumeric fields in a display are to start and end, and you can enter codes to define their attributes, such as their color and whether they are protected.

Information about how to use GDDM-IMD is provided in two places: within GDDM-IMD itself and in the *GDDM Interactive Map Definition* book.

In addition to the position of a field and its attributes, you can define its content to GDDM-IMD and arrange that neither the application program nor the terminal operator can alter it. Such fields are called **constant data fields**. Fields that can be altered are called **variable data fields**.

GDDM-IMD generates a coded form of the maps you create, to be used by GDDM when your program sends data to, and receives data from, the terminal. On output, GDDM builds the display you require by merging variable data supplied by your program with the formatting information and constant data contained in the map. On input, GDDM separates the variable data from the rest of the input, and passes it to your program; the variable data contains any input typed in by the operator.

You can find an example of a simple mapping program in Figure 81 on page 286 and its associated display in Figure 83 on page 288.

The means by which the variable data is passed to and from the application by GDDM is a program variable called an **application data structure (ADS)**. There is an example at **A** in Figure 81. You specify to GDDM-IMD which fields are to appear in the ADS, and thus define them to be variable data fields. The ADS is the only means by which the program can alter fields, so those not represented in it are constant data fields.

The ADS in the example contains only data without any details of its presentation to the end user. It demonstrates the major advantage of GDDM mapping; you concentrate on data processing when you write your program, and leave the presentation entirely to GDDM.

The facilities described in Chapter 5, “Basic procedural alphanumerics” and Chapter 13, “Advanced procedural alphanumerics” are known as **procedural**

© Copyright IBM Corp. 1982, 1996

alphanumerics, to distinguish them from the mapping facilities. As with procedural alphanumerics, GDDM mapping uses hardware cells and fields. Mapping is therefore restricted to display units and printers of the IBM 3270 family, and to system printers. In a dual-screen configuration of the IBM 3270-PC/GX workstation, mapped data appears on the alphanumerics screen. On the 5080 and 6090 Graphics Systems, it appears on the 3270 screen.

GDDM-IMD provides default values for many of the items that it asks you to specify. The maps used in the examples in this guide were created using the GDDM-IMD defaults, except where stated otherwise.

Full GDDM mapping support is limited to programs written in PL/I, COBOL, and System/370 Assembler, because only these languages enable application data structures to be used. FORTRAN programmers, however, can use maps. They can transmit the constant data, and are not precluded from supplying variable data by means other than GDDM-IMD created structures.

If you write applications in C/370 or REXX you cannot use the map generated by GDDM-IMD directly. You must first create the map for another language and then convert the ADS into a form than can be passed to a C/370 or REXX program. To help REXX programmers with this task, GDDM provides a utility program called ERXMSVAR. The ERXMSVAR utility converts an ADS generated by GDDM-IMD to data suitable for a REXX exec.

Note: Do not specify a '.' as part of the "prefix" parameter passed to ERXMSVAR. This could cause the resulting variables to be treated as REXX stem variables and their values could be substituted and changed. Another separator character, such as "_", is safer.

GDDM also provides a subcommand for REXX programs to transfer values from variables into the ADS and vice versa. See the descriptions of the ERXMSVAR exec and the GXSET subcommand in the *GDDM Base Application Programming Reference* book.

The ERXORDER sample exec shows how to use the the GXSET MSADS and GXSET MSVARS subcommands to move mapping data, created using the ERXMSVAR utility, between the application and GDDM-REXX. ERXORDER is described in "REXX sample programs" on page 528.

For an example of how to code an application data structure for use with a C/370 program, see "ADS conversion for mapping applications written in C/370" on page 288.

A simple mapping application

The MAPEX01 program, shown in Figure 81 on page 286 could be used by an enterprise to record orders for its products made by customers. Initially it displays the fields shown in Figure 83 on page 288.

The end user is required to enter a customer number and an invoice number. The program checks that they are numeric. If so, the program does some further processing (not shown here, but it could be, for instance, to display another map for the end user to enter some more information.) If they are not both numeric, the program puts a message on the screen and enables the end user to correct the

error. The position of the message (the line below the heading) was defined when the map was created.

Tasks illustrated by the MAPEX01 program

The MAPEX01 program illustrates several basic concepts of GDDM mapping.

Creating the map: GDDM-IMD provides a quick-path tutorial to introduce you to its facilities. The tutorial tells you how to create a simple map, called ORDER1. The MAPEX01 program uses this map. Its field definitions are shown in Figure 82 on page 285.

```
           1         2         3         4         5         6         7         8
  ----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 |                                #ACME ORDER ENTRY                                |
 |¬                                                                               #|
 |                                                                                 |
 |     CUSTOMER                                          INVOICE                   |
 +      NUMBER:¬      #                                 NUMBER:¬    #              +
 |                                                                                 |
 |                                                                                 |
 |                                                                                 |
 |                                                                                 |
10     ENTER CUSTOMER NUMBER (#PF3#OR#PF15#TO RETURN )                            10
  ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
           0         0         0         0         0         0         0         0
```

Figure 82. Field definitions for map used by MAPEX01

The *GDDM Interactive Map Definition* book tells you how to invoke GDDM-IMD, and how to use the quick-path tutorial. It is probably best if you work through the tutorial before reading this description of the MAPEX01 program.

An overview tying together map creation and program development is given in “Steps in creating a mapping application” on page 291.

Application data structure: When the map was created using GDDM-IMD, three variable data fields were defined: two unprotected, for the customer and invoice numbers; and one, protected, for the error message.

The program accesses these fields using the application data structure, **A**. You do not have to code declarations for Application Data Structures (ADS) in your programs: they are generated by GDDM-IMD. This example, and all the others in this section and Chapter 16, “Variations on a map” on page 307, show the ADS declarations in full to make the programs easier to follow. All you need to do is declare a name for the structure, and then include the generated declaration for the fields. Instead of the lines marked **A**, you would code, in PL/I:

```
DECLARE 1 CUSTINV,              /* Application Data Structure */

%INCLUDE ORDER1;
```

The name under which the generated declarations are stored is the same as the map name, in this case ORDER1. In PL/I, the source code generated by GDDM-IMD contains a variable the value of which is the length of the ADS. In the example, it is called ORDER1_ASLENGTH.

In this application, all the variable data fields must be blank when the map is displayed for the first time. The whole ADS is therefore initially cleared, at **B**.

```
MAPEX01:  PROC OPTIONS (MAIN);

DECLARE 1 CUSTINV,                /* Application Data Structure */  A
         10 MESSAGE                       CHAR(78),                 A
         10 CUSTNO                        CHAR(5),                  A
         10 INVNO                         CHAR(4),                  A
         ORDER1_ASLENGTH                  FIXED BIN(31,0) STATIC
                                          INIT(87);                 A
DECLARE (ATTYPE,ATVAL) FIXED BINARY(31,0);

CALL FSINIT;                      /* Initialize GDDM.           */
CUSTINV = '';                     /* Clear the ADS              */  B
LOOP:                             /* Use MSREAD to display the  */
                                  /* map, and wait for input.   */
CALL MSREAD('ACME00D6',           /* Mapgroup                   */  C
            'ORDER1',             /* Map                        */
            ORDER1_ASLENGTH,      /* Specify length of ADS      */
            CUSTINV,              /* Specify name of ADS        */
            ATTYPE,               /* Set to attention type ...  */
            ATVAL);               /* ... and value by GDDM      */
IF ATTYPE=1 & (ATVAL=3 | ATVAL=15) /* Operator pressed end key?*/
  THEN GO TO FIN;

IF VERIFY(CUSTNO,'0123456789') = 0  /* Are CUSTNO and          */  D
 & VERIFY(INVNO,'0123456789') = 0   /* INVNO numeric?          */  D
  THEN DO;
    /*  .  */                       /* Process CUSTNO and INVNO   */  E
    /*  .  */
    /*  .  */
  MESSAGE = ' ';                    /* Clear any existing message */  F
  END;
  ELSE MESSAGE = 'Invalid Number';  /* If CUSTNO or INVNO not  */  G
                                    /* numeric, set up message.*/

GO TO LOOP;                       /* Redisplay the map and data */
FIN:

CALL FSTERM;                     /* Terminate GDDM.            */
%INCLUDE ADMUPINF;               /* GDDM entry declarations    */
%INCLUDE ADMUPINM;
END MAPEX01;
```

Figure 81. Source code of MAPEX01

Output and input: The GDDM call that handles mapped I/O is MSREAD, shown at **C**. MSREAD sends the map to the terminal, and then waits for the operator to cause an interrupt, such as pressing the ENTER key. In other words, MSREAD both transmits output to the terminal and reads input from it.

MSREAD has six parameters:

- The first, ACME00D6, is the name of the **mapgroup** to which the map belongs. Every map belongs to a mapgroup. When you create a map with GDDM-IMD, you must specify the name of its mapgroup. GDDM-IMD adds a two-character suffix to the name you specify for the mapgroup to identify the device class for which the mapgroup is generated.

- The second parameter is the name of the map, ORDER1 in the example.
- The third parameter is the length of the ADS. Here, the GDDM-IMD generated variable, ORDER1_ASLENGTH, is specified.
- The fourth parameter is the name declared for the ADS, CUSTINV in this example.
- The fifth and sixth parameters are set by GDDM to indicate the type of interrupt received from the terminal. They have the same meanings as the first two parameters of ASREAD. Full details of the possible values for all these parameters are given in the *GDDM Base Application Programming Reference* book.

The MSREAD call merges the variable data from the ADS with the map created by GDDM-IMD, and sends the result to the terminal. In the example, the variable data is all-blank, so the initial display consists of only the constant data fields of the map. Figure 83 on page 288 shows this initial display. When a reply is received from the terminal, GDDM copies any data entered by the operator into the ADS.

The program ends when the fifth and sixth parameters of MSREAD indicate that the operator has pressed PF3 or PF15.

Checking input data: In statement **D**, the program checks the fields CUSTNO and INVNO to verify that they contain all-numeric data. If they do, the example does nothing, but a real production program would have statements at **E** to process them.

At **G**, the example handles invalid input. It puts text into the error message field. The program does not alter the contents of the CUSTNO and INVNO fields, so the next execution of the MSREAD returns them to the operator exactly as entered. The only change the operator sees is the appearance of the error message. The operator can correct the error and resubmit the input to the application.

The error message field is cleared at **F**, to ensure that no message is displayed when the next input is solicited.

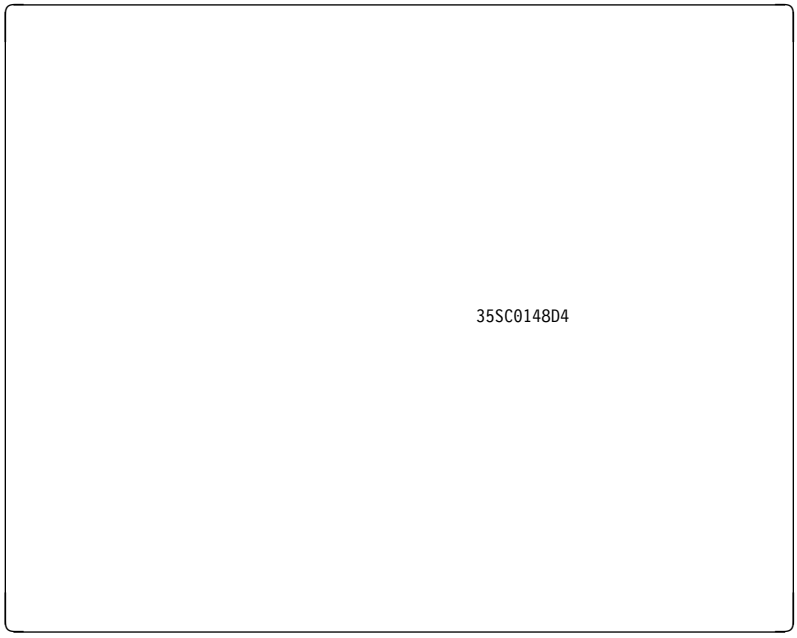


Figure 83. Initial display of MAPEX01

Compilation and execution of a mapping application program

After you have created an Application Data Structure using GDDM-IMD, you must store it in a library. More information is given in “Steps in creating a mapping application” on page 291 and in the *GDDM Interactive Map Definition* book. To compile a mapping program like the one in Figure 81 on page 286, you must make the library available to the compiler. Under CMS, the following commands make the ADS available (together with the GDDM entry point declarations in ADMLIB), and then compile the program:

```
GLOBAL MACLIB ACMEADS ADMLIB
PLIOPT MAPEX01 (INCLUDE
```

ACMEADS is the name of the macro library in which the ADS for the map ORDER1 is stored.

The commands to execute the program are the same as described in “How to compile and run a PL/I GDDM program” on page 14.

ADS conversion for mapping applications written in C/370

If you want to include mapped alphanumeric functions in a C/370 application, you can use the GDDM-IMD product to create the maps as you would for a PL/I program. You can then convert the application data structure generated for PL/I to one suitable for use with C/370.

Here is an example of how you might change the declaration of the ADS in the MAPEX01 program for C/370.

```
struct {                /* Application Data Structure */
        char message[78];
        char custno[5];
        char invno[4];
       } custinv;
int order1_l = 87;
        .                   .
        .                   .
msread("ACME00D6",
       "ORDER1",
        order1_l,
        (char *) &custinv
        &attype,
        &atval);
```

A mapping application that sets up a dialog with the end user

MSREAD is limited to simple output and input of a single map. For complex dialogs, such as ones that require more than one map in a display, the various functions of MSREAD must be performed separately, using a different call for each function. For comparison, the MAPEX02 program in Figure 84 on page 290 shows how the program in Figure 81 on page 286 would be coded using these individual calls.

The functions and calls are:

1. Create a GDDM page to contain one or more maps.

   ```
   CALL MSPCRT(1,-1,-1,'ACME00D6');
   ```

   Procedural alphanumerics can be used on a page created with MSPCRT, provided the procedural fields do not overlap with any mapped field, as defined below. Maps cannot be used on any page created with FSPCRT.

2. Format an area of the page by putting a map onto it.

   ```
   CALL MSDFLD(1,-1,-1,'ORDER1');
   ```

   A mapped area of a page is similar in many respects to a procedural alphanumeric field, and is known as a **mapped field**.

3. Copy data from the ADS into the variable data alphanumeric fields contained in the mapped field.

   ```
   CALL MSPUT(1,0,ORDER1_ASLENGTH,CUSTINV);
   ```

4. Send the page to the terminal and wait for input from it.

   ```
   CALL ASREAD(ATTYPE,ATVAL,COUNT);
   ```

   This is the same call as is used to send procedural alphanumerics to the terminal but its function is slightly changed when used in mapping applications. The third parameter returns the number of maps changed by the end user, not the number of alphanumeric fields. You can also send a mapped page to the terminal with an FSFRCE or a GSREAD call.

5. Extract data from the mapped field and put it into the ADS.

   ```
   CALL MSGET(1,0,ORDER1_ASLENGTH,CUSTINV);
   ```

```
MAPEX02:  PROC OPTIONS (MAIN);
DECLARE 1 CUSTINV,              /* Application Data Structure */
         10 MESSAGE                       CHAR(78),
         10 CUSTNO                        CHAR(5),
         10 INVNO                         CHAR(4),
         ORDER1_ASLENGTH                  FIXED BIN(31,0) STATIC
                                          INIT(87);
DECLARE (ATTYPE,ATVAL,COUNT) FIXED BIN(31);
DECLARE WRITE FIXED BIN(31) INIT(0); /* MSPUT write operation */  A

CALL FSINIT;                    /* Initialize GDDM.           */
CUSTINV = '';                   /* Clear the ADS              */  B
CALL MSPCRT(1,                  /* Create page with id = 1.   */
            -1,                 /* Use mapgroup-defined page  */
            -1,                 /* Width and depth.           */
            'ACME00D6');        /* Specify name of mapgroup.  */
CALL MSDFLD(1,                  /* Format an area of the page.*/  C
            -1,                 /* Use the map-defined row    */
            -1,                 /* and column positions.      */
            'ORDER1');          /* Specify name of map.       */
LOOP:
CALL MSPUT(1,                   /* Put ADS data into map.     */  D
           WRITE,               /* Use all ADS data (write=0).*/
           ORDER1_ASLENGTH,     /* Specify length of ADS.     */
           CUSTINV);            /* Specify name of ADS.       */
CALL ASREAD(ATTYPE,             /* Output the current page, & */
            ATVAL,              /* wait for operator input.   */
            COUNT);
IF ATTYPE=1 & (ATVAL=3 | ATVAL=15)/* Operator pressed end key?*/
  THEN GO TO FIN;
CALL MSGET(1,0,                 /* Get variable data from map.*/
           ORDER1_ASLENGTH,     /* Specify length of ADS.     */
           CUSTINV);            /* Specify name of ADS.       */
IF VERIFY(CUSTNO,'0123456789') = 0   /* Are CUSTNO and        */
 & VERIFY(INVNO,'0123456789') = 0    /* INVNO numeric?        */
  THEN DO;
    /*  .  */                   /* Process CUSTNO and INVNO   */
    /*  .  */
    /*  .  */
   MESSAGE = ' ';               /* Clear any existing message */  F
   END;
   ELSE MESSAGE = 'INVALID NUMBER';/* If CUSTNO or INVNO not  */  G
                                   /* numeric, set up message.*/
GO TO LOOP;                     /* Redisplay the map and data */

FIN:
CALL FSTERM;                    /* Terminate GDDM.            */
%INCLUDE ADMUPINA;              /* GDDM entry declarations    */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINM;
END MAPEX02;
```

Figure 84. Source code of MAPEX02

Why you do not always need to call MSPUT

An MSPUT call transfers data from the ADS to the mapped field. Sometimes, this step is not required. It is, in fact, unnecessary in the program shown in Figure 84 on page 290.

When the map was created, no initial character string value was explicitly assigned to any of the variable data fields, so GDDM-IMD assigned default initial values of all blanks. The MSDFLD call at **C** initializes the variable data fields to their default values, in addition to mapping an area of the page.

The program clears fields in the ADS at **B**, and copies them into the variable data fields by the MSPUT at **D**. Because the MSDFLD call had already initialized the fields to blanks, the MSPUT is unnecessary.

In general terms, an MSPUT call is unnecessary when all the variable data fields are initially to contain their default values.

A typical mapping cycle

The diagram in Figure 85 on page 292 shows some of the major steps that a typical mapping program goes through.

First, the application executes an MSPCRT call to create a mapped page. The page is given the identifier 4, and is associated with a mapgroup called MAPGRPD6. The mapgroup has three maps in it, called MAP99, MAP100, and MAP101.

An MSDFLD call puts the map called MAP99 onto the page. MAP99 contains two constant data fields, the values of which are NAME: and SALARY:, and two variable data fields. The program puts the variable data “J SMITH” and “12345” into this map’s ADS, called ADS99, and then executes an MSPUT call to copy the data into the variable fields on the page. A second MSDFLD call puts a second map, MAP101, onto the page. This contains just the constant data “XXXXXXXX”. The third map in the mapgroup, MAP100, is not used in this execution of the program.

An ASREAD call sends the page to the terminal, and waits for operator input. When this arrives, GDDM updates the page. The application accesses the input by executing an MSGET call to copy the variable data from the page into the ADS.

Steps in creating a mapping application

This is a step-by-step summary of the major operations required to implement a mapping application. To understand it fully, you need familiarity with GDDM-IMD to at least the level provided by the quick-path tutorial.

1. **Allocate the files required to hold the ADSs and the generated mapgroups, if you are using GDDM-IMD under TSO.** More information is given later in this section, and full details of the files are given in the *GDDM Interactive Map Definition* book.

   If you are using GDDM-IMD under CMS or CICS, ignore this step.

2. **Create the required mapgroup(s) and map(s) using GDDM-IMD:**

   a. Create the mapgroup using the mapgroup editor.

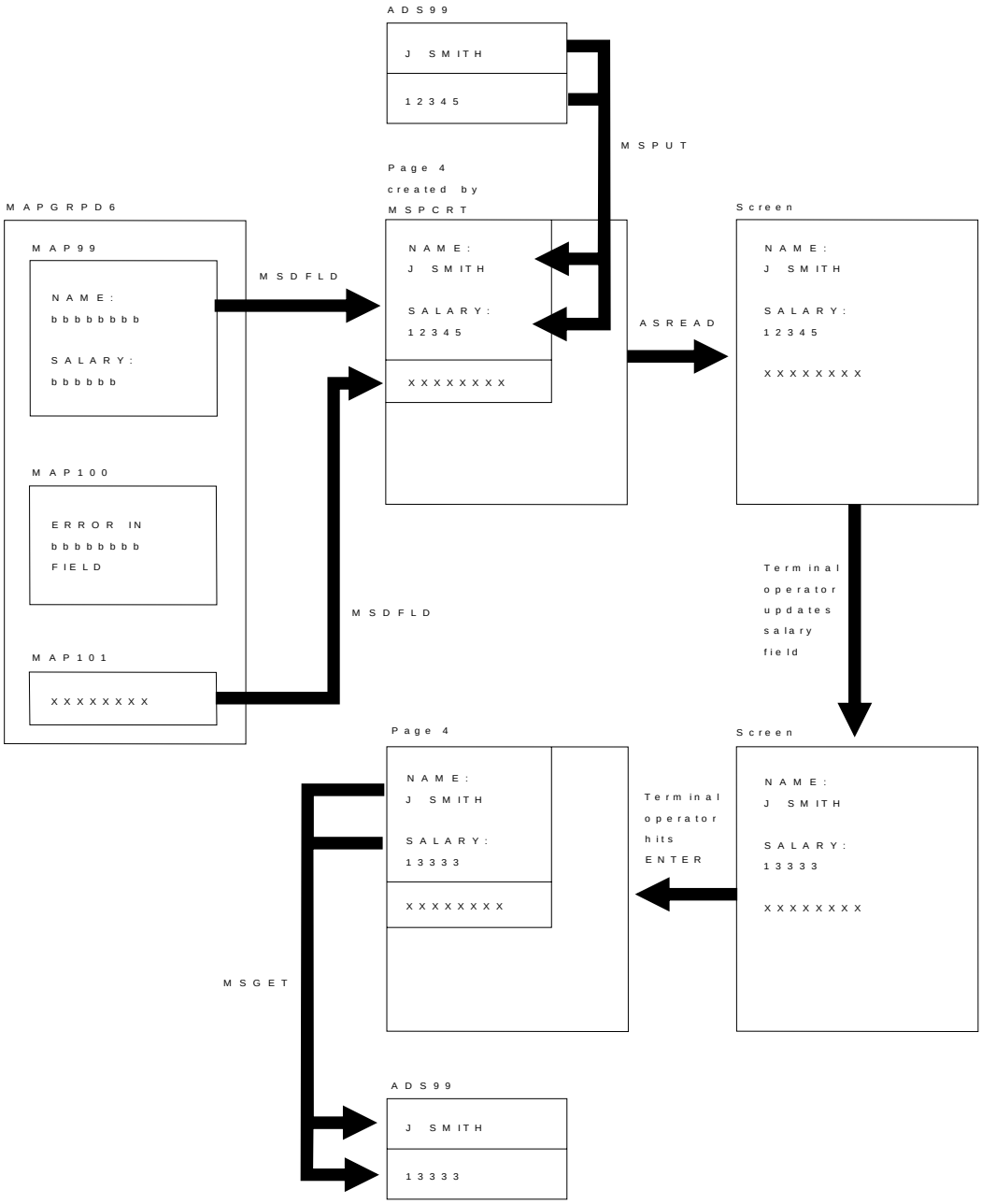


Figure 85. Typical cycle of mapping operations

A simple application such as MAPEX01, in Figure 81 on page 286 requires only one mapgroup, containing only one map. But often you need several maps in a mapgroup, as explained in “Using more than one map to present and process alphanumeric information” on page 296. And you may need several mapgroups to provide an application with several different basic types of presentation.

When creating a mapgroup, you supply information that applies to the presentation as a whole, such as a **device class** specifying the type of device on which it is to appear, and the **presentation area,** in rows and columns, that it occupies.

For instance, if your application is to run on an IBM 3427-G terminal, you would probably use a device class of D6. The D tells GDDM-IMD that the device is a display unit, and the 6 that you require a presentation area of 32 rows by 80 columns. This presentation area occupies the whole of the screen. A full list of device classes is given in the *GDDM Interactive Map Definition* book.

b. Create one or more maps using the map editor. For each map:

   1) Define the map characteristics, such as its size and its default position within the presentation area.

   2) Define the position of each field in the map. At this stage you can also type character string values into the fields. If you type nothing into a field, GDDM-IMD assigns an all-blank character string value to it. The typed-in value or the GDDM-IMD assigned string of blanks is known as the field's **default** or **initial data**.

   3) Define the attributes of each field in the map.

      At this stage, you can use the TEST command to display the map and check the position, attributes, and default data of the fields.

   4) Name the variable data fields. The names you supply are used in the application data structure. In GDDM-IMD, the naming is called **linking**. If you do not link a field by giving it a name, then neither the application program nor the terminal operator can alter its value. It is not advisable to have unlinked variable data fields in your maps.

   5) Review the application data structure. In this step, GDDM-IMD displays information about the application data structure that it generates, and allows you to amend it.

c. Generate the mapgroup.

   In this process, GDDM-IMD generates a coded representation of the mapgroup, for GDDM to use during execution. At the same time, GDDM-IMD generates an application data structure for each map in the mapgroup, for you to include in your source code. The name of the generated mapgroup includes a device-dependent two-character suffix, as explained in "Device-independence for mapped-alphanumeric applications" on page 304.

   GDDM-IMD enables you to display and review the maps in the mapgroup by performing a test generation. The test is particularly useful for multimap mapgroups: you can check the complete presentation, as GDDM-IMD combines specified maps into a single display in their correct positions on the screen. After a satisfactory test generation, you need to do the real mapgroup generation.

   During the real generation, the ADSs and mapgroups are written to files by GDDM-IMD, in ways that depend on the subsystem under which GDDM-IMD is running. The list below is a summary; more information is given in the *GDDM Interactive Map Definition* book.

   - Under CMS, the ADSs go to files with file names the same as the names of the maps they represent, and with a default file type of COPY. The generated mapgroup goes to a file with a file name comprising the mapgroup name plus the device suffix, and with a default file type of ADMGGMAP. GDDM-IMD creates these files.

- Under TSO, the ADSs go to a partitioned data set for which GDDM-IMD uses a default ddname of ADMGNADS, and member names the same as the names of the maps they represent. The generated mapgroup goes to a partitioned data set for which GDDM-IMD uses a default ddname of ADMGGMAP, and a member name comprising the mapgroup name plus the suffix.

  You must ensure that commands allocating the two ddnames to suitable partitioned data sets are executed **before** GDDM-IMD is invoked. The required data-set characteristics are given in the *GDDM Interactive Map Definition* book.

- Under CICS, the ADSs go to a transient data queue with the default name of ADMG. The queue must have a destination defined for it in the CICS destination control table (DCT); this is usually done when GDDM is installed. The output to the queue is in a form that makes it suitable for transferring to a partitioned data set using the IEBUPDTE program for CICS/ESA, or to a library using the MAINT program for CICS/VSE. The members of the partitioned data set or the books in the library have the same names as the maps that the ADSs represent.

  The generated mapgroup goes to a file. The file must be defined in the CICS file control table (FCT), the default FCT name being ADMF. This definition is usually done when GDDM is installed.

d. Convert ADS for other languages

   If you are writing a mapping program in REXX, you need to invoke the ERXMSVAR utility program at this point to convert the application data structure to a form usable by the interpreter. ERXMSVAR is described in the *GDDM Base Application Programming Reference* book.

   For an example of how to convert the ADS for use on a C/370 program, see “ADS conversion for mapping applications written in C/370” on page 288.

3. Put the ADSs into your source code.

You are recommended to use %INCLUDE statements (in PL/I) or COPY statements (in COBOL and Assembler) in your source code to do this.

You could, instead, copy the ADSs directly into your program using the editing facilities that you employ to create the source code. If you do so, you must reedit the source whenever you change the ADS.

In REXX execs, you **must** use this method to include in your program the ADS variables that ERXMSVAR stores in a file or data set with a filetype or member name of GDDMCOPY. (ERXMSVAR sets these variables to blank, so if you require initial values in the mapped fields, you must enter them into the variables yourself.)

In theory, you could choose to code the ADSs yourself as part of the source code. Unless you use only the most basic functions of mapping, this is not advised. Most mapping functions require rather complex ADSs that are difficult to code without errors. These ADSs, and the functions they support, are described in Chapter 16, “Variations on a map” on page 307.

4. Include mapping API calls in your program

Issue the MSPCRT call to create a page that is to contain mapped alphanumeric fields. If you specify a value of –1 for the depth and width of the

page, GDDM takes the depth and width from the mapgroup specified as the fourth parameter of the call.

With the MSDFLD call you define each mapped field on the page. You can also code –1 for the depth and width of each field and GDDM takes these values from the map named in the fourth parameter of the call.

Use the MSPUT call to place data from the ADS into the mapped fields and MSGET to transfer user input from the fields into the ADS. Because these calls require a single variable identifying the ADS, special action is required in REXX execs because ERXMSVAR converts the ADS into several REXX variables.

You can use the GXSET MSADS subcommand to gather together these variables for use with the MSPUT call and use the GXSET MSVARS subcommand to convert the ADS from the fields back to REXX variables when they have been altered by the end user.

5. **Compile or assemble the program.**

   ADSs that are to be included in the program with %INCLUDE or COPY statements must, like any secondary source code, be in a source library before compilation. The actions you need to take are subsystem-dependent:

   - Under CMS, you need to transfer the ADSs from the file into which GDDM-IMD puts them to a macro library defined by you. Before compilation, you must execute a GLOBAL MACLIB command to make the macro library available to the compiler or assembler.
   - Under TSO, GDDM-IMD puts ADSs into a suitable partitioned data set when you generate them, and all you need to do is make this available to the compiler or assembler. You do so in the same way as for any other secondary source code, typically by an ALLOCATE command.
   - For CICS applications, you need to execute the IEBUPDTE program (CICS/ESA) or MAINT program (CICS/VSE) to transfer the ADSs to a partitioned data set or library defined by you. Before compilation, you must make the partitioned data set or library available to the compiler or assembler. You do so in the same way as for any other secondary source code. Typical ways are with a suitable DD statement (CICS/ESA) or ASSIGN statement (CICS/VSE) if you compile in batch mode, or a suitable ALLOCATE or GLOBAL MACLIB command if you compile under TSO or CMS.

6. **Execute the program.**

   GDDM finds the generated mapgroups required by the program with no further action by you (except under IMS, when you must import the generated mapgroups).

   The various GDDM mapping calls that a typical program may need to execute are summarized in “A mapping application that sets up a dialog with the end user” on page 289.

Changing existing maps

The preceding list is intended to help you create maps. When you alter an existing map, you can use the list to check that you do not omit any essential operations. You should take particular care to remember:

- To regenerate the mapgroup after altering a map.
- If you use GDDM-IMD under CMS or CICS, to update the secondary source library with any new or changed ADSs.
- If the ADS has changed, to recompile (or reassemble) the program.

Using more than one map to present and process alphanumeric information

For most applications, you will find it necessary to two or more maps to format the screen. For instance, you might want to use one map to allow the end user to ask a question, and then a second to give the answer. You would probably want the first map to remain on the screen while the second one is displayed.

GDDM enables you to put many maps onto a page provided there is space, and the maps do not overlap with each other. The section “Using maps with positions fixed by GDDM-IMD” gives further information.

In some applications, you may need to repeat a set of fields several times. For a data-entry application, for instance, you might need to fill the screen with many copies of a single set of input fields, each set being one or a few lines deep. You can do this by having several copies of the same map.

For such applications, GDDM-IMD enables you to define **floating maps**. You do not have to calculate where to put these on the page. GDDM positions a floating map at the next available location, rather than at a location specified either to GDDM-IMD, or to GDDM by the program.

An example of using floating maps is given in “Using several maps that position themselves relative to each other” on page 297.

Using maps with positions fixed by GDDM-IMD

The GDDM-IMD operations necessary to create two or more fixed maps are the same as for a single map, except that you go through the map editor steps twice. You do not generate the mapgroup until you have defined all of the maps in it. To put several maps onto a page (or several instances of the same map), your program simply executes an MSDFLD for each one.

Note: Although GDDM-IMD enables you to define maps that overlap, it is an error, if your application issues MSDFLD calls for two such fields. There are three ways you can avoid such errors.

- Take care when using GDDM-IMD, to specify the size and position of maps so that they do not overlap
- Use the MSDFLD call in the application to override the IMD-defined position of maps that overlap with those already on the page
- Only use overlapping maps on different pages of the application

The mapgroup test facility of the GDDM-IMD mapgroup generation step is particularly useful for multimap mapgroups. It diagnoses inadvertently overlapped maps, and lets you check the spacing between maps, and the alignments between fields in different maps.

When you test the mapgroup, you must specify which maps are to be put into the test display, and the order in which they are to be processed. The order is more important with floating maps, though it may affect some aspects of a display containing only fixed maps, such as where the cursor appears initially. It is advisable, therefore, to specify the maps in the order in which you expect to refer to them with MSDFLD calls in your application program.

At execution time, your program can override the specified position of any fixed map by giving an explicit row and column number in the MSDFLD call that puts it onto the GDDM page.

Using several maps that position themselves relative to each other

Creating a floating map differs from creating a fixed one only in the values you put into two fields in the Map Characteristics frame of the map editor. The fields are those in which you specify the position of the map's top left-hand corner. Instead of a number, you enter the value SAME in one of the fields. Maps with the value SAME for the column number are called **vertically floating**, and for the line number, **horizontally floating**. Either type can be fully floating or semifloating. To make a map fully floating, you specify SAME in one of the fields and NEXT in the other one. To make it semifloating, you specify SAME in one and a number in the other, this being a row or column number in relation to the start of the presentation area.

All floating maps are positioned by GDDM within the **floating area**, which is a subdivision of the presentation area. You specify its size and position on the Mapgroup Characteristics frame of the mapgroup editor. The default floating area is the whole of the default presentation area, in other words, the whole screen or printer page.

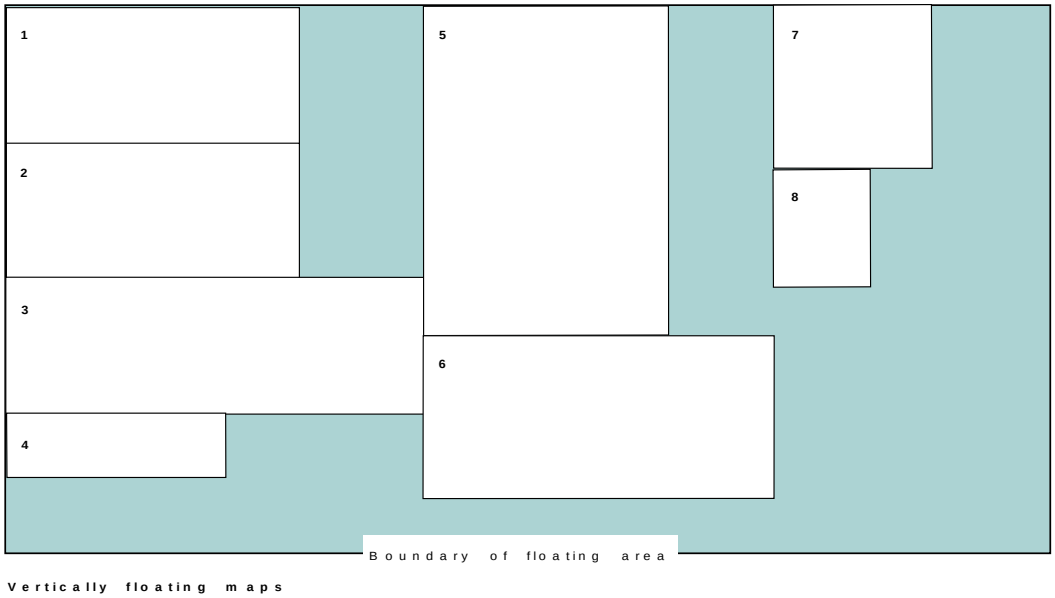
The floating maps in a floating area must be either all vertically floating or all horizontally floating.

GDDM always puts the first fully floating map at the top left of the floating area. Succeeding ones are positioned underneath the previous one if they are vertically floating, or to the right if they are horizontally floating. A fully floating map is positioned in the next available row or column; in other words, it is contiguous with the preceding map.

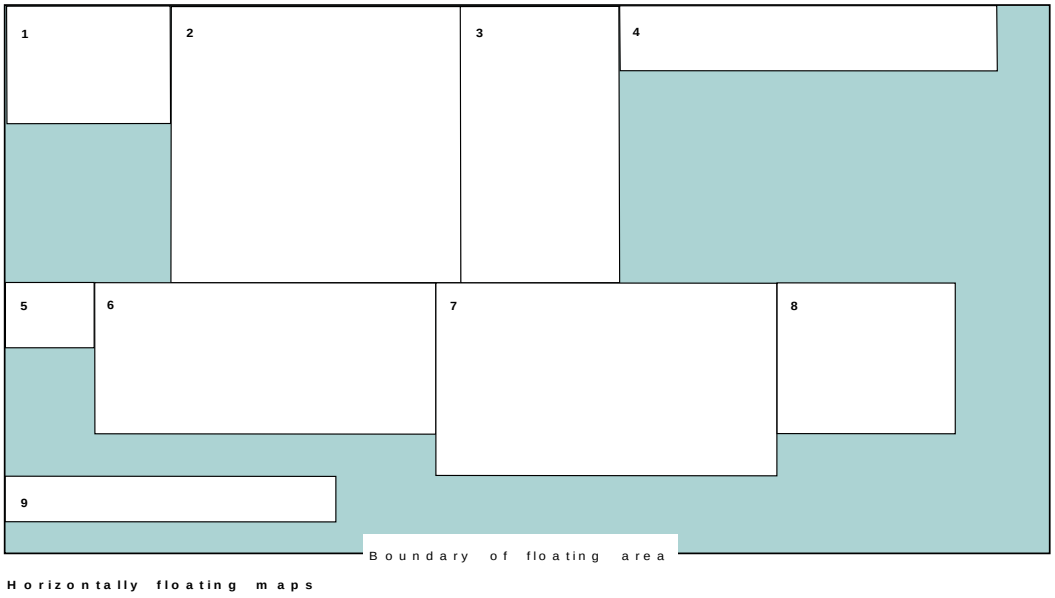
When there is insufficient space beneath a stack of vertically floating maps, a new stack is started to the right of it. Similarly, when there is insufficient space to the right of a row of horizontally floating maps, a new row is started underneath.

Positioning of fully floating maps is summarized in Figure 86 on page 298. Most applications do not use maps of such varied size as those illustrated. The sizes have been chosen to illustrate GDDM's positioning algorithm, in particular where the second and subsequent horizontal and vertical stacks are positioned.

Semifloating maps are always positioned in the specified row or column in relation to the start of the presentation area (not of the floating area). The other coordinate



Vertically floating maps



Horizontally floating maps

Figure 86. Positioning of fully floating maps

(the one specified as SAME) is the same as for the previous map. If the semifloating map is the first floating map on the page, it is put into the first column or row in the floating area.

The main use of semifloating maps is to force the map to always appear at the head of a column or start of a row (by specifying a row or column of 1), or the bottom of a column or end of a row (by specifying a row or column close to that of the bottom or right-hand edge of the floating area).

If you use fixed and floating maps on the same page, you can allow fixed maps to intrude into the floating area, but it is your responsibility to ensure that no fixed and floating maps overlap.

If you want to fix the position of a floating map instead of allowing GDDM to do it, you can give an explicit row and column number in the MSDFLD call that puts the map onto the GDDM page.

Example of a program that uses fixed and floating maps

The MAPEX04 program in Figure 87 on page 300 could also form the basis of an application for displaying customers' orders. The program uses one fixed map and one floating map. Having put the fixed map on the screen, the program determines how many instances of the floating map the screen can accommodate. It then fills the screen with that number of instances of the floating map, in which customers' orders are displayed.

The formats of the two maps are shown in Figure 88 on page 301. A typical display is shown in Figure 89 on page 302.

It is an output-only application. For handling input data from floating maps, see “Input from multiple copies of a map” on page 303.

```
MAPEX04:  PROC;

DCL 1 HEADER,                  /* ADS for heading map        */  A
        10 FILLER_PAD                     CHAR(1),
        HEADER_ASLENGTH                   FIXED BIN(31,0) STATIC
                                          INIT(1);
DCL 1 FLOATER,                 /* ADS for floating map       */
        10 PART_NUM                       CHAR(7),
        10 DESCRIPTION                    CHAR(11),
        10 QUANTITY                       CHAR(3),
        10 UNIT_PRICE                     CHAR(6),
        10 TOTAL_PRICE                    CHAR(9),
        FLOATER_ASLENGTH                  FIXED BIN(31,0) STATIC
                                          INIT(36);
DCL (ATTYPE,ATVAL,COUNT) FIXED BIN(31);/* ASREAD arguments    */
DCL WRITE   FIXED BIN(31) INIT(0);   /* MSPUT write operation */
DCL FMAPNUM FIXED BIN(31);      /* Number of orders           */
DCL PID FIXED BIN(31);          /* Page identifier            */
DCL MID FIXED BIN(31);          /* Mapped field identifier    */

CALL FSQUPG(PID);               /* Get unique page identifier */  B
CALL MSPCRT(PID,                /* Create new page            */  C
            -1,                 /* with GDDM-IMD defined page */
            -1,                 /* width and depth,           */
            'FLOATD6');         /* for mapgroup FLOATD6.      */
MID=1;
CALL MSDFLD(MID,                /* Format header area of page */  D
            -1,                 /* at GDDM-IMD defined row    */
            -1,                 /* and column position,       */
            'HEADER');          /* using map header           */
CALL MSQFIT('FLOATER',FMAPNUM); /* How many maps to fill page?*/  E
DO MID = 2 TO FMAPNUM+1;        /* Put ORDNUM copies of       */
                                /* floating map on page.      */
  CALL MSDFLD(MID,              /* Format an area             */  F
              -1,               /* at floating row            */
              -1,               /* and column position,       */
              'FLOATER');       /* using map floater.         */
  CALL ORDERS(FLOATER);         /* Assign data to ADS         */  G
  CALL MSPUT(MID,               /* Move data to page from ADS */
             WRITE,             /* with write operation,      */
             FLOATER_ASLENGTH,  /* specifying length          */
             FLOATER);          /* and name of ADS            */
END;
CALL ASREAD(ATTYPE,             /* Display page,              */
            ATVAL,              /* and wait for operator      */
            COUNT);             /* input.                     */
CALL FSPDEL(PID);               /* Delete page before exit.   */

%INCLUDE ADMUPINA;              /* GDDM entry declarations    */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINM;
END MAPEX04;
```

Figure 87. Source code of MAPEX04

```
          1         2         3         4         5         6         7         8
 ----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
|                      #ORDER REVIEW#                                            |
| Part No.   Description   Quantity   Unit Price   Total Price                  #|
|                                                                                |
 ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
          0         0         0         0         0         0         0         0
```

```
          1         2         3         4         5         6         7         8
 ----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
|                                                                                |
|¬       #  ¬           # ¬   #      ¬      #      ¬         #                   |
|                                                                                |
 ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
          0         0         0         0         0         0         0         0
```

Figure 88. Field definitions for map used by MAPEX04

Points illustrated by the MAPEX04 programming example

The program in Figure 87 on page 300 displays a heading using a fixed map called HEADER, and formats each line of data with a floating map called FLOATER. The two ADSs have the same names as the maps, and are declared at **A**.

The data for each order is put into the ADS named FLOATER by a subroutine, called at **G**.

The example introduces several new programming techniques and facilities of GDDM.

Unique page identifier: The program is a subroutine within a larger application, so it must ensure that the identifier of the page it creates has not already been used. It obtains a unique identifier by issuing an FSQUPG call at **B**. For a detailed description of this call, see the *GDDM Base Application Programming Reference* book. The unique identifier is returned by GDDM in the variable PID, and this is specified as the page identifier in statement **C**, which creates the page to be mapped.

Positioning of floating maps: When a map has been specified to GDDM-IMD as floating, the value –1 for the row and column in an MSDFLD call means that GDDM is to choose the map's location. All floating maps are positioned within the floating area. You define this area when you create the mapgroup. In the example, –1 is specified in the MSDFLD call **F** for the map FLOATER. In the case of the mapgroup FLOATD6, to which FLOATER belongs, the floating area was defined as the whole page, apart from the lines occupied by the fixed map HEADER. FLOATER was defined to be vertically floating, so successive instances of it are positioned one beneath the other within the floating area.

Number of floating maps: The floating area is just filled with copies of the floating map. The program determines how many copies can be displayed by executing an MSQFIT call, at **E**. The first parameter is the name of the map. In the second parameter, FMAPNUM, GDDM returns the number of instances that the floating area can accommodate. MSQFIT assumes default positioning for the map, as specified to GDDM-IMD. In other words, it assumes that you specify -1 as the second and third parameters of the MSDFLD calls. You can specify a fixed map as

the first parameter of MSQFIT. In this case, GDDM returns either 1 or, if the default position of the map is occupied, 0.

The number of orders for which the example displays data is always equal to FMAPNUM. A real program would include code to handle both fewer and more orders than this, of course.

Unique map identifiers: The fixed map is given an identifier of 1, at **D**. Each instance of the floating map is identified by a number from 2 through to FMAPNUM+1, at **F**.

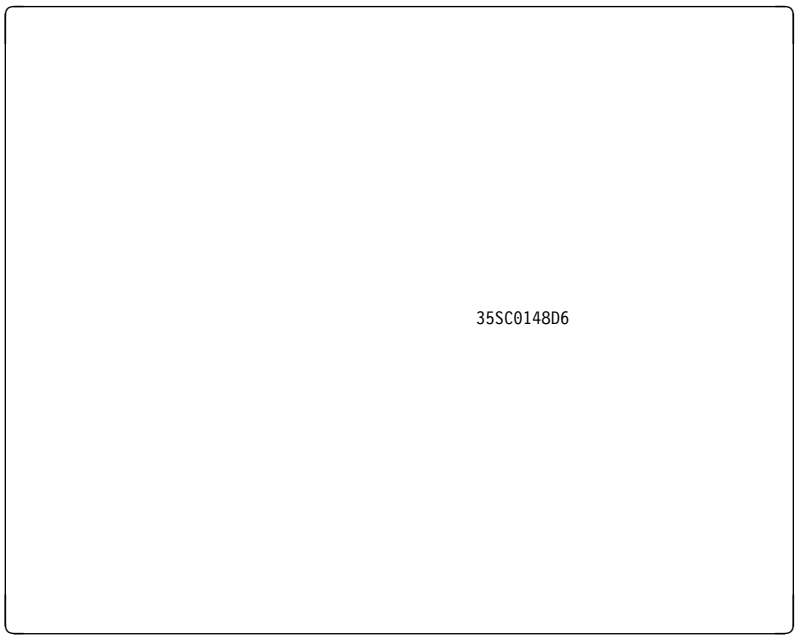


Figure 89. Typical display by MAPEX04

Querying changed maps

You can discover how many maps have been changed by testing the value that GDDM returns in the last parameter of the ASREAD call:

```
CALL ASREAD(ATTYPE,ATVAL,COUNT);
```

If the current page is a mapped one, COUNT is set to the number of maps changed by the operator. (If the page is not mapped, COUNT is set to the number of changed alphanumeric fields.)

If there is only one map on the page, the value of COUNT indicates whether or not it was modified. A 1 means that it was, and a 0 that it was not.

You can discover which maps on a multimap page were changed using the MSQMOD call:

```
CALL MSQMOD(10,IDS,LENGTHS);
```

The last two parameters are fullword arrays. They must both have the same size, which is specified in the first parameter.

IDS returns the identifiers of the maps that were changed by the terminal operator during the last ASREAD. Their order is the same as that in which the corresponding MSDFLD calls were executed. LENGTHS returns, in the same order, the lengths of the maps' application data structures.

If the number of changed maps is less than the number of elements, the unused elements in both arrays are set to 0. If the number of elements is less than the number of changed maps, you can use one or more further MSQMOD calls to obtain further identifiers and ADS lengths.

The last parameter of ASREAD and the second one of MSQMOD can be used together:

```
CALL ASREAD(ATTYPE,ATVAL,COUNT);

IF COUNT>0
  THEN DO;
    CALL MSQMOD(ARRAY_SIZE,IDS,LENGTHS);
    DO I=1 TO COUNT;
      SELECT(IDS(I));
        WHEN(1) DO; /* If map 1 was modified */
            .
            .
            .
          END;
        WHEN(2) DO; /* If map 2 was modified */
            .
            .
            .
          END;
        WHEN(3) DO; /* If map 3 was modified */
            .
            .
            .
          END;
      END;          /* End select group      */
    END;            /* End DO-loop           */
END;
```

Input from multiple copies of a map

To get input data from a display having more than one copy of a map, you can reuse the map's ADS any number of times. For instance, if the operator were allowed to alter the data displayed by the program in Figure 87 on page 300, the following code would reuse the ADS for FLOATER once per changed map:

```
CALL ASREAD(ATTYPE,ATVAL,COUNT);

DECLARE ID(1)              FIXED BINARY(31);
DECLARE LENGTH(1)          FIXED BINARY(31);

DO I=1 TO COUNT;
  CALL MSQMOD(1,ID,LENGTH);/*Get id & length of next changed map*/


  CALL MSGET(ID(1),0,LENGTH(1),FLOATER);/*Retrieve amended order*/

   /* . */
   /* . */                /* Process amended order data in ADS */
   /* . */

END;
```

Another way is to declare an array of ADSs, and read all the input data into the array before processing any of it:

```
CALL ASREAD(ATTYPE,ATVAL,COUNT);

DECLARE 1 FLOATER_INPUT(2:41)  /* Max. no. copies on screen */
%INCLUDE FLOATER;              /* Assumed to be 40.         */

DO MID=2 TO FMAPNUM+1;
  CALL MSGET(MID,0,FLOATER_ASLENGTH,FLOATER_INPUT(MID));
END;
```

The subscript of each ADS in the array FLOATER_INPUT is the same as the identifier of the floating map from which its data came.

Device-independence for mapped-alphanumeric applications

One of the advantages of mapping is that it allows your application programs a measure of device-independence. Terminals vary in the sizes of their display areas and in their features, but GDDM provides a way of ensuring that a program runs without change on several different types of terminal.

When you create a mapgroup, you must specify a device class to indicate to GDDM-IMD the type of terminal on which the mapgroup is to be used. Subsequently, you can specify additional device classes, and then generate different versions of the mapgroup for any or all of the specified classes.

Each generated mapgroup has a two-character suffix appended to the mapgroup name you specify to GDDM-IMD. The suffix is the same as the GDDM-IMD device class. A list of suffixes and their meanings is given in the *GDDM Interactive Map Definition* book. All the mapgroup names in the preceding examples have a suffix of D6, which means a display unit with 32 rows and 80 columns.

If your program is likely to run on several different types of terminal, you can leave the choice of suffix to GDDM. Instead of an explicit suffix on the mapgroup name in the MSPCRT call, you can code one or two dots, for example:

```
CALL MSPCRT(1,-1,-1,'MAPGRP..');
```

or

```
CALL MSPCRT(1,-1,-1,'MAPGRPD.');
```

GDDM replaces the dot or dots and creates the most suitable suffix for the current device. You must ensure that a generated mapgroup with the fully-suffixed name is available to GDDM.

In summary, you need to remember that GDDM uses the full name of the generated mapgroup, which is the name you assigned **plus the device suffix**. Your source code must either specify this name in full, or use the dot notation.

If you specify the name in full, the mapgroup need not match the device on which it is to be displayed. A mapgroup with an explicit suffix of D6, for instance, could be specified for a printer, or for a device with a display area that is not 32 rows by 80 columns.

If the mapgroup has been defined for a display area larger than the device possesses, some of the data may not be displayed. However, it is not removed from the GDDM page. If the page is too wide or too deep for the screen, it may still be displayable by hardware or software scrolling (as described in “Large and small pages” on page 473).

If you know that your program runs solely or mainly on a particular type of terminal, it is advisable to generate a mapgroup for it, and to include the corresponding suffix explicitly in the mapgroup name in the MSPCRT call. This is to save GDDM searching the library for a suitable mapgroup every time the MSPCRT call is executed.

Attribute handling when mapgroup does not match device

GDDM may produce unexpected results if the size of presentation area in a mapgroup is different from the display area of the device on which your program is executing, or if the map is being displayed in an emulated partition or an operator window.

One way to avoid problems is to ensure that the presentation area matches the device's display area. If this is not possible, the best solution is to terminate every field, on the same row on which it was started, with a protected or protected with autoskip field attribute.

Mismatches between the presentation area and the device's display area have the additional disadvantage that they cause extra processing by GDDM at execution time.

Output-only displays

You can use maps to format displays that do not require operator input. Such displays can be sent to screens or printers.

If the device is output-only, the program does not wait for input following an ASREAD or MSREAD call. For devices that do have input capabilities, you can use FSFRCE if you want your program to continue without waiting for the operator to cause an interrupt (by pressing the ENTER key, for instance).

Mapping queries

GDDM provides a number of calls for enquiring about maps and associated matters. One of them is described in “Querying changed maps” on page 302. In addition to changed maps, you can query, for instance, a mapgroup's or map's characteristics, or the position of a map on a page and its size. The calls all start with MSQ, and are described in the *GDDM Base Application Programming Reference* book.

Chapter 16. Variations on a map

Chapter 15, “Mapped alphanumerics” describes how to use maps to supply the basic framework of a dialog with the end user. This section introduces further GDDM and GDDM-IMD facilities that help you with the details. Mainly, it describes how your program can vary the format defined by the map. There is also a section that tells you how to add graphics to maps.

The procedures for varying the format maps may seem complicated if you are new to the techniques of mapping. However, they are designed to simplify the programming of complex dialogs, by allowing GDDM to do more of the work. These procedures are not essential, but are intended to help you. If you prefer, you can get similar results in most cases with the facilities described in Chapter 15, “Mapped alphanumerics” on page 283.

This section does not apply to graphics-only devices such as plotters.

Selecting fields from a map for use in complex dialogs

A map may contain fields that you intend to use in some circumstances and not in others. For example, a data-entry map might include column headings, some of which are not always required. GDDM lets your program decide at execution time which fields are to display data.

Your program can select particular fields if you follow this procedure. When you define the map, you specify default data for the fields in question. During execution, your program chooses, for each I/O operation and each field, either to use the default data, or to use data from the ADS, or to leave the data already present in the field as it is. In a column-heading field, for instance, the default data could be the heading text. Before sending the page to the terminal, your program might either put the default data into the field, or put blanks into it from the ADS, or leave it unchanged from previous operations.

A field that is to be treated in this way must have an extra element associated with it in the ADS, called a **selector adjunct**. When you create a map, you must tell GDDM-IMD which fields are to have selector adjuncts. You do so on the Field Naming or Application Data Structure Review frame of the GDDM-IMD map editor.

In your program, you put a code into the selector adjunct. The code is interpreted when you execute an MSPUT call. It tells GDDM whether MSPUT is to update the field, and if so, whether default data or data from the ADS is to be used.

The ADS in Figure 90 on page 308 has a selector adjunct at **A**. GDDM-IMD gives selector adjuncts the same names as the associated fields, with a suffix of “_SEL” in PL/I, “-SEL” in COBOL, and “S” in Assembler. Selector adjuncts are one byte long.

In addition to selector adjuncts, there are several other types of adjunct. They are a general control mechanism used for several different purposes. You can use them to:

- Set field attributes
- Position the cursor

© Copyright IBM Corp. 1982, 1996

- Extend highlighting
- Set the color of fields
- Select programmed symbols

Most types of adjunct are introduced in this section; a full list is given in the *GDDM Base Application Programming Reference* book.

Programming example using a selector adjunct to display a message

The program in Figure 90 uses a selector adjunct to control an error message field. The output of the program is the same as for MAPEX01, as shown in Figure 83 on page 288.

Although their output is similar, the maps used by the two programs differ. In addition to having a selector adjunct, the map used by MAPEX05 has the message text as default data in the message field, whereas the one used by MAPEX01 has blanks.

The selector adjunct is declared at **A**. The complete ADS is cleared at **B**. The selector adjunct for the message field is set to 1 at **C**. This value means that the write-type MSPUT call, **D**, updates the field with data from the ADS. Initially, then, all the fields, including the message field, are blank.

If the end user of the program makes an error, the message field selector adjunct is reset to 2 at **E**. This value means that the MSPUT, **D**, updates the message field with default data. The default data is the error message, as defined to GDDM-IMD when the map was created.

```
MAPEX05:  PROC OPTIONS (MAIN);

DCL 1 CUSTINV,                  /* Included ADS              */
      10 MESSAGE_SEL                    CHAR(1),                A
      10 MESSAGE                        CHAR(78),
      10 CUSTNO                         CHAR(5),
      10 INVNO                          CHAR(4),
      ORDER1_ASLENGTH                   FIXED BIN(31,0) STATIC
                                          INIT(88);
DCL (ATTYPE,ATVAL,COUNT) FIXED BIN(31);    /* ASREAD arguments*/
DCL WRITE FIXED BIN(31) INIT(0);     /* MSPUT write operation */
DCL VALID BIT(1) INIT('1'B);    /* on until invalid data found*/
CALL FSINIT;                    /* Initialize GDDM.          */
CUSTINV = '';                   /* Clear the ADS, so that map-*/ B
                                /* defined values are taken.  */
MESSAGE_SEL = '1';              /* Set message selector to    */ C
```

Figure 90 (Part 1 of 2). Listing of MAPEX05 source code

```
                                /* issue blank message.       */
CALL MSPCRT(1,                  /* Create page.               */
            -1,                 /* Use mapgroup-defined page  */
            -1,                 /* depth and width            */
            'ACME00D6');        /* for mapgroup 'ACME00D6'.   */
CALL MSDFLD(1,                  /* Map an area of page,       */
            -1,                 /* using the map-defined row  */
            -1,                 /* and column positions       */
            'ORDER1');          /* and map ORDER1.            */
LOOP:
CALL MSPUT(1,                   /* Put data into map on page, */  D
           WRITE,               /* with write operation,      */
           ORDER1_ASLENGTH,     /* specifying the ADS length, */
           CUSTINV);            /* and the data length.       */
CALL ASREAD(ATTYPE,             /* Output the current page, & */
            ATVAL,              /* wait for end-user input.   */
            COUNT);
IF ATTYPE=1 & (ATVAL=3 | ATVAL=15) /*End user pressed end key?*/
  THEN GO TO FIN;
IF COUNT > 0 THEN DO;           /* Data entered, so check it. */
  CALL MSGET(1,0,               /* Get data from map          */
             ORDER1_ASLENGTH,   /* into the ADS.              */
             CUSTINV);
  IF VERIFY(CUSTNO,'0123456789') = 0 /* Are CUSTNO and        */
   & VERIFY(INVNO,'0123456789')  = 0 /* INVNO numeric?        */
    THEN DO;
      /*  .  */                 /* Process CUSTNO and INVNO   */
      /*  .  */
      /*  .  */
     MESSAGE = ' ';             /* Clear any existing message */
      END;
      ELSE VALID = '0'B;        /* Indicate error.            */
    END;
    ELSE VALID = '0'B;          /* No data entered, so        */
                                /* indicate error.            */
IF ¬VALID THEN DO;              /* Error found, so redisplay  */
  VALID = '1'B;                 /* the map.                   */
  MESSAGE_SEL = '2';            /* Set selector, so that map  */  E
END;                            /* message appears            */
GO TO LOOP;
FIN:
CALL FSTERM;                    /* Terminate GDDM             */
%INCLUDE ADMUPINA;              /* GDDM entry declarations    */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINM;
END MAPEX05;
```

Figure 90 (Part 2 of 2). Listing of MAPEX05 source code

Write, rewrite, and reject

The MSPUT call updates the alphanumeric fields contained in a mapped field. In the simple case of an ADS without selector adjuncts, this means it moves the data from the ADS into the variable data fields.

When the ADS contains selector adjuncts, what happens depends on two things: the codes in the adjuncts, and the type of MSPUT operation.

There are three types of MSPUT operation. They are called write, rewrite, and reject. The operation is specified by the second parameter of the MSPUT. A 0 means write, 1 rewrite, and 2 reject.

In a write operation, MSPUT:

1. Sets all variable data fields to their initial values. If you specified no initial data for a field when you created the map, it is set to blanks: blanks are the default initial data.
2. Inspects the selector adjunct of each field, and makes changes according to its value:

   blank — No further change to the field.

   1 — Update the field with variable data from the ADS.

   2 — Update the field with default data from the map (or with blanks if you specified no default data). For a write operation, a 2 has the same effect as a " ", as the field already has default data in it.

   3 — Means the same as a 1 character.

In a rewrite operation, MSPUT does the same as in a write, except that it omits the first step. The fields are not set to their defaults before the selector adjuncts are processed. In a rewrite, a 2 character is not the same as " ", because the field does not necessarily contain default data.

In a reject operation, MSPUT does the same as in a rewrite. The difference between rewrite and reject becomes apparent only on input. It is explained in "Effect of reject operation" on page 311.

The differing applications of a write operation, compared with a rewrite (or reject), can be summarized as follows. You should use a write when you create a new display from scratch. You should use a rewrite (or reject) when you update some of the fields of an existing display; you indicate which fields are to be updated by setting their selector adjunct to a 1 or 2 character.

Selector adjuncts on input

Selector adjuncts are used on input, as well as output. When you execute an MSGET call, GDDM puts a code into the adjunct to indicate whether the field has been modified.

You may well find that input codes are the most useful aspect of adjuncts. They provide a simple means of discovering which fields have been changed by the end user. Without them, your program might have to store the old values of all the

updatable fields, and compare them with the new values in the ADS after the MSGET.

The code indicates the state of the field as it exists on the current page, as follows:

blank
: The field has no value. Either it has not had any data in it since the start of execution, or your program has emptied it and no data has been put into it since. You empty a field by clearing it to blanks or nulls, setting its selector adjunct to " ", and executing a write-type MSPUT call.

1
: The field has a new value set by the end user. Except when the preceding MSPUT was a reject type, it indicates that the field was updated during the last ASREAD (or MSREAD). The precise meaning in the reject case is explained in "Effect of reject operation."

2
: Not used on input.

3
: The field has an old value. In other words, it contains a value that was put into it either by the application program, or by the end user during an ASREAD (or MSREAD) other than the last one.

Effect of reject operation

In some circumstances, it is necessary to repeatedly send a map back to the terminal. For instance, the end user may need several attempts to supply completely valid data.

You can send a map back to the terminal by a reject-type MSPUT operation followed by an ASREAD. Then, for each field changed by the end user of the program, MSGET returns a code of 1, the same as after a write or rewrite. A reject results in a different setting only if you resend such fields to the terminal, and the end user leaves them unchanged. On the next input, MSGET would return a 1 character instead of a 3. A 1 still indicates new data supplied by the end user, but it was not necessarily supplied during the most recent ASREAD.

In hardware terms, a reject does not reset the modified data tags (MDTs) of the previously modified fields, whereas write and rewrite do. The possibly different value of the selector adjunct on input is the only way in which this difference is apparent to your program.

The reject facility enables you to accumulate the changes made by the operator over a number of ASREADs, without having to store the data in your program.

Uses of selector adjuncts

The following outline of a program illustrates the most important uses of selector adjuncts on both output and input.

Initially the program creates the following display:

```
                                  ←—— Space for error message
 DEPT   DOLLARS     WEEK          ←—— Constant data headings
 ****                             ←—— Operator input line
```

The operator should enter a four-character department code, an expenditure figure of up to ten digits, and a week number of two digits. The three input data fields have constant data headings of DEPT, DOLLARS, and WEEK. The department code field has map-defined default data of four asterisks; the expenditure and week number fields have no default data. A field at the top of the display is used for error messages; it has no default data. The department code, expenditure, week number, and message fields have selector adjuncts.

This is the ADS:

```
1 DEPEXP
  10 MSG_SEL      CHARACTER(1),
  10 MSG          CHARACTER(30),
  10 DCODE_SEL    CHARACTER(1),
  10 DCODE        CHARACTER(4),
  10 EXP_SEL      CHARACTER(1),
  10 EXP          CHARACTER(10),
  10 WEEK_SEL     CHARACTER(1),
  10 WEEK         CHARACTER(2),
```

The program creates the display by setting the ADS to all-blanks. The four blank selectors cause the ASREAD to send the default data to the screen, In the department-code field the default data consists of asterisks, and in the other three it is blanks (because these fields have no default data specified in the map).

Suppose the end user updates the display, as follows, and presses the ENTER key:

```
 DEPT   DOLLARS     WEEK
 *876   HABY
```

The program executes an MSGET, which puts the following values into the ADS:

```
MSG_SEL MSG     DCODE_SEL DCODE     EXP_SEL EXP       WEEK_SEL WEEK

Blank   Blank   1         *876      1       HABY      Blank    Blank
```

This MSGET is contained in a loop that checks, first, that none of the three input data fields has a blank selector code, and then, that the department code field is alphabetic and the other two input fields are numeric. If either check fails, it puts the text of an error message into the message field, sets the error message

selector to a 1 character, and executes a reject-type MSPUT followed by an ASREAD. Because the error message selector field is set to 1, the ASREAD sends the message text to the terminal.

In this case, the ADS has the following values, before the MSPUT:

```
MSG_SEL MSG     DCODE_SEL DCODE     EXP_SEL EXP       WEEK_SEL WEEK

1       Message 1         *876      1       HABY      Blank    Blank
        text
```

The program executes an ASREAD after the MSPUT, putting this display on the screen:

```
WEEK NUMBER MISSING

DEPT   DOLLARS     WEEK

*876   HABY
```

The end user then updates the display as follows:

```
WEEK NUMBER MISSING

DEPT   DOLLARS     WEEK

*876   HABY        22
```

In the program, control returns to the MSGET at the top of the loop, which updates the ADS as follows:

```
MSG_SEL MSG     DCODE_SEL DCODE     EXP_SEL EXP       WEEK_SEL WEEK

3       Message 1         *876      1       HABY      1        22
        text
```

The program finds that all the selectors in the input fields are set, but that the department code and expenditure are invalid. It therefore sets the message selector to a 1 character again, and puts the text of another message into the message field. After the reject and ASREAD, the screen looks like this:

```
ERROR(S) IN DEPT, DOLLARS

DEPT   DOLLARS     WEEK

*876   HABY        22
```

The end user corrects the input as follows:

```
ERROR(S) IN DEPT, DOLLARS

DEPT   DOLLARS     WEEK

HABY   876         22
```

After this MSGET, the ADS contains the following data :

```
MSG_SEL MSG     DCODE_SEL DCODE     EXP_SEL EXP       WEEK_SEL WEEK

3       Message 1         HABY      1       876       1        22
        text
```

Because all the fields are now valid, control drops out of the loop.

After the program has processed the input data, it redisplays the page for the end user to provide the next input. All the program has to do is change the message-field selector to a 2 character, to remove the message from the screen, and execute a rewrite-type MSPUT and an ASREAD. It therefore changes the ADS to:

```
MSG_SEL MSG     DCODE_SEL DCODE     EXP_SEL EXP       WEEK_SEL WEEK

2       Message 1         HABY      1       876       1        22
        text
```

After the ASREAD, the screen looks like this:

```


DEPT   DOLLARS     WEEK

HABY   876         22
```

The end user amends the screen to:

```


DEPT   DOLLARS     WEEK

HABY   1234        23
```

After an MSGET, the values in the ADS are as follows:

```
MSG_SEL MSG     DCODE_SEL DCODE     EXP_SEL EXP       WEEK_SEL WEEK

Blank   Blank     3       HABY          1    1234         1      23
```

Because the department-code field has a selector character of 3, meaning that it contains old data, there is no need for the program to check it.

Alarm and keyboard locking

GDDM-IMD enables you to specify for each map whether the alarm is to sound when it is sent to the terminal, and whether the keyboard is to lock or allow free input. When the keyboard locks, the terminal operator has to press RESET before the terminal accepts any more input.

Effects of maps

When you create a map, you can specify to GDDM-IMD different options for each type of MSPUT operation. If your program updates any map on the current page using an operation for which the alarm has been specified, the alarm sounds; and similarly if keyboard locking has been specified, the keyboard locks when the map is updated by the specified operation. Otherwise, no action is taken.

The GDDM-IMD defaults are that GDDM should sound the alarm and lock the keyboard only after a reject operation. If these defaults apply to all maps on the page, the keyboard is locked and the alarm sounded if any of the maps is updated by a reject-type MSPUT.

Other considerations

The FSALRM call (see “Example: Program using procedural alphanumerics to display a bank balance” on page 77) sounds the alarm when the page current at the time of its execution is sent to the terminal. It happens irrespective of the map specification and type of MSPUT operation.

You can use the DSOPEN call (see Chapter 18, “Device support in application programs” on page 371) to tell GDDM to unlock the keyboard after every output operation. This overrides the effect of maps. If there is no overriding DSOPEN specification, the keyboard is always locked after an FSFRCE, and may be locked or unlocked after an ASREAD, MSREAD, or GSREAD, as described in “Effects of maps.”

Protecting fields from the end user

The 3270 display unit enables you to protect fields from change by the operator. It does so by either locking the keyboard if the operator tries to type into it, or by making the cursor skip over it if the operator tries to move the cursor into it. The first type of field is called **protected**, and the second type **autoskip**. Fields that the operator is allowed to type into are called **unprotected**.

It might seem that variable data fields should always be unprotected. This is not the case, however, because “variable” means capable of being changed by the operator or by the program. If your application uses fields that may be changed by the program but need to be protected from change by the end user, you would make them variable but give them the protected or autoskip attribute.

You can specify which of the three protection attributes a field is to have on the Field Attribute Definition frame of GDDM-IMD’s map editor. At execution time, you can override the map-defined attribute by using a **base attribute adjunct**. As with selector adjuncts, you specify which fields are to have base attribute adjuncts on the Field Naming or Application Data Structure Review frame of the GDDM-IMD

map editor. Like selector adjuncts, they appear to the application as extra elements in the ADS.

The base attribute adjunct

The base attribute adjunct consists of two elements, each one byte long. There is an example in “Defining the base attributes that are to apply to mapped fields.” The second byte indicates what the program-defined field attributes are to be. The first contains a code that indicates whether GDDM should use these attributes, use the ones in the map, or leave the field's attributes unchanged.

You may notice that this discussion refers to attributes in the plural. This is because the second byte can be used to define several types of attribute, not just the one concerned with protecting the field from end-user input. The other types are listed in “The base attribute adjunct.”

The effect of an MSPUT call on the base attributes is analogous to its effect on the data. It depends on the type of operation and the code in the first byte of the base attribute adjunct. A write operation first resets all base attributes in the mapped field to the values specified in the map, or, where none were specified, to GDDM-IMD defined defaults. The GDDM-IMD defaults are listed in “The base attribute adjunct.”

The write-type MSPUT call sets each field's base attributes according to the contents of first byte of the adjunct, as follows:

A blank character	Leave the base attributes unchanged.
A 1 character	Set the attributes to those specified in the second byte of the adjunct.
A 2 character	Apply the map-defined (or GDDM-IMD default) attributes.
A 3 character	The same as a 1 character.

For rewrite and reject operations, MSPUT sets the base attributes according to the code in the first byte of the adjunct, without first resetting them to the map-defined (or GDDM-IMD default) values.

Defining the base attributes that are to apply to mapped fields

The second byte of the base attribute adjunct is used to define all the attributes that the 3270 hardware stores in the attribute byte. They are called the **base attributes**, and consist of:

- Protection attribute, which, as already explained, can be set to one of these values:
  - Protected
  - Unprotected
  - Autoskip.

  If you do not specify a value when you define a field, GDDM-IMD generates a default of autoskip for a constant field, or unprotected for a variable field.
- Intensity attribute, which can be set to one of these values:
  - Normal.
  - Intensified. This value also makes the field light-pen detectable.

  – Non-display.

  The GDDM-IMD generated default is normal.

- Light-pen attribute, which can be set to detectable or nondetectable. The GDDM-IMD generated default is nondetectable.
- MDT bit, which can be either on or off. The setting of the MDT bit in the base attribute adjunct overrides the settings made by GDDM in response to the write, rewrite, and reject operations of MSPUT. The GDDM-IMD generated default is MDT off.
- Data-type attribute, which can be set to alphanumeric or numeric. The GDDM-IMD generated default is alphanumeric.

The second byte of the base-attribute adjunct must be set to the bit pattern representing the required 3270 attribute byte. The bit patterns are described in the *GDDM Base Application Programming Reference* book.

GDDM supplies sets of special variables for inclusion in your programs to help with setting base-attribute and other adjuncts. The names of these sets of variables are:

ADMUAIMC For mapped alphanumerics in assembler language programs

ADMUBIMC For mapped alphanumerics in C/370 programs

ADMUCIMC For mapped alphanumerics in COBOL programs

ADMUPIMC For mapped alphanumerics in PL/I programs

They are stored in the library called ADMLIB on VM/CMS, or in the sample library (GDDMSAM) on TSO or CICS. (ADMLIB and GDDMSAM also hold the PL/I declarations of the GDDM entry-points.) You need to make the library available to your program, as outlined in “How to compile and run a PL/I GDDM program” on page 14. They contain mnemonically named variables for every base attribute, and for combinations of attributes. The variables are initialized to the bit patterns required in the 3270 attribute byte.

Here is an ADS containing a base attribute adjunct, and statements to protect and brighten the field called CUSTNUM.

```
DCL 1 CUSTN,

        10 MESSAGE_FIELD                 CHAR(78),
        10 CUSTNUM_ATTR_SEL              CHAR(1),
        10 CUSTNUM_ATTR                  CHAR(1),
        10 CUSTNUM                       CHAR(5),
        CUSTNMAP_ASLENGTH                FIXED BIN(31,0) STATIC
                                         INIT(85);

CUSTNUM_ATTR_SEL = '1';           /* Tell GDDM to use adjunct-  */
                                  /* defined base attributes.   */
CUSTNUM_ATTR = PROTECT_BRIGHT;    /* Use variable from ADMUPIMC */
                                  /* to define base attributes. */
CALL MSPUT(1,0,CUSTNMAP_ASLENGTH,CUSTN);

%INCLUDE ADMUPIMC;                /* INCLUDE GDDM-supplied base */
                                  /* attribute variables        */
```

The position of the cursor

You can control the position of the cursor on output, and find out where the end user placed it on input. In both cases you can use the **cursor adjunct**. The cursor adjunct is a one-byte field. As for all adjuncts, you need to tell GDDM-IMD which fields are to have them, using the Field Naming or Application Data Structure Review frame of GDDM-IMD's map editor.

Positioning the cursor when your program sends output to the display

You can often improve the usability of your displays by putting the cursor where the end user is most likely to start entering data. There are three ways in which GDDM determines the cursor position on output:

1. Your program can specify the position dynamically, using cursor adjuncts and cursor-positioning calls.
2. If the program does not specify a dynamic position, then GDDM uses a **static** position specified during map definition.
3. If your program does not specify a dynamic position, and no map on the page has a specified static position, GDDM uses a default position.

Positioning the cursor dynamically

Your program can position the cursor dynamically using cursor adjuncts, possibly in conjunction with the MSCPOS call. If the program specifies more than one dynamic position, GDDM ignores all except the latest.

You use the cursor adjuncts by setting one of them to a 1 character and the others to blank before executing an MSPUT. This causes the cursor to be placed under the first character of the field with the 1 character in its cursor adjunct.

To position the cursor under a character other than the first one in a field, you can execute an MSCPOS call before the MSPUT. An example is:

```
CALL MSCPOS(10);
```

which would put it under the tenth character.

The MSCPOS call is put into effect at the next MSPUT. After that it has no effect, and you must call MSCPOS again if you want to control the position at any later MSPUT.

MSCPOS can only affect a field with a cursor adjunct character of 1. It has no effect if you have not defined a cursor adjunct for the field using GDDM-IMD or if the cursor adjunct for that field has been set to blank.

Static positioning of the cursor

You specify a static position using the ATTRIBUTE CURSOR command on the Field Attribute Definition frame of GDDM-IMD's map editor. The static specification is put in effect by an MSDFLD call or a write-type MSPUT call, as follows:

- For a new page, a static cursor position is established by the first MSDFLD that refers to a map that has a static specification.
- After an ASREAD (or other I/O operation), a static position is reestablished by the first MSDFLD or write-type MSPUT that refers to such a map.

Rewrite- and reject-type MSPUT calls have no effect on the static cursor position.

Default positioning of the cursor

When a page is first sent to the terminal, GDDM's default action is to put the cursor in the top left-hand corner of the screen. Subsequently, the default action is to leave the position of the cursor unchanged.

Simple example using cursor adjuncts on output

For this example, CUSTNUM and INVOICE are both input fields. Cursor adjuncts have been defined for both. If an error is found in one of them, its cursor adjunct is set to 1 and that of the other is set to blank. A reject-type MSPUT call is then executed.

```
DCL 1 CUSTNO,

        10 MESSAGE_FIELD                 CHAR(78),
        10 CUSTNUM_CURSOR                CHAR(1),
        10 CUSTNUM                       CHAR(5),
        10 INVOICE_CURSOR                CHAR(1),
        10 INVOICE                       CHAR(4),
        CUSTNO_ASLENGTH                  FIXED BIN(31,0) STATIC
                                         INIT(89);

  /* . */
  /* . */

IF CUST_INVALID
  THEN DO;
  MESSAGE_FIELD  = 'ERROR IN CUSTOMER NUMBER FIELD';
  CUSTNUM_CURSOR = '1';
  INVOICE_CURSOR = ' ';
END;

IF INV_INVALID
  THEN DO;
  MESSAGE_FIELD  = 'ERROR IN INVOICE NUMBER FIELD';
  CUSTNUM_CURSOR = ' ';
  INVOICE_CURSOR = '1';
END;

CALL MSPUT(1,2,CUSTNO_ASLENGTH,CUSTNO);
```

A typical cursor-positioning sequence

A typical application might use two maps, one to solicit a request from the end user, and a second one, displayed beneath or beside the first, to provide the response. It is assumed that both maps have had static cursor positions defined with ATTRIBUTE CURSOR commands, and have cursor adjuncts on their variable data fields.

The first map might be displayed using an MSDFLD call followed by an ASREAD, without any variable data being added – in other words, without an MSPUT call being executed:

```
CALL MSPCRT(1,-1,-1,'MAPGRP1');      /* Create new mapped page. */
CALL MSDFLD(1,-1,-1,'MAP1');         /* Put first map onto page.*/
CALL ASREAD(1,TYPE,VALUE);
```

The cursor would be displayed in the static position defined by MAP1.

The ASREAD would be followed by an MSDFLD call to add the second map to the page. One or two write-type MSPUT calls might then be executed to add variable data to one or both maps:

```
CALL MSDFLD(2,-1,-1,'MAP2');          /* Put MAP2 onto the page.*/
CALL MSPUT(1,1,MAP1_ASLENGTH,MAP1_ADS); /* Write-type operation */
                                        /* for MAP1             */
CALL MSPUT(2,1,MAP2_ASLENGTH,MAP2_ADS); /* Write-type operation */
                                        /* for MAP2             */
CALL ASREAD(1,TYPE,VALUE);
```

Assuming that no cursor adjuncts had been set to 1 characters, the MSDFLD would cause the cursor to be displayed in the static position defined by the second map. A cursor adjunct character of 1 in the ADS for MAP1 would override the static positioning, and one in the ADS for MAP2 would override one in the ADS for MAP1.

If the MSPUT for the first map preceded the MSDFLD for the second, like this:

```
CALL MSPUT(1,1,MAP1_ASLENGTH,MAP1_ADS); /* Write-type operation */
                                        /* for MAP1             */
CALL MSDFLD(2,-1,-1,'MAP2');            /* Put MAP2 onto page   */
CALL MSPUT(2,1,MAP2_ASLENGTH,MAP2_ADS); /* Write-type operation */
                                        /* for MAP2             */
CALL ASREAD(1,TYPE,VALUE);
```

then the cursor would be replaced in the static position defined by the first map, assuming no cursor adjuncts had been set in either ADS.

Determining the cursor position following input by the end user

You discover in which field the end user left the cursor by inspecting the cursor adjuncts after an MSGET. This call sets the cursor adjunct of the field that contains the cursor to 1, and he cursor adjuncts of all the other fields to “ ”. With this facility, you can create menus from which the program end user makes a selection using the cursor.

You can discover the position of the cursor within a field by executing an MSQPOS call, for example:

```
CALL MSQPOS(POSN);
```

This call returns the position of the cursor within the field that had its adjunct set to 1 by the last MSGET. To determine the exact cursor position, your program would

execute an MSGET, inspect the adjuncts, and if one of them is set to 1, execute an MSQPOS.

If the cursor was outside the map, or within a field that does not have a cursor adjunct, MSQPOS returns the value –1.

In some applications, the end user positions the cursor under a field without typing data into it, for example to select from a menu. In such cases, the map must be designated a **cursor receiver**. You make the designation on the Map Characteristics frame of the map editor.

Padding mapped fields with null characters

GDDM-IMD pads default data with blanks to fill the field. If you specify no default data, GDDM-IMD fills the complete field with blanks. If you want to pad with nulls, perhaps to allow the end user to use the insert key, you must provide the field with a **length adjunct**.

Here is an ADS containing two fields, the first of which has a length adjunct:

```
DCL 1 CUSTOMER,

        10 CUSTNUM_LENGTH                FIXED BIN(15),
        10 CUSTNUM                       CHAR(5),
        10 INVOICE                       CHAR(4),
        CUSTOMER_ASLENGTH                FIXED BIN(31,0) STATIC
                                         INIT(11);
```

On output, your program sets the length adjunct to the length of data in the field, and GDDM pads the remainder of the field with nulls. If the end user modifies the field, then on input, GDDM sets the length adjunct to the new length of the data.

Light pen and CURSR SEL key

If the terminal has a light pen, you can arrange for the end user to use it to select fields in a mapped display. Some terminals have a CURSR SEL key. This provides an equivalent function to the light pen. Instead of positioning the pen over a field and pressing it, the end user moves the cursor to the field and presses CURSR SEL.

To enable the end user to use the light pen (or CURSR SEL key) on a field, you must first give it the detectability attribute. This is a base attribute, and can be given to the field using the Field Attribute Definition frame of the GDDM-IMD map editor. Another way is for you to give the field a base attribute adjunct which your program can make detectable at execution time, as outlined in “The base attribute adjunct” on page 316.

You must specify that GDDM-IMD is to create selector adjuncts for detectable fields, because GDDM uses this adjunct to indicate which field has been selected. You specify that adjuncts are required using the Field Naming or Application Data Structure Review frame of the map editor.

In addition to making the fields detectable, you must put a **designator character** in the first position of each field. These characters indicate the precise action that the terminal must take when a field is selected. These actions are described in *GDDM Base Application Programming Reference* book. Here is a summary:

? Delayed detection. Nothing is transmitted to your program until the end user takes some other action that causes an interrupt, such as pressing the ENTER key, or selecting an immediate detection field. On selection, the ? changes to a >. The operator can cancel this action by reselecting the field; the > then changes back to a ?.

" " Immediate detection without data. Selection causes an immediate transmission to your program, but without any data.

& Immediate detection with data. When the field is selected, the data in all the fields in the display is transmitted, as if the operator had pressed the ENTER key.

The designator characters appear in the first character positions of the fields. You can put them into the fields as default data from the map, or variable data from the ADS.

The operator may overtype the designator character if the field is unprotected. You could set the protection attribute on for all detectable fields. However, this would mean that the cursor could not be moved into the field using a tabbing key, which would inhibit the use of the CURSR SEL key. The solution to this problem is to make the field unprotected but ensure that the program writes the designator character into it at each ASREAD (or MSREAD).

On input, the selector adjunct codes have the same meanings after light-pen detection as when the operator types in data. The MSGET call sets the selector adjuncts of any newly selected fields to 1 characters. For a field selected earlier, the code is a 3 character; and for a field that has not been selected or had data put into it, the code is " ". The 1 character is retained over a series of reject operations, as described in "Effect of reject operation" on page 311.

If an immediate light-pen field contains the cursor when it is selected, its cursor adjunct, if it has one, is set to a 1 character.

Example of selection with cursor, light pen, and PF key

The program in Figure 91 on page 324 creates a display from which the terminal operator must select one of four options. The format of the map it uses is shown in Figure 92 on page 326. All the text is constant data.

There are three methods of selection:

- With one of the four specified PF keys
- Positioning the cursor under the selected option and pressing the ENTER key
- With the light pen (or CURSR SEL key). The first character of each selectable field is a blank, which means immediate selection with no data.

The map was designated a cursor receiver on the GDDM-IMD Map Characteristics frame. GDDM-IMD enables you to group similar fields into arrays, using the Field Naming frame of the map editor. This feature has been used for the four option

fields in this example. Selector and cursor adjuncts were specified on the Application Data Structure Review frame, and these are shown in the ADS at **A** and **B**. An initial position was specified for the cursor, namely, under the one-byte field called DUMMY.

The main loop of the program is executed once each time the operator makes a selection. The first statement of the loop, **C**, clears the ADS. This removes any message outstanding from a previous iteration, and sets the selector and cursor adjuncts to blank. This means that GDDM uses the map-defined cursor position and the default data for all the option fields.

If the last input was incorrect, the error message is then copied into the ADS.

The MSPUT, **D**, updates the page with all the changes resulting from **C**, and with the error message, if this is required. It specifies a write-type operation, so the blank designator characters specified in the map are put into the selectable fields before every execution of the ASREAD. This prevents any problems arising from the operator overtyping these characters.

The SELECT statement, **E**, discovers which selection method the end user used, by testing the first ASREAD parameter, ATTYPE. If the value 0 is found at **F**, meaning that the ENTER key was pressed, the group of statements starting at **G** is executed. These find which of the cursor adjuncts contains a character 1. The program calls a subroutine to perform the requested function, or sets a flag if terminate was requested. If no option was selected, the error flag is set.

If the value 1 for ATTYPE is detected at **H**, the second ASREAD parameter, called ATVAL, is tested by the group of statements at **I**, to discover which PF key was pressed.

If the value 2 for ATTYPE is found at **I**, meaning that the light pen (or CURSR SEL key) was used, the selected field is discovered in the statements at **J**.

If ATTYPE has some value other than 0, 1, or 2, the end user must have pressed an invalid key, so the error flag is set at **K**.

```
MAPEX08:  PROC OPTIONS (MAIN);

DCL 1 INITSEL,                    /* Application Data Structure */

        10 MESSAGE_FIELD                CHAR(78),
        10 DUMMY                        CHAR(1),
        10 OPTION_ARRAY(4),
         15 OPTION_SEL                  CHAR(1),                A
         15 OPTION_CURSOR               CHAR(1),                B
         15 OPTION                      CHAR(30),
        INITSEL_ASLENGTH                FIXED BIN(31,0) STATIC
                                        INIT(207);

DCL (ATTYPE,ATVAL,COUNT) FIXED BIN(31);/* ASREAD arguments.     */
DCL WRITE   FIXED BIN(31) INIT(0);     /* MSPUT write operation.*/
DCL PROCESS BIT(1) INIT('1'B);    /* On until terminate chosen. */
DCL INVALID BIT(1) INIT('0'B);    /* On if invalid option chosen*/

CALL FSINIT;                      /* Initialize GDDM.           */

CALL MSPCRT(1,                    /* Create mapped page 1,      */
            -1,                   /* with GDDM-IMD specified    */
            -1,                   /* page width and depth,      */
            'ACME00D6');          /* for mapgroup ACME00D6.     */

CALL MSDFLD(1,                    /* Format an area of the page,*/
            -1,                   /* at GDDM-IMD specified row  */
            -1,                   /* and column position,       */
            'INITSEL');           /* using map INITSEL.         */

DO WHILE (PROCESS);               /* Until end option chosen.   */

  INITSEL = '';                   /* Clear the message field    */ C
                                  /* adjuncts.                  */
  IF INVALID THEN DO;             /* Error noted.               */
    INVALID = '0'B;
    MESSAGE_FIELD =
                  'INVALID SELECTION';
    CALL FSALRM;                  /* Sound the alarm.           */
  END;
  CALL MSPUT(1,                   /* Add ADS data to map on page*/ D
             WRITE,               /* with write operation,      */
             INITSEL_ASLENGTH,    /* specifying ADS length      */
             INITSEL);            /* and data area.             */
```

Figure 91 (Part 1 of 3). Listing of MAPEX08 source code

```
CALL ASREAD(ATTYPE,              /* Send mapped page to        */
            ATVAL,               /* terminal and wait for      */
            COUNT);              /* end-user input.            */

CALL MSGET(1,0,                  /* Get response into ADS.     */
           INITSEL_ASLENGTH,
           INITSEL);

SELECT (ATTYPE);                 /* Analyze interrupt type -   */  E
  WHEN (0) DO;                        /* ENTER key, so inspect */  F
         IF OPTION_CURSOR(1) = '1'    /* the cursor adjuncts   */  G
           THEN CALL CPROC;           /* to see which field    */
    ELSE IF OPTION_CURSOR(2) = '1'    /* (if any) the cursor   */
           THEN CALL DPROC;           /* was in.               */
    ELSE IF OPTION_CURSOR(3) = '1'
           THEN CALL PPROC;
    ELSE IF OPTION_CURSOR(4) = '1'
           THEN PROCESS = '0'B;
    ELSE INVALID = '1'B;              /* Not in a valid field. */
  END;                                /* End cursor inspection.*/

  WHEN (1)                            /* PF key interrupt, so  */  H
    SELECT (ATVAL);                   /* analyze the value     */  I
      WHEN (10) CALL CPROC;           /* returned in ATVAL.    */
      WHEN (11) CALL DPROC;
      WHEN (12) CALL PPROC;
      WHEN (3)  PROCESS = '0'B;
      OTHERWISE INVALID = '1'B;       /* Invalid PF key chosen.*/
    END;                              /* End PF key inspection.*/

  WHEN (2) DO;                        /* Light pen, so analyze */  I
         IF OPTION_SEL(1) = '1'       /* the selector adjuncts */  J
           THEN CALL CPROC;           /* to see which field    */
    ELSE IF OPTION_SEL(2) = '1'       /* was selected.         */
           THEN CALL DPROC;
    ELSE IF OPTION_SEL(3) = '1'
           THEN CALL PPROC;
    ELSE PROCESS = '0'B;
  END;                                /* End l-pen inspection. */

  OTHERWISE INVALID = '1'B;           /* Invalid interrupt.    */  K
END;                                  /* End select group.     */
```

Figure 91 (Part 2 of 3). Listing of MAPEX08 source code

```
END;                                    /* End DO WHILE loop.    */

CALL FSTERM;                     /* Terminate GDDM.             */

%INCLUDE ADMUPINA;               /* GDDM entry declarations     */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINM;

END MAPEX08;
```

Figure 91 (Part 3 of 3). Listing of MAPEX08 source code

```
          1         2         3         4         5         6         7         8
 ----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
|                        #A C M E   O R D E R   E N T R Y#                       |
|¬                                                                              #|
|                                                                                |
|     SELECTION IS MADE BY :                                                     |
+                                                                                +
|          - USING THE INDICATED PROGRAM FUNCTION KEY (PFKN)                     |
|          - POSITIONING THE CURSOR, THEN PRESSING ENTER                         |
|          - USING THE LIGHT PEN (OR 'CURSR SEL' KEY)                            |
|                                                                                |
10     SELECT FROM :¬ #                                                          10
|                                                                                |
|          ¬ PFK10 = ORDER PROCESSING     #                                      |
|          ¬ PFK11 = DISPLAY CUSTOMER FILE#                                      |
|          ¬ PFK12 = PRINT CUSTOMER FILE  #                                      |
+          ¬ PFK3  = TERMINATE PROGRAM    #                                      +
 ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
          0         0         0         0         0         0         0         0
```

Figure 92. Field definitions of map used by MAPEX08

Specifying a PF key for alphanumeric input

The program end user's input is sometimes one of a number of predetermined character strings. GDDM provides a facility to save the end user having to type such strings. It is called **AID translation**. Its effect is to put a character string into a field when a PF key, or any other interrupt-generating key, such as a PA or the ENTER key, is pressed. In other words, GDDM translates an attention identifier (AID) into a character string.

You do all the necessary work when you define the map. Full details are given in *GDDM Interactive Map Definition* book. Briefly, you first define one or more tables that associate character-string values with selected PF keys. You do this using the GDDM-IMD table editor. Then, using the Application Data Structure Review frame of the map editor, you specify a table or tables to be associated with one or more fields in the map.

The receiving field for an AID translation string need not appear on the screen. You can simply add it to the ADS using GDDM-IMD's ADS Review frame.

When the map is displayed on the screen, and the end user presses a PF key, GDDM looks up each table specified for the map, to check if a character string has been specified for that PF key. Each field for which such a table has been specified has the associated character string inserted into it by GDDM.

The result is the same as if the end user had typed the character strings into the fields. The application has no way of telling that AID translation was used. Any selectors, and the parameters of ASREAD, return the same values as if the end user had typed the data into the field.

If your program needs to discover which PF key was pressed, it should inspect the values returned in the parameters of ASREAD. Whether or not AID translation was in use, these indicate which key caused the interrupt that satisfied the ASREAD.

In addition to PF keys, you can set up AID translation for any terminal facility that causes an interrupt, such as the CLEAR key or a magnetic card reader. The method is the same as for PF keys.

An example of using AID translation is shown in Figure 93. It is the same program as the one shown in Figure 81 on page 286, except that after receiving correct input, it redisplays the map, instead of terminating.

To terminate, the end user presses PF3 or PF15. An AID table was set up using the GDDM-IMD table editor, in which the character-string value END was associated with PF3 and PF15. Using the ADS Review frame of the map editor, this table was associated with the field called USER_FIELD. The result is that either PF3 or PF15 puts the character string END in this field. No field corresponding to USER_FIELD appears on the screen.

Changing the highlighting, color, and symbol sets of mapped fields

Often, and particularly when the end user has made a mistake, it is desirable to highlight a field or change its color. You can create adjuncts that enable your application to do this. As for other adjuncts, you create them using the Field Naming or Application Data Structure Review frame of GDDM-IMD's map editor. You can also create adjuncts to control the programmed-symbol set that fields use.

You can brighten a field by setting its intensity attribute, using the base attribute adjunct, as outlined in "The base attribute adjunct" on page 316. You can make it blink, or display reverse video or underscored characters, by the **extended highlighting adjunct**. You can change its color by the **color adjunct**. You can change its symbol set by the programmed symbol set adjunct, commonly called the **PS adjunct**.

In some circumstances, you may get undesirable visual effects with reverse video or underscored fields. Large areas of the screen may appear in reverse video or be underscored while the display is being built up. The remedy is to use character attributes (see "Changing the attributes of individual characters in a mapped field" on page 330), instead of the field attributes described here.

All three types of adjunct are two bytes long, and you use them in the same way as the base attribute adjunct. The codes you can put into the first byte are the same as for the base attribute adjunct, as described in "Protecting fields from the end user" on page 315. Briefly, " " (a blank character) or a 3 character means leave the attribute unchanged; a 1 character means use the attribute defined in the second byte; and a 2 character means use the map-defined attribute, or, if none was specified, the GDDM-IMD defined default.

```
MAPEX09:  PROC OPTIONS (MAIN);

DECLARE 1 CUSTINV,             /* Application Data Structure */

         10 USER_FIELD                    CHAR(3),
         10 MESSAGE                       CHAR(78),
         10 CUSTNO                        CHAR(5),
         10 INVNO                         CHAR(4),
         ORDER1_ASLENGTH                  FIXED BIN(31,0) STATIC
                                          INIT(90);
DECLARE (ATTYPE,ATVAL) FIXED BINARY(31,0);
                                          INIT(90);
CALL FSINIT;                    /* Initialize GDDM.           */
CUSTINV = '';                   /* Clear the ADS              */
LOOP:
                                /* Use MSREAD to display the  */
                                /* map, and wait for input.   */
CALL MSREAD('ACME00D6',         /* Mapgroup.                  */
            'ORDER1',           /* Map.                       */
            ORDER1_ASLENGTH,    /* Specify length of ADS.     */
            CUSTINV,            /* Specify name of ADS.       */
            ATTYPE,             /* Set to attention type ...  */
            ATVAL);             /* ... and value by GDDM      */

IF USER_FIELD='END'             /* If PF key 3 or 15 pressed, */
  THEN GO TO FIN;               /* end the program.           */
IF VERIFY(CUSTNO,'0123456789') = 0  /* Are CUSTNO and         */
 & VERIFY(INVNO,'0123456789') = 0   /* INVNO numeric?         */
  THEN DO;
    /*  .  */                   /* Process CUSTNO and INVNO   */
    /*  .  */
    /*  .  */
  MESSAGE = ' ';                /* Clear any existing message */
  END;
  ELSE MESSAGE = 'INVALID NUMBER'; /* If CUSTNO or INVNO not  */
                                   /* numeric, set up message.*/
GO TO LOOP;                     /* Redisplay the map and data.*/

FIN:
CALL FSTERM;                    /* Terminate GDDM.            */

%INCLUDE ADMUPINF;              /* GDDM entry declarations.   */
%INCLUDE ADMUPINM;

END MAPEX09;
```

Figure 93. Listing of MAPEX09 source code

The difference between a write operation and a rewrite (or reject) is the same as for the base attributes. A write resets the extended highlighting, color, and PS for all alphanumeric fields in the map, before the adjuncts are interpreted. It resets the attributes to their map-defined values, or the GDDM-IMD defined defaults where none were specified in the map. Rewrite and reject do not reset the attributes before the adjuncts are interpreted.

You set the second byte of the extended-highlighting adjunct to one of these values:

" " (Blank character) or X'00' – no extended highlighting
1 Blinking
2 Reverse video
4 Underscore.

The possible values for the second byte of the color adjunct are as follows:

0 Default (green on color displays, black on printers)
1 Blue
2 Red
3 Pink
4 Green
5 Turquoise
6 Yellow
7 Neutral (white on display, black on printers).

The values are the same as for the ASFCOL call. However, the parameter of the ASFCOL call is a fullword integer, whereas the second byte of the color adjunct is a character.

You set the second byte of the PS adjunct to the identifier of the required symbol set. This must contain image symbols of the same size as the device's hardware cells.

The symbol-set identifier can be assigned using the PS Management frame of the GDDM-IMD mapgroup editor. The symbol sets specified on this frame are loaded by GDDM when you execute an MSPCRT call specifying the mapgroup. Or you can load a symbol set dynamically and assign the symbol-set identifier, using the PSLSS call, which is described in "Loading symbol sets for alphanumeric text" on page 238. The information given in that section about loading symbol sets, device suffixes, and the use of PS stores for graphics applies to mapping, as well as to procedural alphanumerics.

On input, the selector (first) bytes of all three types of adjunct are set to 3.

Here is an example that uses a color adjunct to draw the end user's attention to invalid input and issues a message at the same time. The input is in the field called OPTION.

```
DCL 1 SELN,

          10 MESSAGE_FIELD_SEL              CHAR(1),
          10 MESSAGE_FIELD                  CHAR(78),
          10 OPTION_SEL                     CHAR(1),
          10 OPTION_COL_SEL                 CHAR(1),
          10 OPTION_COL                     CHAR(1),
          10 OPTION                         CHAR(1),
          SELN_ASLENGTH                     FIXED BIN(31,0) STATIC
                                            INIT(83);
DCL RED CHAR(1) INIT('2');
 /*   .   */
 /*   .   */
MESSAGE_FIELD_SEL = '1';
MESSAGE_FIELD     = 'INVALID OPTION - PLEASE CORRECT';
OPTION_COL_SEL    = '1';
OPTION_COL        = RED;

CALL MSPUT(1,1,SELN_ASLENGTH,SELN);
```

Changing the attributes of individual characters in a mapped field

In addition to setting the attributes for a field as a whole, you can set the color, highlighting, and symbol-set attributes for individual characters within the field.

First, you create one copy of the ADS for each type of attribute, in addition to the ADS that holds the variable data. To control just the colors of individual characters, for instance, you would need one additional copy of the ADS. To control all possible character attributes, you would need three.

In each field of each of these ADSs, you can put a string of attribute characters. The string has the same form as the third parameter of the ASCHLT, ASCCOL, and ASCSS calls, used to set character attributes in procedural alphanumeric calls. These are described in “Setting the attributes of alphanumeric characters” on page 76. Each attribute character specifies the attribute that the data character in the corresponding position of the field is to have. A blank attribute character means use the field attribute. Here is an example:

```
DATA.YEAR  = '1982';
COLOR.YEAR = '  22';
```

DATA is the name of the ADS that holds the variable data, and COLOR of the one that holds the color character attributes. The ADSs are identical, apart from these names. The two statements put the characters 1982 into the display. 19 has the color defined by the field attribute, and 82 the color 2, which, on a 3279 display, is red.

Suppose there is a further ADS, called HIGHL, that holds the highlighting attributes. The next statement assigns type 1 highlighting to the character 2, that is, make it blink:

```
HIGHL.YEAR = '   1';
```

After assigning the character attribute string to an ADS, you must execute an MSPUT call to update the page. The second parameter of MSPUT indicates which

type of attribute the ADS contains: a 3 character means highlighting, a 4 means color, and a 5 means PS. Typical calls would be:

```
CALL MSPUT(1,0,DATA_ASLENGTH,DATA);  /*Add variable data to map.*/
CALL MSPUT(1,3,DATA_ASLENGTH,HIGHL); /*Add highlight char. attr.*/
CALL MSPUT(1,4,DATA_ASLENGTH,COLOR); /*Add color character attr.*/
```

The data-assigning MSPUT (type 0, 1, or 2) clears all character attributes. It must therefore be executed before any attribute-setting MSPUT calls (type 3, 4, or 5).

You can simplify the declarations of the ADSs by putting them all into an array of structures (in PL/I terms). For instance:

```
DCL 1 EXAMPADS(4),
%INCLUDE EXAMPMAP;

YEAR(1) = '1982';                          /* Add data          */
CALL MSPUT(1,0,EXAMPMAP_ASLENGTH,YEAR(1)); /* to page.          */

YEAR(2) = '   1';                          /* Make last         */
CALL MSPUT(1,3,EXAMPMAP_ASLENGTH,YEAR(2)); /* character blink.  */

YEAR(3) = '  22';                          /* Change last two   */
CALL MSPUT(1,4,EXAMPMAP_ASLENGTH,YEAR(3)); /* characters to red */
```

The fourth structure would be used for PS character attributes.

You do not need separate copies of the ADS for the character attributes. You could reuse the one used for the variable data, like this:

```
DCL 1 EXAMPADS,
%INCLUDE EXAMPMAP;

YEAR = '1982';
CALL MSPUT(1,0,EXAMPMAP_ASLENGTH,YEAR);

YEAR = '   1';
CALL MSPUT(1,3,EXAMPMAP_ASLENGTH,YEAR);

YEAR = '  22';
CALL MSPUT(1,4,EXAMPMAP_ASLENGTH,YEAR);
```

The ADSs that you use for character attributes can contain adjuncts of all types. Selector adjuncts control the fields' character attributes. The codes are similar to those that you put into the ADS containing the data. They are:

blank
: Ignore the character-attribute string; in other words, leave the character attributes unchanged.

1
: Take the character attributes from the ADS.

2
: Use all-blank character-attribute characters. This causes the field attributes to apply to all characters.

Type 3, 4, and 5 MSPUT calls act in a similar way to a type 1 (rewrite) call: there is no resetting of character attributes before the selector adjuncts are interpreted.

Other adjuncts have exactly the same effects as in a rewrite operation. Base attribute adjuncts control the base field attributes, cursor adjuncts control the position of the cursor, and so on.

Discovering which character attributes have been changed by the user

Character attributes can be changed by the end user. To get information about these changes, you must first allow the input of character attributes to the current page by executing a CALL ASMODE(2) statement. You can then set the second parameter of the MSGET call to tell GDDM to return information about character attributes in the ADS. The permissible values of this parameter and their meanings are:

0 Supply information about the data. All the previous examples of MSGET calls in this guide use this value.

3 Supply information about highlighting character attributes.

4 Supply information about color character attributes.

5 Supply information about PS character attributes.

A type 3, 4, or 5 MSGET call updates the specified ADS with the current character attributes of all variable data characters. It also sets adjuncts in the same way as a type 0 MSGET. The meanings of selector adjunct codes on input are, for each type of attribute:

blank No character attributes of this type have been set for this field.

1 The field has character attributes of this type that were set by the end user in the last ASREAD.

3 The field has character attributes of this type, and they were set either in an earlier ASREAD or by the program.

Folding and justification of input

Your programs may be simplified if you can assume that a field contains only uppercase letters, and has no leading blanks or no trailing blanks.

When you define a field to GDDM-IMD, you can specify that GDDM is to fold lowercase letters to uppercase on input. Similarly you can specify that the data in the field is to be either left- or right-justified. Left-justification removes leading blanks, and right-justification the trailing ones.

You specify folding and justification on the Field Naming or Application Data Structure Review frame of GDDM-IMD's map editor.

Mapping and graphics

You can display mapped data and graphics together, and you can use GDDM's interactive graphics facilities on mapped pages. There are two ways of putting graphics onto mapped pages.

One way is simply to define a graphics field on a mapped page using the GSFLD call (see “The graphics field and the image field” on page 112). If you use this method, it is inadvisable to let any graphics overlap a mapped area of the page, because the results are unpredictable.

The other way is to specify to GDDM-IMD an area for graphics within a map, called a **graphic area**. After an MSDFLD call specifying such a map, the graphic area becomes the graphics field.

You define the graphic area on the Field Definition frame of GDDM-IMD's map editor. You enter an AREA command, specifying the graphic area's size and position in rows and columns. GDDM-IMD shows the graphic area by filling it with % signs, or some other specified symbol.

Whatever the method of creation, GDDM never allows more than one graphics field on a page.

There is always a column of blank spaces one character wide down the left-hand edge of a graphics area. This is because each row of the graphics area starts with an attribute byte, to prevent the attributes of any preceding alphanumeric fields from interfering with the graphics. It has the effect of making the width of the graphics field one character less than that specified to GDDM-IMD.

In a dual-screen configuration of the IBM 3270-PC/GX workstation, the graphics appear on the graphics screen, and the maps appear on the alphanumerics screen. The graphics occupy the same part of the screen as they would in a single-screen configuration. On the IBM 5080 graphics system, the graphics field fills the graphics monitor, and the maps appear on the 3270 screen.

Remember that the depth and width of the graphic area are specified in rows and columns, not physical dimensions. An equal number of rows and columns does not give a square graphic area. This may lead to your graphics having unexpected proportions: circles appearing as ovals and squares as rectangles. One solution is to create a uniform set of world coordinates by issuing a GSUWIN call before opening any graphics segment:

```
CALL GSUWIN(-100.0,100.0,-100.0,100.0);
```

More information is given in “Setting up a coordinate system for drawing graphics” on page 28 and “Uniform world coordinates” on page 118.

Graphics cannot be used with MSREAD, because this call creates, transmits, and discards a page without providing an opportunity for the program to create graphics on it.

Example of graphics in a mapped display

The program shown in Figure 94 on page 334 provides the terminal operator with a menu from which a shape and a color can be selected. The program draws the chosen shape in the chosen color. The format of the map it uses is shown in Figure 96 on page 337. A typical display is shown in Figure 95 on page 337.

The program uses several calls, marked **A**, that refer to the graphics concepts of segment and picture space. The concepts are described in Chapter 7, “Hierarchy of GDDM concepts” on page 107.

```
MAPEX11:  PROC OPTIONS (MAIN);

DCL 1 DRAW,                        /* Application Data Structure */
        10 MESSAGE_FIELD_SEL              CHAR (1),
        10 MESSAGE_FIELD                  CHAR (30),
        10 SHAPE_ARRAY(3),
         15 SHAPE_SEL                     CHAR(1),
         15 SHAPE                         CHAR(11),
        10 COLOR_ARRAY(7),
         15 COLOR_SEL                     CHAR(1),
         15 COLOR                         CHAR(12);
        DRAW_ASLENGTH                     FIXED BIN(31,0) STATIC
                                          INIT(158);
DCL (ATTYPE,ATVAL,COUNT) FIXED BIN(31); /*   ASREAD arguments   */
DCL OPERATION FIXED BIN(31);         /* Type of output required */
DCL WRITE   FIXED BIN(31) INIT(0);   /* MSPUT write operation   */
DCL REJECT  FIXED BIN(31) INIT(2);   /* MSPUT reject operation  */
DCL SHAPE_CHOSEN FIXED BIN(31);      /* Identifies chosen shape */
DCL COLOR_CHOSEN FIXED BIN(31);      /* Identifies chosen color */
DCL ERROR FIXED BIN(15) INIT(0);     /* Indicates type of error */
DCL 1 MSG(4) CHAR(30) INIT(          /* Error messages          */
        'NO SELECTIONS MADE - RETRY',
    'CONFLICTING SELECTIONS - RETRY',
          'SHAPE NOT CHOSEN - RETRY',
         'COLOR NOT CHOSEN - RETRY');
DCL (I,J) FIXED BIN(15);             /* Work variables          */
CALL FSINIT;                         /* Initialize GDDM         */
CALL MSPCRT(1,                       /* Create page using       */
            -1,                      /* GDDM-IMD defined page   */
            -1,                      /* width and depth         */
            'DRAWD6');               /* for mapgroup DRAWD6.    */
CALL MSDFLD(1,                       /* Format an area of the   */
            -1,                      /* page at GDDM-IMD defined*/
            -1,                      /* row and column          */
            'DRAW');                 /* for map draw.           */
```

Figure 94 (Part 1 of 3). Listing of MAPEX11 source code

```
DRAW = '';                          /* Clear ADS.              */
OPERATION = WRITE;                  /* Initially use write.    */

CALL GSPS(1,1);                     /* Set picture space aspect*/  A
                                    /* Ratio to 1:1            */
CALL GSSEG(1);                      /* Define graphics segment.*/  A

PUT_MAP:
CALL MSPUT(1,                       /* Add data to map         */
           OPERATION,               /* with preset operation,  */
           DRAW_ASLENGTH,           /* specifying the ADS      */
           DRAW);                   /* length & the data area. */
CALL ASREAD(ATTYPE,                 /* Send page to terminal & */
            ATVAL,                  /* wait for operator input.*/
            COUNT);
IF ATTYPE = 1                       /* PF key 3 or 15 pressed, */
& (ATVAL = 3 | ATVAL = 15)          /* so terminate.           */
  THEN GO TO EXIT;
CALL GSCLR;                         /* Clear the segment.      */  A
CALL GSSEG(1);                      /* Define graphics segment.*/  A
IF COUNT = 0 THEN DO;               /* No data input - error.  */
  ERROR = 1;
  GO TO REJECT_MAP;
END;
CALL MSGET(1,0,                     /* Get data from map.      */
           DRAW_ASLENGTH,           /* Length of data area.    */
           DRAW);                   /* Data area.              */
MESSAGE_FIELD_SEL = ' ';            /* Remove any error message*/
DO I = 1 TO 3;                      /* Check if shape chosen.  */
  IF SHAPE_SEL(I) = '1' THEN DO;    /* Shape has been chosen.  */
    DO J = I+1 TO 3;                /* Is it unique?           */
      IF SHAPE_SEL(J) = '1' THEN DO; /* No, so indicate error. */
        ERROR = 2;
        GO TO REJECT_MAP;
      END;
    END;
    SHAPE_CHOSEN = I;               /* Store chosen shape.     */
    GO TO CHECK_COLOR;
  END;
END;
ERROR = 3;                          /* No shape chosen.        */
GO TO REJECT_MAP;
CHECK_COLOR:
DO I = 1 TO 7;                      /* Check if color chosen.  */
  IF COLOR_SEL(I) = '1' THEN DO;    /* Color has been chosen.  */
    DO J = I+1 TO 7;                /* Is it unique?           */
      IF COLOR_SEL(J) = '1' THEN DO; /* No, so indicate error. */
        ERROR = 2;
        GO TO REJECT_MAP;
      END;
    END;
    COLOR_CHOSEN = I;               /* Store chosen color.     */
    GO TO PUT_GRAPHICS;
  END;
END;
```

Figure 94 (Part 2 of 3). Listing of MAPEX11 source code

```
ERROR = 4;                          /* No color chosen.        */
REJECT_MAP:                         /* Set up reject of map.   */
MESSAGE_FIELD_SEL = '1';            /* Set selector adjunct.   */
MESSAGE_FIELD = MSG(ERROR);         /* Move in message.        */
ERROR = 0;                          /* Clear indicator.        */
OPERATION = REJECT;                 /* Specify reject operation*/
DO I = 1 TO 3;                      /* Set the selector        */
  IF SHAPE_SEL(I) ¬= '1'            /* adjuncts to take        */
    THEN SHAPE_SEL(I) = '2';        /* map-defined values.     */
END;
DO I = 1 TO 7;                      /* Set the selector        */
  IF COLOR_SEL(I) ¬= '1'            /* adjuncts to take        */
    THEN COLOR_SEL(I) = '2';        /* map-defined values.     */
END;
GO TO PUT_MAP;
PUT_GRAPHICS:                       /* Create chosen shape.    */
CALL GSCOL(COLOR_CHOSEN);           /* Set color.              */
CALL GSAREA(0);                     /* Start an area.          */
IF SHAPE_CHOSEN = 1 THEN DO;        /* Circle selected.        */
  CALL GSMOVE(2,50);                /* Move to center.         */
  CALL GSARC(50,50,360);            /* Draw arc.               */
END;
ELSE IF SHAPE_CHOSEN = 2 THEN DO;   /* Square selected.        */
  CALL GSMOVE(0,0);                 /* Move to initial position*/
  CALL GSLINE(100,0);               /* Draw                    */
  CALL GSLINE(100,100);             /*      sides              */
  CALL GSLINE(0,100);               /*            of           */
  CALL GSLINE(0,0);                 /*               square.   */
END;
ELSE DO;                            /* Triangle selected.      */
  CALL GSMOVE(0,0);                 /* Move to initial position*/
  CALL GSLINE(100,0);               /* Draw                    */
  CALL GSLINE(50,100);              /*      three              */
  CALL GSLINE(0,0);                 /*            lines.       */
END;
CALL GSENDA;                        /* Close the area.         */
OPERATION = WRITE;                  /* Specify write operation.*/
SHAPE_SEL = ' ';                    /* Clear selector          */
COLOR_SEL = ' ';                    /* adjuncts.               */
GO TO PUT_MAP;                      /* Redisplay the panel.    */
EXIT:
CALL FSTERM;
%INCLUDE ADMUPINA;                  /* GDDM entry declarations.*/
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
%INCLUDE ADMUPINM;
END MAPEX11;
```

Figure 94 (Part 3 of 3). Listing of MAPEX11 source code

35SC0148D9

Figure 95. Typical display by MAPEX11

```
           1         2         3         4         5         6         7         8
  ----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 |-------------------------#Mapping with Graphics Area#-------------------------|
 |¬                            #                                                |
 | Select the shape to be drawn and                                             |
 | the color it is to be drawn in using                                         |
 + the LIGHT PEN (or CURSR SEL key).                                            +
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |#Shape  :#                       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
10   ¬?  Circle  #                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  Square  #                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  Triangle#                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 +#Color :#                        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  Blue                      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  Red                       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  Pink                      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
20   ¬?  Green                     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  Turquoise                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  Yellow                    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |   ¬?  White                     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 +                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |Use PF3/15 to End                %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
30                                 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  |
 |                                                                              |
 |                                                                              |
  ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
           0         0         0         0         0         0         0         0
```

Figure 96. Field definitions of map used by MAPEX11

Chapter 17. Using GDDM's advanced image functions

This section deals with the following tasks:

- Querying image devices
- Converting a gray-scale image to binary data
- Querying image-related device characteristics
- Scaling an image to fit the display screen
- Interactive manipulation of an image
- Transferring an image into or out of your program
- Controlling host offload by specifying image quality
- Direct transmission from a scanner, and to a 3193
- Combining an image with text or graphics
- Printing an image
- Improving the performance of image-processing programs
- Device variations

Querying image devices

The current status of the scanner (whether it is switched on, ready, or jammed) can be queried using the ISQSCA call. However, the status is detected and set for querying by GDDM at the time it implicitly opens the scanner, and at each transfer operation that has the scanner as its source, **not** dynamically at the time the ISQSCA call is issued. It can be used therefore as a check on the scanner status only after such transfer operations.

The ISQSCA call and its parameter settings for various states of the scanner and automatic document feeder (ADF) are described fully in the *GDDM Base Application Programming Reference* book.

Some scanner error conditions, such as power off, cause a GDDM error message such as

```
ADM3477 E SCANNER NOT READY, MAY BE POWERED OFF
```

to be issued as a result of transfer operations that have the scanner device image as their source. Such error messages can be detected using the general GDDM error-handling technique described in "Querying the GDDM error record, using FSQERR" on page 133. Error recovery is then possible by instructing the terminal user to correct the scanner error, and restarting the program.

Some scanner configuration and basic characteristics, such as whether a scanner is attached, the maximum scan area in pixels, and scanner type (flat bed or roller feed), can be queried using the FSQURY call.

© Copyright IBM Corp. 1982, 1996

Here is an example of its use:

```
DCL ARRAY(1)  FIXED BIN(31); /* Array for returned
                                characteristics              */
CALL FSQURY(5,1,1,ARRAY);    /* Query scanner (Code=5)       */
IF ARRAY(1)=1
  THEN DO;                   /* Scanner is attached          */
    ... (scanner initialization)
       END;
  ELSE                       /* Scanner is not attached      */
    ... (notify end user)
```

You can also use FSQURY to query several scanner or 3193 device characteristics. The call and its parameters are fully described in the *GDDM Base Application Programming Reference* book.

Scanning gray-scale images

GDDM supports the scanning of gray-scale images, such as photographs, but your programs may need to perform special transform functions to ensure a good representations as bi-level (black and white) image.

Chapter 6, “Image basics” on page 85 described projections, and introduced most of the image transform calls, that have the format IMRxxx. They are used to define image transform sequences that can be invoked in image transfer operations.

The three remaining transform functions and their calls control these algorithm definitions:

- Brightness conversion
- Contrast conversion
- Image type conversion

Before doing this, you define **gray-scale** and **halftone** (monochrome) images.

A **gray-scale** image is one in which the gradations between black and white are represented by discrete **gray-levels**, commonly coded 0 through 255. This is a representation amenable to digital image processing. Each pixel therefore has a value in the range 0 through 255.

A **halftone** or **bi-level** image is one in which each pixel is simply either black or white (value 0 or 1), and the intermediate shades of gray are achieved by pixel groups of mixed black and white–in effect shading patterns.

In GDDM, the only permitted gray-scale images are those on paper, at input to the scanner. The range 0 through 255, although not fully supported by GDDM, is used below merely to illustrate the workings of the algorithms.

Defining brightness conversion definition, using call IMRBRI

You can use the IMRBRI call to lighten or darken gray-level images only. It has no effect on bi-level images. Here is an example call to darken a 3118 scanner image:

```
DCL ARRAY(1) FLOAT DEC(6);            /* Array of conversion factors  */
ARRAY(1)=-0.1;
/*       proj-id  alg count  alg-data                                 */
CALL IMRBRI(20,     0,   1,   ARRAY);  /* Define brightness conversion */
```

The parameters are as follows:

proj-id The first parameter, 20, is the projection identifier.

alg Specifies the algorithm to be used. Here, 0 specifies the default algorithm. This is device-dependent. For 3117 and 3118 scanners it is the same as specifying 1, which selects a simple linear brightness conversion algorithm, explained below.

count The third parameter, specifies the number of elements, 1, in the array parameter that follows.

alg-data The name of the array of conversion algorithm factors.

The linear brightness algorithm defines the new gray-level of any pixel in terms of the old value of that same pixel as:

```
new = old + (ARRAY(1) * white)
```

where white is the maximum gray-level, for example 255. The ARRAY(1) value specifies the required change in the brightness level as a number in the range –1 through +1, where –1 is totally dark, 0 no change, and +1 is totally light.

For example, consider a pixel with an old gray-level value of 150. The new value, for the call above, is:

```
150 + (-0.1 * 255) = 125
```

A negative value as the conversion factor reduces the gray-level, and so darkens the image; a positive value brightens the image.

The 3117 and 3118 scanners provide three brightness levels only; which one is used depends on the value of ARRAY(1) as follows:

–1.0 through –0.5	Darken the image (use for light original).
>–0.5 through <0.5	No change (use for normal original)
0.5 through 1.0	Lighten the image (use for dark original)

Defining contrast conversion, using call IMRCON

You can use the IMRCON call to change the contrast of gray-scale images only. It has no effect on bi-level images. Here is an example call to increase the contrast of a scanner image:

```
DCL ARRAY(1) FLOAT DEC(6); /* Array of conversion factors       */
ARRAY(1)=2;
CALL IMRCON(15,1,1,ARRAY);
```

The parameters are as follows:

- The first parameter, 15, is the projection identifier.
- The second parameter, 1, specifies a linear contrast conversion algorithm, explained below. For the 3117 and 3118 scanners, this is also the default, which could have been specified by coding 0 instead of 1.

- The next parameter, 1, is a count value, giving the number of elements used in the array parameter.
- The last parameter specifies the name of the array giving the conversion algorithm factors.

The linear contrast conversion algorithm is:

```
new = ((old - mean) * ARRAY(1)) + mean
```

where old and new are the old and new gray-level values of a given pixel, and mean is the mid-point between black and white. For the example range 0 through 255, the value of mean is therefore 128.

So for an old gray-level value of 90, the new value, for the call as coded above, is:

```
(( 90 - 128) * 2) + 128 = -76 + 128
                        = 52
```

The 3117 and 3118 scanners provide three contrast values only; which one is used depends on the value of ARRAY(1) as follows:

0 through 0.5	Decrease the contrast
>0.5 through <2.0	No change
≥2.0	Increase the contrast

Defining the conversion algorithm, using call IMRCVB

You can use IMRCVB to specify a particular conversion process between gray-scale and bi-level (halftone) images. Here is an example, specifying that conversion algorithm 10, which is halftoning type A, is to be used:

```
DCL ARRAY(1) FLOAT DEC(6);           /* Array for conversion factors  */
/*       proj-id alg  count  alg-data                                */
CALL IMRCVB(3,    10,   0,    ARRAY); /* Define conversion to bi-level */
```

The parameters are as follows:

- The first parameter, 3, is, as usual, the projection identifier.
- The second parameter, 10, specifies halftoning type A. This is best for intricate pictures.

  Other possible values are:

  `0` Device-dependent (the default). For the 3117 and 3118 this is the same as 1.

  `1` Threshold. A threshold is defined for comparison with each source pixel. Pixels above the threshold gray-level specified in ARRAY(1) become white and below it become black.

  `11` Halftoning type B, best when gray-levels vary gradually.
- The next parameter is a count, specifying the number of elements in the array parameter that follows. You are recommended to use a value of 0 if you are specifying the default algorithm in the second parameter.
- The last parameter is the array of factors, if any, for the specified algorithm.

  For algorithm 1, ARRAY(1) specifies the required threshold level as a number in the range 0 through 1, where 0 is black and 1 is white. The default threshold is 0.5.

The 3117 and 3118 scanners provide three threshold levels only, depending on ARRAY(1) values as follows:

0 through 0.25	Dark original
>0.25 through <0.75	Normal original
0.75 through 1.0	Light original.

For algorithm 10 or 11, the fourth parameter is not used.

Ordering of brightness, contrast, and image type conversion calls

The order of IMRBRI, IMRCON, and IMRCVB calls is significant. An IMRBRI or IMRCON call following an IMRCVB call has no effect, because it is applied to the bi-level image resulting from the IMRCVB call. So, if you do need brightness or contrast conversion, code the IMRBRI or IMRCON call before the IMRCVB call.

Querying image-related device characteristics

You have met the FSQURY call for general device queries, and the ISQSCA call for querying image scanner readiness status. Three more query calls are introduced here , to determine the image data formats, compression algorithms, and resolution values supported by scanner, display, printer, or plotter devices.

Note: Pen plotters are not recommended for image output.

Firstly on data formats and compressions–these are of particular concern in image processing, because of the frequently large volumes of the data compared with alphanumerics or graphics data streams. But note that when sending or receiving data in a device-supported format and compression, the formatting is performed by the device, not by GDDM. If the data is in a format not supported by the device, GDDM converts the data automatically in the host.

Querying formats supported by a device, using call ISQFOR

You can use the ISQFOR call to query the format(s) supported by a display-attached scanner. You can use values other than the supported ones, but with reduced performance in the host, as GDDM automatically converts to or from the format you specify.

```
DCL ARRAY(3) FIXED BIN(31);
DCL DEV FIXED BIN(31);

CALL FSQURY(5,8,1,ARRAY);         /* Query number of formats    */
                                  /* Supported by image scanner */
                                  /* (for image display use     */
                                  /*  FSQURY(4,4,...and so on)  */
DEV=-1;                           /* Device is a scanner        */
COUNT=ARRAY(1);
CALL ISQFOR(DEV,COUNT,ARRAY);     /* Query supported formats    */
DO I=1 TO COUNT;
 IF ARRAY(I)=1
  THEN ... unformatted data is supported
  ELSE IF ... and so on.
END;
```

The parameters of the ISQFOR call are as follows:

- The first parameter, –1, specifies the device to be the scanner.

- The second parameter is a count specifying the number of elements required to be returned in the array parameter.
- The last parameter is the array in which GDDM is to return the supported format codes, that can have the following values:

This call queries the formats supported by a scanner and GDDM returns codes from the following range of values in the third parameter:

1 Unformatted data
2 3193 data-stream structures
3 CPDS structures.

The image format(s) thus determined can then be used in a routine sending or retrieving an image data object. This topic is dealt with below (see “Transferring images into and out of your program” on page 355).

Querying compressions supported by a device, using call ISQCOM

You can use the ISQCOM call to query the compression algorithm(s) supported by the current primary output device (display, printer, or plotter). You can use values other than the supported ones, but doing so lowers performance in the host, as GDDM automatically converts to or from the compression you specify.

```
DCL ARRAY(4) FIXED BIN(31);
DCL DEV FIXED BIN(31);

CALL FSQURY(4,3,1,ARRAY);         /* Query number of compressions*/
                                  /* Supported by image display */
                                  /* (for image scanner use     */
                                  /*  FSQURY(5,7,....and so on) */
DEV=0;                            /* Current primary device     */
COUNT=ARRAY(1);
CALL ISQCOM(DEV,COUNT,ARRAY);
DO I = 1 TO COUNT;
  IF ARRAY(I)=1
    THEN .....   uncompressed data is supported
    ELSE IF ... and so on.
END;
```

The parameters are as follows:

- The first parameter, 0, specifies the display, printer or plotter device (whichever is the current primary device).

  Alternatively, –1 would specify the display-attached scanner.
- The second parameter, COUNT, specifies the number of elements in the array parameter.
- In the array named in the third parameter, GDDM returns the supported compression codes from the following range of values:

  1 Uncompressed
  2 MMR
  3 4250
  4 3800.

Querying resolutions supported by a device, using call ISQRES

The ISQRES call has a complex parameter list. Here is an example of its use, to query the scanner resolutions, if any, that are nearest to, and greater than or equal to the values 100 pixels per inch horizontally and 150 pixels per inch vertically:

```
DCL (H_RES,V_RES) FLOAT DEC(6);
DCL INFO FIXED BIN(31);
CALL ISQRES(-1,  0,   1, 100, 1, 150, H_RES, V_RES, INFO);
```

The parameters are as follows:

- The first parameter specifies the device: –1 for a scanner and 0 for the current primary device (display, printer, or plotter).
- The next parameter, 0, specifies inch units for the resolution values in later parameters. 1 would specify meters.
- The next two pairs of parameters (1, 100 and 1, 150) each specify a relation and a reference value, for horizontal and vertical resolutions respectively. The first number, 1, in each pair, requests return of a value that is nearest to and greater than or equal to the reference value that follows (100 for the horizontal and 150 for the vertical).

  Other possible values and meanings for the relation parameter are:

  `-2` Nearest to and less than
  `-1` Nearest to and less than or equal to
  `0` Nearest to
  `2` Nearest to and greater than.
- The next two parameters, H_RES and V_RES, are the variables in which GDDM returns the horizontal and vertical resolution values meeting the specified relationships with the reference values.

  If for example, the scanner being queried by the example call had a choice of pairs of horizontal and vertical resolutions of (120,120), (240,240), and (240,120), all in pixels per inch units, the returned values in H_RES and V_RES would be 240 and 240 respectively.
- The last parameter, INFO, returns further information about the values returned in H_RES and V_RES, as follows:

  `0` The returned values are a specific pair of supported resolutions.
  `1` Any resolution is supported, in which case the returned resolution values would be equal to the reference values specified in the earlier parameters.

If no supported resolution meets the requirement specified, a value of 0 is returned in H_RES, or V_RES, or both, as appropriate.

It would, therefore, be normal to follow the ISQRES call with statements such as:

```
IF (H_RES=0) | (V_RES=0)
  THEN DO;
          ....... error handling
       END;
```

You could go on to initialize the scanner with the returned values, for a specified paper size (8 inches wide by 11 inches deep for instance), using the IMACRT call, as follows:

```
CALL IMACRT(-1, 8*H_RES, 11*V_RES, 0, 1, 0, H_RES, V_RES);
```

Scaling an image to fit the display screen

If an image is scanned, saved, restored, and displayed using identity projections throughout, it is displayed at real size, that is, at the same size as the original image on paper. It is also displayed with the top left corner of the original image aligned with the top left corner of the image field, and truncated if necessary at the bottom and right edges.

This may not be what you require. You may want to scale the original image up or down to just fill the display screen (or, more generally, the image field) in either the horizontal or vertical dimension as appropriate, while maintaining the correct aspect ratio.

Note: On graphics terminals other than the 3193, GDDM's User Control facility allows end users of application programs to pan and zoom image fields.

This programming example shows you how to change the scale of an image:

```
IMPROG4: PROC OPTIONS(MAIN);
DCL MIN BUILTIN;
DCL APPL_ID FIXED BIN(31);     /* Application image identifier */
DCL PROJ_ID FIXED BIN(31);     /* Projection identifier        */
DCL H_PIXELS FIXED BIN(31);    /* Application image horizontal */
                               /* size in pixels               */
DCL V_PIXELS FIXED BIN(31);    /* Application image vertical   */
                               /* size in pixels               */
DCL DH_PIXELS FIXED BIN(31);   /* Display image  horizontal    */
                               /* size in pixels               */
DCL DV_PIXELS FIXED BIN(31);   /* Display image  vertical      */
                               /* size in pixels               */
DCL IM_TYPE FIXED BIN(31);     /* Image type                   */
DCL RES FIXED BIN(31);         /* Defined/undefined resolutn.  */
DCL RES_UNIT FIXED BIN(31)     /* Resolution                   */
             INIT(0);          /* Units to be inches           */
DCL H_RES FLOAT DEC(6);        /* Application image horizontal */
                               /* resolution (pixels per inch) */
DCL V_RES FLOAT DEC(6);        /* Application  image  vertical */
                               /* resolution (pixels per inch) */
DCL DH_RES FLOAT DEC(6);       /* Display image  horizontal    */
                               /* resolution (pixels per inch) */
DCL DV_RES FLOAT DEC(6);       /* Display image  vertical      */
                               /* resolution (pixels per inch) */
DCL H_SIZE FLOAT DEC(6);       /* Application image hor. size  */
DCL V_SIZE FLOAT DEC(6);       /* Application image ver. size  */
DCL DH_SIZE FLOAT DEC(6);      /* Display image horiz. size    */
DCL DV_SIZE FLOAT DEC(6);      /* Display image vert. size     */
```

Figure 97 (Part 1 of 2). Program that scales an image to fit the display screen

```
DCL H_RATIO FLOAT DEC(6);      /* Horizontal and vertical      */
DCL V_RATIO FLOAT DEC(6);      /* size ratios of display       */
                               /* image to appln. image        */
DCL SCALE FLOAT DEC(6);        /* Scale factor                 */
DCL (ATTYPE,ATTVAL,COUNT)      /* ASREAD parameters            */
       FIXED BIN(31);
DCL DESCR CHAR(30);            /* IMARST parameter             */

CALL FSINIT;
CALL IMAGID(APPL_ID);
CALL IMARST(APPL_ID,0,'IMAGNAME',30,DESCR);/*Restore saved     */  A
                               /* image to application image   */

                               /* Query the application image  */
CALL IMAQRY(APPL_ID,H_PIXELS,V_PIXELS,IM_TYPE,                     B
            RES,RES_UNIT,H_RES,V_RES);

                               /* Query the display image      */
CALL IMAQRY(0,DH_PIXELS,DV_PIXELS,IM_TYPE,                         C
            RES,RES_UNIT,DH_RES,DV_RES);

H_SIZE=H_PIXELS/H_RES;         /* Application image size inches*/
V_SIZE=V_PIXELS/V_RES;         /* (horizontal and vertical)    */
DH_SIZE=DH_PIXELS/DH_RES;      /*Display image size in inches  */
DV_SIZE=DV_PIXELS/DV_RES;      /*(horizontal and vertical)     */
H_RATIO=DH_SIZE/H_SIZE;        /* Size ratios of display       */
V_RATIO=DV_SIZE/V_SIZE;        /* image to application image   */
SCALE=MIN(H_RATIO,V_RATIO);    /* Required scale factor        */  D
CALL IMPGID(PROJ_ID);          /* Get a projection identifier  */
CALL IMPCRT(PROJ_ID);          /* Create a new projection      */
CALL IMRSCL(PROJ_ID,SCALE,SCALE);/* Scale the image to suit    */
CALL IMRPLR(PROJ_ID,0,0.0,0.0,0);/*End of projection definition*/
CALL IMXFER(APPL_ID,0,PROJ_ID);  /* Transfer to display screen */  E
CALL ASREAD(ATTYPE,ATTVAL,COUNT);
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;
END IMPROG4;
```

Figure 97 (Part 2 of 2). Program that scales an image to fit the display screen

In the above program, at **A**, a previously saved image IMAGENAME is restored using the identity projection.

At **B**, the application image attributes are queried, to obtain sizes in pixels, and resolutions.

At **C**, attributes of the image field on the current GDDM page are similarly queried. By default, this image field occupies the entire display screen.

At **D**, the lesser of the horizontal and vertical size ratios (of display image to application image) is assigned to SCALE, that is subsequently used as the horizontal and vertical scale factor in a projection definition. This calculation works only for images with defined resolution.

At **E**, this projection is applied to the restored image as it is transferred to the display screen.

Interactive image manipulation, using image cursors

This section has the following subsections:

- The FSENAB, ISENAB, ISQLOC, and ISQBOX calls
- Initializing the image cursors, using calls ISILOC and ISIBOX
- Local operations on the 3193 display station
- Interactive image manipulation example.

The 3193 provides three cursors; one alphanumeric cursor and two image cursors – an image cross cursor and an image rectangle (box) cursor.

The image cross cursor is used to inform a host program of an operator-selected point on the screen. In GDDM terms it is a locator cursor.

The image box cursor is used to inform a host program of an operator-selected rectangular area of the screen.

Image cursors must be enabled before they can be used. The normal technique is to enable whichever of the two image cursors you decide to use, and not both (although this is permitted). In GDDM image processing, unlike graphics, there is no concept of an image input queue.

Enabling or disabling device input, using call FSENAB

This is not an image-specific call, but is required for the use of image cursors. Here is an example of its use in this context:

```
CALL FSENAB(3,1);                /* Enable device input       */
```

- The first parameter specifies the input type, 3 for image.
- The second parameter is 0 for disable, 1 for enable.

Enabling or disabling an image cursor, using call ISENAB

This call is required to enable or disable a specific image cursor, that is, either the cross cursor or the box cursor. Here is an example call that enables the image cross cursor:

```
CALL ISENAB(1,1);                /* Enable image cross cursor */
```

The first parameter specifies the type of image cursor: 1 for the cross cursor, 2 for the box cursor.

The second parameter is 0 for disable, 1 for enable.

Enabling a cursor makes it appear on the display screen, and it can be moved around the screen under control of the terminal user. The call would normally be followed by an ASREAD, GSREAD, or MSREAD call, to wait for operator repositioning of the cursor, after which your program can query the cursor position by use of an ISQLOC or ISQBOX call. Disabling a cursor makes it disappear from the screen.

Querying the image locator cursor, using call ISQLOC

Here is an example of how the ISQLOC call would be used:

```
DCL (ECHO,              /* Reserved parameter                  */
     H_POS,V_POS,       /* Horizontal and vertical position in  */
                        /* pixels                              */
     IN_IMAGE,          /* In/out of image indicator           */
     STATUS)            /* Enabled/disabled indicator          */
    FIXED BINARY(31);
DCL (TYPE,MOD,COUNT)    /* ASREAD parameters                   */
    FIXED BINARY(31);
CALL FSENAB(3,1);       /* Enable image input                  */
CALL ISENAB(1,1);       /* Enable cross cursor                 */
CALL ASREAD(TYPE,MOD,COUNT); /* Output and wait for input      */
CALL ISQLOC(ECHO,H_POS,V_POS,IN_IMAGE,STATUS);
                        /* Query cursor position               */
CALL ISENAB(1,0);       /* Disable cross cursor                */
```

The ISQLOC parameters are as follows:

- The first parameter always returns a value of 0.
- The next two parameters return the horizontal and vertical position, in pixels, of the cross cursor.
- The next parameter returns a value of 0 if the cursor is outside the image, or 1 if it is within the image, on the current GDDM page.
- The last parameter returns a value 0 if the cursor is disabled, or 1 if it is enabled. (You can use the ISQLOC call with the cursor disabled.)

Querying the image box cursor, using call ISQBOX

This call is used similarly to the ISQLOC call, for querying the position, size, and status of the image box cursor. Here is an example of its use:

```
DCL (ECHO,              /* Reserved parameter                  */
     LEFT_EDGE,         /* Left edge of the rectangle in pixels */
     RIGHT_EDGE,        /* Right ,,  ,,  ,,     ,,    ,,  ,,    */
     TOP_EDGE,          /* Top   ,,  ,,  ,,     ,,    ,,  ,,    */
     BOTTOM_EDGE,       /* Bottom ,, ,,  ,,     ,,    ,,  ,,    */
     IN_IMAGE,          /* In / out of image indicator         */
     STATUS)            /* Enabled/disabled status indicator   */
      FIXED BINARY(31);
CALL ISQBOX(ECHO,LEFT_EDGE,RIGHT_EDGE,TOP_EDGE,BOTTOM_EDGE,
            IN_IMAGE,STATUS);
```

The parameters are as follows:

- The first parameter always returns the value 0.
- The next four parameters are self-explanatory.
- The next parameter, IN_IMAGE, indicates whether all four corners of the box cursor are within the image on the current GDDM page:

  `0` All four corners of the box are outside the image, and none of the image is inside the box.
  `1` All four corners of the box are within the image.
  `2` One or more corners of the box are outside the image, and part or all of the image is inside the box.

Coordinates that are outside the image on the current GDDM page are given appropriate values, extrapolated from the pixel coordinate range of the image on the current GDDM page, that is, of the image field.

- The last parameter, STATUS, returns the value 0 if the box cursor is disabled, or 1 if it is enabled. (You can use the ISQBOX call with the cursor disabled.)

Initializing the image cursors, using calls ISILOC and ISIBOX

There are calls for defining the echo type and initial position of the image cursors. Image cursors can be initialized when disabled or when enabled. Initializing does not change the disabled or enabled state.

You can use the ISILOC call to initialize the image locator cursor. Here is a typical example:

```
CALL ISILOC(0,150,25);
```

The parameters are as follows:

- The first parameter must be set to 0. It specifies that the default locator echo, a small cross, is to be used.
- The next two parameters specify the initial position of the cross cursor, in pixels, horizontally and vertically respectively.

You can use the ISIBOX call to initialize the image box cursor. By default, the image box cursor is of device cell size and is positioned at the center of the image field. Here is a typical call:

```
CALL ISIBOX(0,15,45,200,250);
```

The parameters are as follows:

- The first parameter must be set to 0. It specifies that the default echo, a box, is to be used.
- The next four parameters specify respectively the left, right, top, and bottom edges of the box, in pixel coordinates.

Local operations on the 3193 display station

The local operations that can be performed by the end user on the 3193 are:

- Cursor type selection
- Cursor movement
- Box cursor size or shape change.

Cursor type selection is required if either or both of the image cursors are enabled. In this case, either two or three cursors (the alphanumeric cursor, and one or two image cursors) are displayed, at their initial position.

The **cursor mode key** on the 3193 keyboard switches cyclically between the three cursors if both image cursors are enabled, or alternates between the two if only one image cursor is enabled. If no image cursor is enabled, pressing this key has no effect. (There is no immediate screen feedback of cursor selection, but whichever has been selected responds to use of the cursor-move key.)

Cursor movement is done by the same up, down, left, and right keys as are used for moving the alphanumeric cursor. The currently selected cursor, as determined by use of the cursor mode key, is moved appropriately by these keys.

For the image cursors, one key press moves the cursor by two pixels. Sustained pressure results in accelerating cursor movement. Use of two keys (for example, down and left) at the same time causes the cursor to move diagonally.

Movement off the edge of the screen is prevented.

Box cursor size or shape change is obtained by using the cursor move keys in upper shift. Their operation is effectively on the bottom right corner of the rectangle, while the top left corner remains fixed.

Thus, pressing the cursor downward movement key deepens the rectangle by moving down the bottom edge. If this key is kept pressed, the rectangle bottom edge moves until it reaches the bottom edge of the viewport and then it stops. Pressing the cursor left key reduces the width of the rectangle by moving the right edge to the left. If this key is kept pressed, the rectangle right edge moves until the rectangle becomes just a vertical line, and then it stops. And so on.

Interactive image manipulation example

In the next two examples, the end user uses the box cursor to indicate the boundaries to which a displayed image is subsequently trimmed.

The example in Figure 98 on page 352 restores the image from a saved GDDM image object to the default image field, which implies a full screen image field. The box cursor can therefore never be positioned outside this field.

The example in Figure 99 on page 354 shows how an image field covering part of the screen can be used.

```
IMPROG5: PROC OPTIONS(MAIN);
DCL H_PIXELS FIXED BIN(31);      /* Display image  horizontal  */
                                 /*  size in pixels            */
DCL V_PIXELS FIXED BIN(31);      /* Display image  vertical    */
                                 /*  size in pixels            */
DCL IM_TYPE FIXED BIN(31);       /* Display device image type  */
DCL RES FIXED BIN(31);           /* Defined/undefined resolutn.*/
DCL RES_UNIT FIXED BIN(31)       /* Display device resolution  */
             INIT(0);            /*  units to be inches        */
DCL H_RES FLOAT DEC(6);          /* Display image  horizontal  */
                                 /*  resn. in pixels per inch  */
DCL V_RES FLOAT DEC(6);          /* Display image  vertical    */
                                 /*  resn. in pixels per inch  */
DCL (ATTYPE,ATTVAL,COUNT)        /* ASREAD parameters          */
       FIXED BIN(31);
DCL BOX_ECHO FIXED BIN(31);      /* ISQBOX parameter           */
DCL BOX_LEFT    FIXED BIN(31);   /* Box left edge position     */
                                 /*                 in pixels  */
DCL BOX_RIGHT   FIXED BIN(31);   /* Box right edge position    */
                                 /*                 in pixels  */
DCL BOX_TOP     FIXED BIN(31);   /* Box top edge position      */
                                 /*                 in pixels  */
DCL BOX_BOTTOM  FIXED BIN(31);   /* Box bottom edge position   */
                                 /*                 in pixels  */
DCL BOX_IN_IMAGE FIXED BIN(31);  /* Box within image           */
DCL BOX_STATUS FIXED BIN(31);    /* Box status (enabled or not)*/
DCL DESCR CHAR(30);              /* Imarst parameter           */
CALL FSINIT;
CALL IMARST(0,0,'IMAGNAME',30,DESCR); /* Restore saved image   */ A
                                      /*  to display screen    */
CALL IMAQRY(0,H_PIXELS,V_PIXELS,IM_TYPE,                         B
            RES,RES_UNIT,H_RES,V_RES);
                                 /* Query the display image    */
CALL ISIBOX(0,0.25*H_PIXELS,0.75*H_PIXELS,                       C
              0.25*V_PIXELS,0.75*V_PIXELS);
                                 /* Initialize box cursor      */
CALL FSENAB(3,1);                /* Enable image input         */ D
CALL ISENAB(2,1);                /* Enable box cursor          */ E
```

Figure 98 (Part 1 of 2). Interactive program that enables end users to trim the edges of an image

```
LOOP:
DO WHILE(1=1);                    /* Cursor process loop        */
  CALL ASREAD(ATTYPE,ATTVAL,COUNT);
  IF ATTYPE=1 THEN
    IF ATTVAL=3 THEN LEAVE LOOP; /* Exit if PF3 key pressed    */ F
                ELSE
    IF ATTVAL=12 THEN             /* Restore original image     */
       CALL IMARST(0,0,'IMAGNAME',30,DESCR);/* If PF12 pressed */ G
                 ELSE;            /* Ignore other PF keys       */
              ELSE DO;            /* Trim image to box size     */
  CALL ISQBOX(BOX_ECHO,           /* Query box cursor           */ H
              BOX_LEFT,BOX_RIGHT,
              BOX_TOP,BOX_BOTTOM,
              BOX_IN_IMAGE,BOX_STATUS);
  CALL IMATRM(0,BOX_LEFT,BOX_RIGHT, /* Trim the display image  */ I
                BOX_TOP,BOX_BOTTOM);
                   END;           /* Trim image to box size     */
END LOOP;                         /* Cursor process loop        */
CALL ISENAB(2,0);                 /* Disable box cursor         */ J

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;

END IMPROG5;
```

Figure 98 (Part 2 of 2). Interactive program that enables end users to trim the edges of an image

In the above program, at **A**, a previously saved image with the file name IMAGNAME is restored to the display screen. The image field defaults to full screen size.

At **B**, the size of the display device image (the image field) is queried. This is used, at **C**, to set the box cursor size to half of this size, and to position it centrally.

At **D** and **E**, image input is enabled. You must code both of these statements. The first enables image input as the input type, and the second specifically enables the box cursor.

The loop following these statements enables the terminal user to reposition and change the size of the box cursor, and press the ENTER key, after which the displayed image is trimmed to the box; all of this can be repeated as many times as required.

At **F**, the user can exit from the loop by pressing PF3.

At **G**, the user is able to restore the original, untrimmed image by pressing PF12.

At **H**, the box cursor is queried, and in this simple example the returned values are used directly, at **I**, to trim the displayed image to the box size.

At **J**, the box cursor is disabled before terminating GDDM. In a real application, other functions might precede the GDDM termination, and it is good practice to disable the image cursor once the associated processing is completed.

Figure 99 shows an extension of the above example demonstrating how to handle an image field that occupies only part of the screen. In this case, the box cursor can lie partly or completely outside the image field.

```
IMPROG6: PROC OPTIONS(MAIN);
     .
     .
/* Declarations as in previous example, plus:                    */
DCL ERROR BIT(1);                /* On for error in box position*/
DCL NO  BIT(1) INIT('0'B);
DCL YES BIT(1) INIT('1'B);

CALL FSINIT;

CALL ISFLD(10,15,20,50,0);       /* 20 row by 50 col field     */ A
CALL IMARST(0,0,'IMAGNAME',30,DESCR); /* Restore saved image   */
                                  /*  to display screen    */
CALL IMAQRY(0,H_PIXELS,V_PIXELS,IM_TYPE,
            RES,RES_UNIT,H_RES,V_RES);
                                 /* Query the display image    */
CALL ISIBOX(0,0.25*H_PIXELS,0.75*H_PIXELS,
              0.25*V_PIXELS,0.75*V_PIXELS);
                                 /* Initialize box cursor      */
CALL FSENAB(3,1);                /* Enable image input         */
CALL ISENAB(2,1);                /* Enable box cursor          */
LOOP:
DO WHILE(1=1);                   /* Cursor process loop        */
  CALL ASREAD(ATTYPE,ATTVAL,COUNT);
  IF ATTYPE=1 THEN
    IF ATTVAL=3 THEN LEAVE LOOP; /* Exit if PF3 key pressed    */
                ELSE
    IF ATTVAL=12 THEN            /* Restore original image     */
       CALL IMARST(0,0,'IMAGNAME',30,DESCR);/* If PF12 pressed */
                 ELSE;           /* Ignore other PF keys       */
              ELSE DO;           /* Trim image to box size     */
  CALL ISQBOX(BOX_ECHO,          /* Query box cursor           */
              BOX_LEFT,BOX_RIGHT,
              BOX_TOP,BOX_BOTTOM,
              BOX_IN_IMAGE,BOX_STATUS);
  IF BOX_IN_IMAGE=0 THEN ERROR=YES;/* Box is completely outside*/ B
                                  /*  image                   */
    ELSE
    IF BOX_IN_IMAGE=1 THEN ERROR=NO;/* Box is fully within     */ C
                                   /*  image                  */
```

Figure 99 (Part 1 of 2). Program manipulating an image that is larger than the screen

```
     ELSE   /*BOX_IN_IMAGE=2*/    /* Box is partly outside image*/
      DO;                         /* Sub-box process            */  D
        ERROR=NO;
        IF BOX_LEFT < 0 THEN BOX_LEFT = 0;
        IF BOX_TOP  < 0 THEN BOX_TOP  = 0;
        IF BOX_RIGHT > H_PIXELS-1 THEN BOX_RIGHT = H_PIXELS-1;
        IF BOX_BOTTOM > V_PIXELS-1 THEN BOX_BOTTOM = V_PIXELS-1;
      END;                        /* Sub-box process            */
   IF ¬ERROR THEN                                                  E
    CALL IMATRM(0,BOX_LEFT,BOX_RIGHT,/* Trim the display image  */
                BOX_TOP,BOX_BOTTOM);
                   END;           /* Trim image to box size     */
 END LOOP;                        /* Cursor process loop        */

 CALL ISENAB(2,0);                /* Disable box cursor         */

 CALL FSTERM;

 %INCLUDE ADMUPINA;
 %INCLUDE ADMUPINF;
 %INCLUDE ADMUPINI;

 END IMPROG6;
```

Figure 99 (Part 2 of 2). Program manipulating an image that is larger than the screen

At **A** in Figure 99 on page 354, an image field 20 rows deep by 50 columns wide is created.

At **B**, the error switch is set if the box cursor has been positioned completely outside the image field. In a real program, an alphanumeric prompting message might be provided, telling the end user to reposition the cursor.

At **C**, when the box is fully within the image field, the switch setting ensures that the processing is the same as in the previous example.

At **D**, the box that is partly outside the image field is redefined, to force it to be entirely within the image field boundaries. If this were not done, an IMATRM error condition would occur.

At **E**, the actual image trimming takes place, and is precluded if the box cursor is completely outside the image field.

Transferring images into and out of your program

If you need to transfer image data to or from devices not supported by GDDM, or if you need to convert images to or from other application programming interfaces, then you require some means of transferring image data between your application program and GDDM.

There are two groups of image calls that enable you to do this, subject to specific requirements on the format and compression of the image data.

A “PUT” operation, using the call group IMAPTS, IMAPT, and IMAPTE, permits the entry of image data into GDDM, if it is held in one of several standard formats, or it

consists of unformatted data. In addition, several different compression types are permitted, but only in specific combinations with formats.

Likewise a “GET” operation, using the call group IMAGTS, IMAGT, and IMAGTE, permits the retrieval of image objects from GDDM to your program, again with specific format/compression rules.

The supported combinations of format and compression types are shown by a **Yes** in Table 5. **Yes–direct** signifies that the format/compression combination permits **direct transmission**, discussed under “Direct transmission” on page 364. Note that the values indicated by **Yes–direct** are those returned by the ISQFOR and ISQCOM calls.

Table 5. Acceptable combinations of format and compression

	Unformatted	**3193 data stream**	**CPDS**
Uncompressed	**Yes**	**Yes–direct**	No
MMR	**Yes**	**Yes–direct**	**Yes**
4250	No	No	**Yes**
3800	No	No	**Yes**

For further information see the *GDDM Base Application Programming Reference* book.

The “PUT” and “GET” operations are **transfer operations**, so they can invoke a projection to transform the data as it is transferred.

Starting a PUT operation, using call IMAPTS

Here is an example of IMAPTS, to start transfer of an unformatted, uncompressed image from your application program to the image on the current GDDM page:

```
CALL IMAPTS(0,0,1,1);
```

The parameters are as follows:

- The first parameter specifies the target image identifier. 0 means the image on the current GDDM page. –1 is invalid.
- The second parameter identifies a projection to be applied to formatted data (only). If specifying unformatted and uncompressed data (in the next two parameters), this must be 0, for the identity projection.
- The next parameter defines the format of the source image; 1 means unformatted. Other possible values are:

  `0` Default (same as 2)
  `-1` Unformatted (reversed polarity)
  `2` 3193 data-stream structures
  `-2` 3193 data-stream structures (reversed polarity)
  `3` CPDS structures
  `-3` CPDS structures (reversed polarity).

Normally, for GDDM images, 0 is black and 1 is white. **Reversed polarity** implies that 0 is white and 1 is black.

- The last parameter specifies the compression type of the source image. Possible values are:

  `0` For unformatted data, this is the same as 1. For formatted data, the compression is to be determined by inspection of the data.
  `1` Uncompressed
  `2` MMR
  `3` 4250
  `4` Advanced function printers

Here is another example of IMAPTS, that starts transfer to the image on the current GDDM page, of a CPDS formatted image with 4250 compression. In addition it invokes projection 17:

```
CALL IMAPTS(0,17,3,3);
```

Note that only the formats marked by **Yes–direct** in Table 5 on page 356 give **direct transmission**, if the projection can be offloaded to the device. See "Controlling host offload by specifying image quality" on page 359 and "Direct transmission" on page 364 for more details.

PUTTING data into an image, using call IMAPT

Here is an example to transfer the contents of a 400-byte buffer area named BUFFER:

```
DCL BUFFER CHAR(400);
CALL IMAPT(0,400,BUFFER);
```

The parameters are as follows:

- The first parameter is as usual the image identifier. Again, 0 specifies the image on the current GDDM page.
- The second parameter specifies the data length to be transferred from the buffer named in the third parameter.
- The third parameter names the source image data buffer in your program.

Ending a PUT operation, using call IMAPTE

Here is an example call:

```
CALL IMAPTE(0);
```

where the only parameter is the image identifier.

Here is an example showing how the IMAPTx calls are combined. This time you can assume that the source image is contained in an array of buffers, with a second array specifying the image data length in each buffer.

The code to transfer all of this data to the current GDDM page could be as follows:

```
DCL BUFDATA(100) CHAR(400);/* Application program image buffers */
DCL BUFLEN(100) FIXED BINARY(31);/* Data lengths in each BUFDATA
                                  /*  buffer                    */
DCL (BUFCOUNT,             /* Count of used buffers            */
     FORMAT,               /* Format code                      */
     COMPN)                /* Compression code                 */
           FIXED BINARY(31);
 .....
BUFCOUNT=55;               /* Number of used buffers -     55  */
FORMAT=1;                  /* Unformatted data                 */
COMPN=1;                   /* Uncompressed data                */
CALL IMAPTS(0,0,FORMAT,COMPN);
DO I=1 TO BUFCOUNT;
  CALL IMAPT(0,BUFLEN(I),BUFDATA(I));
END;
CALL IMAPTE(0);
 .....
```

Starting a GET operation, using call IMAGTS

Here is an example of IMAGTS, to start transfer of a formatted, compressed image from a scanner device image to your application program:

```
CALL IMAGTS(-1,105,0,2);
```

The parameters are as follows:

- The first parameter is an image identifier. As usual, –1 specifies the display-attached scanner. 0 would specify the image on the current GDDM page.
- The second parameter specifies projection identifier 105.

  The “GET” function is always a transfer operation, so a projection identifier other than 0 can be used, if the associated projection has been created or accessed by your program.
- The third parameter, 0, specifies the format as the default format, the same as if 2 had been coded, meaning that 3193 data-stream structures are used. The permitted values and their meanings are the same as for the format parameter of the IMAPTS call.
- The last parameter, 2, specifies MMR compression. 0 would specify the default, the same as 1, which is uncompressed data. The values 3 and 4 are also permitted, with the same meanings as for the compression parameter of the IMAPTS call.

Note that only the formats marked by **Yes–direct** in Table 5 on page 356 give **direct transmission**, if the projection can be offloaded to the device. See “Controlling host offload by specifying image quality” on page 359 and “Direct transmission” on page 364 for more details.

GETTING data from an image, using call IMAGT

Here is an example of this call:

```
DCL BUFFER CHAR(800);
DCL (BUFLEN,            /* Data area length                    */
     DATALEN)           /* Data actual length                  */
           FIXED BINARY(31);
BUFLEN=800;
CALL IMAGT(-1,BUFLEN,BUFFER,DATALEN);
```

The parameters are as follows:

- The first parameter is the image identifier, –1 for the scanner.
- The next parameter is the available buffer length.
- The third parameter is the name of the data area to receive the image data.
- The last parameter is the length of image data placed in the data area (buffer) by GDDM. If it is 0, all the image data has been returned.

Ending a GET operation, using call IMAGTE

Here is an example call:

```
CALL IMAGTE(-1);
```

The single parameter specifies the image identifier.

Here is an example of the three IMAGTx calls used together to retrieve several buffers of image data:

```
DCL DATABUF(100) CHAR(800);/* Array of data buffers to receive  */
                           /*  image data                       */
DCL DATALEN(100) FIXED BINARY(31);/* Array of data length values*/
                                  /* to be returned             */
DCL BUFLEN FIXED BINARY(31);
BUFLEN=800;
CALL IMAGTS(-1,105,0,2);   /* Start data retrieval, parameters  */
                           /*  as before                        */
DO I=1 BY 1 UNTIL(DATALEN(I)=0);
                           /* Continue till no more data        */
  CALL IMAGT(-1,BUFLEN,DATABUF(I),DATALEN(I));
                           /* Retrieve scanner image data       */
END;
CALL IMAGTE(-1);           /* End data retrieval from scanner   */
```

For unformatted or 3193 data-stream format, all buffers, except possibly the last, are filled. For CPDS format, all buffers are partly filled.

Controlling host offload by specifying image quality

You have already met projections and the transform calls that they can contain. Using these calls you can define a projection to do such tasks as:

- Extracting one or more rectangular sub-image(s) from the source image
- Applying a scaling factor to the extracted image
- Choosing the scaling/resolution conversion algorithm

- Placing one or more extracted images within the target image.

As mentioned earlier, defining a projection does not specify the source or target image on which it is to act, **nor where the operations are to be performed**.

For example, in a transfer operation that has a 3193 device image as its target, some or all of the projection transforms can be performed in the device itself, if the transforms are within the capability of the 3193. The processing by the device offloads processing from the host, and is known as host offload.

The first requirement for host offload of image transforms is offload of the target image itself. If GDDM determines that the image can be kept by the device, GDDM does not keep a copy. The conditions for this are:

1. The image field is write-only.
2. User control has not been made available (the default).
3. Real partitions are specified, if partitions are required.

Because image data cannot be retrieved from the 3193, the specification of a read-write image field forces GDDM to keep a copy of the target image, and to perform all transforms on it within GDDM. The result is then available within GDDM, and can be used as the source of any subsequent transfer operation. See “Defining an image field, using call ISFLD” on page 365.

Generally, GDDM in the host has more precise image processing ability than image devices. However, GDDM processing by the host carries a performance penalty (increased response times, processing, and storage), that you can avoid by choosing to accept the lower quality function offered by the devices. This can be particularly useful in a system environment of multiple concurrent users.

The two calls, ISCTL and ISXCTL, described below, give you some control over the trade-off between quality of function and performance, by controlling whether particular transform calls are performed in the host or in the image device. Accepting lower quality allows GDDM to approximate the precise requirements of your program to those supported by the device, depending on the factors stated below under each function subheading.

Here are the descriptions of the variable operation conditions.

Image size rounding

The 3117 and 3118 scanners can scan only an area of the paper that is a multiple of 8 pixels wide. Also, the left edge of the scanned area must be a multiple of 8 pixels from the left edge of the scanner detector. If you do not mind about image-size rounding, GDDM may round the scanner image size, or extracted image size, to suit the scanner limitations. If, on the other hand, you do not want your image sizes to be rounded, GDDM processes the scanned images to ensure that the effects of the rounding are not noticeable by the application.

Scaling and resolution conversion

The 3193 device supports scaling factors of 1/4, 1/3, 1/2, 2/3, 3/4, 1, 4/3, 3/2, 2, 3, and 4 only. If you specify a scaling factor that is not supported by the 3193, 0.1 or 1.25 for example, and specify that the factor must be applied precisely, GDDM, not the 3193, must do the scaling. Or, it could be acceptable for the scale factor to be within the range 0.9 times the specified value through 1.11 times the specified value. The values, 0.9 and 1.11, define a range for the **scale factor multiplier**. If you insist on precise scaling, this can be stated as needing a scale factor multiplier value of 1.00.

Scaling algorithm (also used in resolution conversion)

You may not mind whether the target image device supports a specific scaling algorithm called for in your projection, or uses another. Instead, you may require rigid adherence to the algorithm specified, even if GDDM has to perform it.

The 3193 supports pixel replication. GDDM can perform the black pixel retention or white pixel retention algorithm; see the description of the IMRRAL call under “Defining the resolution conversion algorithm, using call IMRRAL” on page 102.

Multiple extraction and placing of rectangles

The 3193 can handle four or fewer transforms per projection. For a projection, any more than four transforms involve extra overhead for GDDM in the host. If you do not mind this extra overhead, you can specify this to GDDM. Or, if you want to avoid the overhead, you can ask GDDM to limit the number of extractions to four.

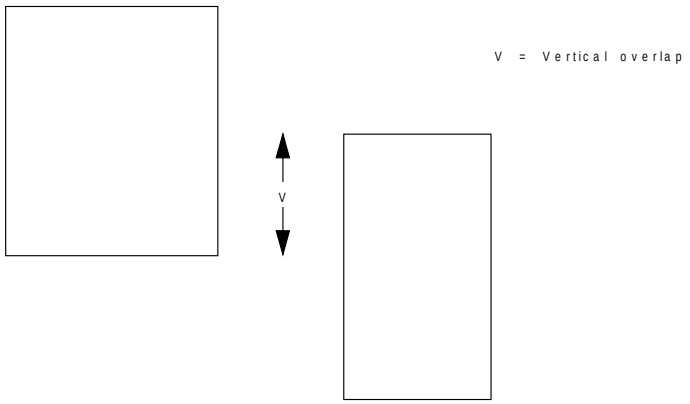


Figure 100. Vertical overlap

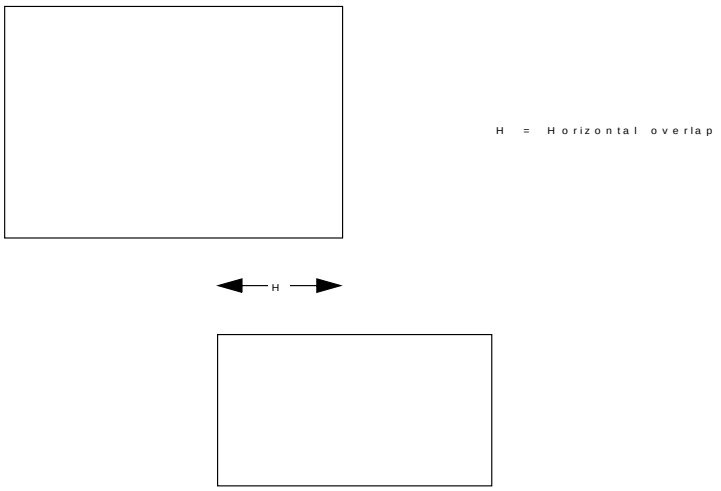


Figure 101. Horizontal overlap

Even within this limit, it may give incorrect results in any overlapped areas. This depends on the image-mixing modes defined in the IMRPL or IMRPLR calls described in Chapter 6, “Image basics” on page 85. If you do not mind whether one or more transforms are in error where they overlap in the target image, you can specify this to GDDM. Or, you can request GDDM to avoid incorrect overlap, which it does either by performing the transform in the host, or by sending the transform separately to the device, which means GDDM sends the image more than once.

Note: Unexpected overlaps can occur because of scale factor modification already described under “Scaling and resolution conversion” on page 361.

Controlling image quality, using call ISCTL or ISXCTL

You can use call ISCTL or ISXCTL in your application to control the above four variable operations by specifying the image quality that is acceptable for the current page or scanner device.

Here is an example of the ISCTL call required to specify that all extracts are to be processed, that the scale factor multiplier is to be constrained to the range 0.9 through 1.11, the specified scaling algorithm is to be honored, incorrect results in placing overlapped rectangles are to be avoided, and image size rounding is to be avoided:

```
CALL ISCTL(0,4);
```

The parameters are as follows:

- The first parameter is a device image identifier:

  `0` The current page
  `-1` The scanner

- The second parameter is a value n in the range 0 through 5, specifying the required quality. 0 is the default and is the same as 3. 1 through 5 have the following meaning:

  (1=low quality, 5=high quality)

n	Process all extracts	Scale factor multiplier	Honor scaling algorithm	Avoid overlapped rectangles	Avoid image size rounding
1	Don’t care	Any - any	Don’t care	Don’t care	Don’t care
2	Don’t care	0.4 - 2.5	Don’t care	Don’t care	Don’t care
3	Yes	0.8 - 1.25	Don’t care	Don’t care	Don’t care
4	Yes	0.9 - 1.11	Yes	Yes	Yes
5	Yes	1.0 - 1.0	Yes	Yes	Yes

Or, you can use the ISXCTL call for more selective control of the function/performance trade-off. Here is an example specifying that on the current page all extracted rectangles are to be processed, you do not mind whether the specified scaling algorithm is used, and the overlapped rectangle treatment is to be unchanged from its previous setting. Further, the scaling/resolution conversion limits to be applied are as follows. The lower scaling limit is to be the exact value, and the upper scaling limit is to be 1.3. This means that any specified scale factor can be modified by a multiplier within the range 1.0 through 1.3.

Here is the call and its declaration and assignment statements:

```
DCL ARRAY1(3) FIXED BINARY(31);
ARRAY1(1)=1;            /* Process all extractions              */
ARRAY1(2)=0;            /* Scaling algorithm may be varied      */
ARRAY1(3)=-1;           /* Leave unchanged rectangle
                           specification                        */
DCL ARRAY2(2) FLOAT DECIMAL(6);
ARRAY2(1)=1.0;          /* Lower scaling limit exact (1.0)      */
ARRAY2(2)=1.3;          /* Upper scaling limit 1.3              */
CALL ISXCTL(0,3,ARRAY1,2,ARRAY2);
```

The parameters are as follows:

- The first parameter, 0, is a device image identifier specifying the current page; –1 would specify the scanner.
- The second parameter, 3, is a count of the number of elements specified in the array parameter.
- The third parameter is an array of up to four elements that specify respectively whether GDDM is to process all extractions, honor the scaling algorithm specified, avoid overlapping rectangles, and avoid image-size rounding.

  The setting for any one of these four array elements can be one of the following values:

  `-1` Unchanged. This is the default if the element is not included (see note below).
  `0` Don’t care.
  `1` Yes.

- The fourth parameter is a count of the number of elements in the further array parameter.
- The final parameter is an array of up to two elements specifying respectively the lower and upper scaling limits. Each of these elements can have one of the following values:

-1 Unchanged. This is the default if the element is not included (see note below).
1 Exact.

Alternatively, the lower scaling limit can have a value in the range 0 through 1.0, and the upper scaling limit can have any value greater than 1.0.

Note: "Unchanged" means unchanged from a previous setting, if any, by ISCTL or ISXCTL, or if this is not done, the ISCTL default parameter settings apply.

Direct transmission

Because the 3193 supports host offload of transforms, image data passed to GDDM by IMAPT calls may be sent directly to the device, so GDDM does not have to accumulate the entire image. This is known as **direct transmission**.

Direct transmission has the following advantages for an application:

- It minimizes storage usage, thereby improving system performance.
- It improves usability by showing the first part of an image sooner.
- It enables the operator to start making decisions earlier, thereby improving throughput.

Direct transmission to the 3193 is used by default, if the data is in 3193 data-stream format, if the 3193 can perform the **entire** projection, and if GDDM does not otherwise need to perform the projection itself (for example, to maintain a read-write image field). GDDM sends the data directly to the 3193 without keeping a copy of the entire image.

If the 3193 cannot perform the entire projection, GDDM performs the functions in the host where necessary. GDDM may need to construct a copy of the entire image from the buffer contents to do this. This also happens for devices other than the 3193.

As described in "Controlling host offload by specifying image quality" on page 359, the application can specify, by the ISFLD call, whether the image field on the current GDDM page is to be read-write or write only. If it is specified as read-write but the device has write-only function, GDDM buffers the entire image, and direct transmission is not used.

Direct transmission from a scanner

When using the IMAGTx calls, direct transmission from a scanner can take place, if all the following conditions are met:

1. The current ISCTL values for the scanner must specify that you don't care about avoiding image size rounding.
2. The projection must contain only one transform.
3. The transform must not contain IMRSCL, IMRREF, or IMRORN.
4. The scanner can only supply image data in the negated format (that is, where 1 = black) so the IMAGTS call must specify a format of +2 if the transform contains a negate element, or –2 if it does not.
5. Compression must be either uncompressed or MMR.

6. When echoing is required, it must be possible for the device to perform the echoing. See “Direct echoing when scanning” on page 365.

If the above restrictions are not met, GDDM scans the data into a temporary image, and performs the projection as part of the IMAGTS processing. The subsequent IMAGT calls use data from the temporary image.

Direct echoing when scanning

Usually, echoing can be performed by the 3193 to which the scanner is attached, saving host processing, if the following conditions are met:

1. Offload of the target image (see “Controlling host offload by specifying image quality” on page 359).
2. The projection can be performed by the 3193, within the quality requirements specified by the ISCTL or ISXCTL call.

Combining an image with text or graphics

Chapter 7, “Hierarchy of GDDM concepts” on page 107 introduced the concept of an image field similar to a graphics field. In addition, several sections in this section, Chapter 6, “Image basics” on page 85 described the use of image identifier 0 to refer to the image field on the current GDDM page, assuming such a field exists.

Only one image field can exist per page, and as for a graphics field, it can be created explicitly or by default. Usually you let GDDM create the image field for you. If, however, you want the image field to extend over only part of the page, you must create one explicitly. The most likely reason for doing this is to share the page between image and alphanumerics or graphics.

Like alphanumeric and graphics fields, an image field is defined in page row and column coordinates.

The image field and alphanumeric field(s) can overlap, just as graphics and alphanumerics can overlap. However, image and graphics fields can coexist on the same page only if they do **not** overlap.

Where a device does not accept image data streams, GDDM supports image processing by internally using graphics calls (emulation), and this can be done only if there are no graphics on the same page. If there is a graphics field on the page, GDDM displays or prints its contents in preference to those of the image field.

You can, however, display a graphics field and an image field at the same time, on family-1 display devices other than the 3193, by placing the fields in separate partitions.

Defining an image field, using call ISFLD

Here is an example of the ISFLD call used to create an image field that begins on row 5, at column 10, and is 15 rows deep and 50 columns wide:

```
CALL ISFLD(5,10,15,50,0);
```

The parameters are just as for the GSFLD call, except for an additional parameter at the end:

- The first two parameters specify respectively the row and column of the top left corner of the image field.
- The next two parameters specify respectively the depth and width of the image field, in row and column units.
- The last parameter specifies a **control** value.

  0 specifies the default control action, the same as value **1**, that means the image is to be a write-only image. A value of **2** specifies read-write.

As with GSFLD, if any of the first four parameters is given the value zero, the field is deleted.

As an example of the need to use read-write for a display image, consider an application that displays an image on a 3193. A terminal user may want to select parts of this image, using the image box cursor, compose an image using those parts, and save away the result.

To do this, GDDM must be told to buffer the entire image by initially defining the image field as read-write. This impairs performance, because GDDM keeps a copy of the image.

Querying the attributes of an image field, using call ISQFLD

This is an example of the use of this query call:

```
DCL (ROW,              /* Starting row                        */
     COL,              /* Starting column                     */
     DEPTH,            /* Depth in rows                       */
     WIDTH,            /* Width in rows                       */
     CONTROL)          /* Control parameter                   */
      FIXED BINARY(31);
CALL ISQFLD(ROW,COL,DEPTH,WIDTH,CONTROL);
```

Apart from being returned by GDDM rather than set by the caller, the parameters are the same as for ISFLD, with the exception that a control parameter returned value of 0 means no image field exists.

Printing images

The Image Print Utility is a program which you can use to convert GDDM image (ADMIMG) files into files which can be printed on a page printer. When you invoke the program, you need to specify a number of parameters that allow you adjust the scale and rotation of the printed picture. For details of these parameters and their effects, see the *GDDM System Customization and Administration* book.

Printers that accept intelligent printer data stream, such as the IBM 4028 are best suited to printing image.

Printing an image on an IPDS printer

On nonIPDS printers, image output is done by emulation with some associated performance overheads. Because GDDM uses graphics (GSIMG) for this emulation, there must be no graphics on the same page as the image data.

The plotting of images is supported, but not recommended. Image transforms and output are done by emulation, as above. Each pixel is drawn as a very short

vector and resolution is determined by the pen width. Images other than small images take a long time to be plotted, and subject the pens to greater than usual wear.

4028 as the primary output device

So far, you have implicitly selected the display screen as the output device. GDDM has opened it for you, automatically.

Here is an example that uses DSOPEN to establish the 4028 as the primary device for printing image. The device calls, DSOPEN (open) and DSCLS (close) are introduced in Chapter 18, "Device support in application programs" on page 371.

```
IMPROG8: PROC OPTIONS(MAIN);

DCL PLIST(2) FIXED BIN(31);      /* DSOPEN PROCOPT list        */
DCL NLIST(1) CHAR(8);            /* DSOPEN name list           */
DCL DESCR CHAR(30);              /* For file description       */

CALL FSINIT;
NLIST(1)='cuu';                                                  A
PLIST(1)=42;                     /* PROCOPT for image inversion*/ B
PLIST(2)=2;                      /* White on screen print black*/

CALL DSOPEN(4028,1,'X4028A4',0,PLIST,1,NLIST);                   C
CALL DSUSE(1,4028);                        /* As primary device  */
CALL IMARST(0,0,'IMAGNAME',30,DESCR); /* Restore filed image   */
                                      /* to the GDDM page      */
CALL FSFRCE;                     /* Output the current page    */
CALL DSCLS(4028,0);              /* Close the printer          */
CALL FSTERM;

%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;

END IMPROG8;
```

At **A**, in the above program, *cuu* is an example of a CMS device address. You need to change this in accordance with your subsystem and installation.

At **B**, processing option 42 is specified for the DSOPEN call for the primary device. The value 2 supplied as the parameter of this procopt causes those parts of the image that appear white on the screen to appear black on the printer.

At **C**, the value 4028 is used just by choice–any unused device identifier can be used. The values 0 and 1 should be avoided. The device token 'X4028A4' is an example and needs to be changed to suit your installation.

Another topic introduced in Chapter 18, "Device support in application programs" on page 371 is the use of **nicknames**. By using nickname statements in a file external to your program, end users or their systems-support personnel can change the primary device used **without changing your program**. Thus you can avoid using the DSOPEN and DSCLS calls. The use of nickname statements and their syntax is all dealt with under "Coding a partial device definition for end users to change with nicknames" on page 374.

Printing an image on a printer as the alternate device

Chapter 20, “Sending output from an application to a printer” on page 399 explains how you can specify that a device such as a printer or plotter can be used as a secondary (alternate) device, while your program still uses the display screen for primary output. You can then issue the DSCOPY call:

```
DCL OPTIONS(3) FIXED BIN(31)   INIT( 1, 0, 1);

/*          WIDTH   DEPTH  HORZ_OFFSET VERT_OFFSET   COUNT  OPT_ARRAY */
CALL DSCOPY( 90,      90,      5,         10,          3,    OPTIONS   );
```

to copy the displayed output to the printer.

Improving the performance of image programs

Image processing involves the capture, storage, transmission, manipulation, and display of very large amounts of data. Anything that you can do to reduce the sheer volume of data will improve the performance of image applications. This section explains how to do this, and also tells you how to take advantage of the processing power available on image devices.

Image processing on image devices

You can reduce the amount of image data to be processed by excluding any information that is not required. For example, if your application involves the processing of several standard forms that have a certain amount of common information, there is little point in capturing, transmitting, or storing that common information. It makes more sense to keep just the parts of the form that differ. You can do this by defining sub-images to be extracted, using:

- IMREX or IMREXR calls in a transfer operation
- IMATRM call to trim an image, without transferring it

You can also use the IMRSCL call to reduce the size of an image, if appropriate.

If an image is only intended for display at a terminal, or on a low-resolution printer, there is no point in creating, keeping or sending it at a high resolution. The resolution of an image is specified when it is created, using call IMACRT, or changed using IMARES.

The application has control over compression only on IMAPTx calls. Compressing image data shortens image data streams and results in faster transmission between the host and the device. This could be a benefit for remotely-connected devices. However, you should also consider the time taken for the host and the device to compress or decompress the data. The MMR compression algorithm is not suited to images containing photographs or half-tone pictures, where the pixels alternate frequently between the on and off states. In many cases, MMR compression can actually expand the data.

The 3193 can perform transfer operations in the device itself. That is, it can apply a projection to image data. This is called **host offload**, and has the benefit of improving performance for the end-user, and using less system storage.

The factors affecting host offload are:

- The capabilities of the device

- The level of quality specified as acceptable in your program
- Whether the data is to be used after transmission.

If you are sending image data from your program to a 3193, using IMAPTx calls, any associated projection may be performed in the device, instead of by GDDM in the host, as long as the 3193 can cope with the projection, and depending on the level of quality that you specify as acceptable, using the ISCTL call. ISCTL is fully described in “Controlling host offload by specifying image quality” on page 359.

Not all transform elements are within the 3193’s capabilities. For example, the 3193 is capable of scaling by factors of 0.25, 0.33, 0.5, 0.66, 0.75, 1.0, 1.33, 1.5, 2.0, 3.0, and 4.0. If an application requires an image to be scaled by 2.4 on output, then GDDM has to do it in the host, unless you specify that a certain level of approximation is acceptable. You can do this with the quality parameter in the call ISCTL. If you specify a low value in this parameter, for example, GDDM will allow the device to approximate 2.4 to the device scaling of 2.0. (a value of 2.5 would be approximated to 3.0). If you specify a high value, you are saying that the device’s approximation would be unacceptable to you, and the scaling of 2.4 would be performed in the host.

You can use the ISFLD call to control whether an image is to be write-only or read-write. If you do not require the image data to be read subsequently, you should specify “write-only.” GDDM can then offload processing to devices, such as the 3193, which support image processing in write-only mode. If you do require the image data to be read subsequently, you should specify read-write. Host offload will then not occur, and GDDM will emulate the processing in the host.

Host offload is a prerequisite for direct transmission. Direct transmission only occurs with IMAPTx transfer operations that have the 3193 as their target.

Projections associated with transfers from image –1 to image 0 can be carried out in the device. This saves processor utilization and usually means that GDDM doesn’t need to keep a copy of the transformed image, so saving virtual storage as well.

Image processing on graphics devices

Image processing is emulated by GDDM on graphics devices such as, the 3472-G. So an image application will use more host and communications resource when running on a graphics device than it will on an image device.

The 4224 printer supports image data without emulation.

Device variations for image

This section deals with the use of devices other than those principal image processing devices covered in the preceding sections of this section and Chapter 6, “Image basics” on page 85.

Displays that support graphics

This information applies to all display devices that can be opened for family-1 output except the IBM 3193 image display. (This includes the IBM 3179-G, 3192-G, 3472-G, 3270-PC including /G and /GX, 3279, 3290 5080 and 6090 displays and the 5550 Multistation but does not include the IBM 3193 display.)

Image transforms and output are done entirely by emulation, with some associated performance overheads. Because GDDM uses graphics (GSIMG) for this emulation, there must be no graphics on the same page as the image output. End users of image-processing application programs can manipulate images on the screen using the User Control facility. This facility enables users to view parts of the image in detail and to print what they see on the display.

The image locator cursor echo, normally a cross symbol, is the same as the alphanumeric cursor. On 3279 and 3290 displays (as on 3277 and 3278), cursor positioning is only to the nearest cell, not to the nearest pixel.

The image box cursor is not supported, and an error message is issued if any attempt is made to enable it.

Image input to GDDM

3117 and 3118 scanners can be attached to an IBM 3193 Display Stations, to a PC, or to a PS/2. Image data input for display or printing must therefore be done using either an ADMIMG file, restored from auxiliary storage by use of the IMARST call, or an appropriately formatted image transferred from your application to GDDM by use of the IMAPTS/IMAPT/IMAPTE calls (see “Transferring images into and out of your program” on page 355 for admissible formats). The image can then be transferred to the GDDM page in the usual way.

Chapter 18. Device support in application programs

Many programs can be written without much knowledge of the way GDDM supports different devices but you need to understand it to perform the following tasks:

- Defining a device's characteristics to GDDM
- Sending output to a device other than the end user's terminal
- Copying output from the main display to a printer or plotter
- Communicating with more than one device
- Saving a data stream suitable for subsequent output on a different device
- Specifying device-dependent or subsystem-dependent processing options

As explained in Chapter 7, "Hierarchy of GDDM concepts" on page 107, the device is at the top of the hierarchy. Any graphics, alphanumeric, or image objects that your program puts on the GDDM page belong to the device that is current when the page is created. This means that before your program creates any objects, it must tell GDDM to which device they belong and must be sent as output. You can do this by issuing a pair of calls in your program; first a DSOPEN call and then a DSUSE.

With the DSOPEN call, you define for GDDM a conceptual device to which your application can later send its output.

With the DSUSE call, you can supply a device identifier from an earlier DSOPEN call, to use that device as the current primary or alternate device. All subsequent alphanumerics, graphics, and image calls apply to that device until a new device is made current. Your program's output is created for the current device with respect to its definition on the DSOPEN call.

There is generally no need for explicit device control when the output is to appear on the invoking terminal. The current device defaults to the invoking device – called the **user console**. For the default device, GDDM issues an internal DSOPEN.

If you want your application to use a device other than the invoking terminal, you must explicitly open that device using a DSOPEN call. If you want to use the invoking terminal, but in a nonstandard way, you can issue an explicit DSOPEN call associated with the physical address of the user console or modify the internal DSOPEN with a nickname statement (see "Coding a partial device definition for end users to change with nicknames" on page 374).

Using DSOPEN to tell GDDM about a device you intend to use

The DSOPEN call is one of the most powerful calls in the GDDM API and understanding it is crucial, if you write programs for many end users on different types of hardware.

You've got a choice as to how you use DSOPEN.

- You can give a complete definition of the device and its characteristics by specifying each parameter of the DSOPEN call explicitly.

or

© Copyright IBM Corp. 1982, 1996

- You can give a partial definition of the device on the DSOPEN call and allow the nicknames, created by end users of the program or by their systems-support personnel, to customize the definition to their own needs and hardware.

If you choose the latter option, not only do you simplify the coding of the DSOPEN call for yourself, you also simplify the work of those creating matching nicknames and make your program less specific to any particular type of device. See "Avoiding dependencies when opening and using devices" on page 392 for advice on writing device-independent programs.

Coding a complete device definition on the DSOPEN call

This is a typical call where nicknames are not used:

```
DCL PROCOPT_LIST(10) FIXED BIN(31); /* Processing options list  */
DCL NAME_LIST(1) CHAR(8);           /* Device-control name list */

PROCOPT_LIST(1)=28;             /* Option code 28 specifies     */
                                /* availability of User Control */
PROCOPT_LIST(2)=1;              /* Option value 1 makes         */
                                /* User Control available       */

NAME_LIST(1)='062';             /* CMS device address           */

/******************/
/* OPEN DEVICE 11 */
/******************/
/*   DEVICE-ID  FAMILY  TOKEN  PROCESSING-OPTIONS  PHYSICAL-DEVICE*/
CALL DSOPEN(11,    1, 'L3472GM',  2,PROCOPT_LIST,     1,NAME_LIST);
```

This is the meaning of the seven parameters:

`11` The **device identifier**. A number that you assign to the device, which you use for all subsequent references to that device in your program.

`1` The **output-family code**, specifying the type of output.

The value 1 sends output to a device (display or printer) attached directly to the end user's virtual machine. Other permitted settings are:

- `2` Queued printing. Output to a print file, which can be printed using the GDDM Print Utility Program.
- `3` System printing. Output to a print file, which is passed to the subsystem's spooling program.
- `4` Advanced-function printing. Output to a print file, which is passed to a utility program for printing on advanced-function printers.

`L3472GM` The **device token**, telling GDDM the properties of the device. The token L3472GM indicates that the device is a local 3472-G with a mouse attached. There are four sets of device tokens supplied with GDDM, called **device definition tables**. They are listed in the *GDDM Base Application Programming Reference* book.

A token parameter of * tells GDDM to discover the device's properties itself; usually by querying the device. This setting is recommended whenever possible. If you code an explicit device token, your program is dependent on that type of device.

`2` The number of fullwords in the processing options list that is passed in the next (the fifth) parameter.

`PROCOPT_LIST` The name of an array containing the **processing options list**. This list may contain one or more **option groups**, each requesting a particular processing option. Some of these options depend on the output family, others are valid only on a particular subsystem.

This example contains just one option group, a User Control option group. The first fullword in a group identifies the option type. Here 28 indicates “User-Control group.” The remaining fullwords give the setting of the option. For this option type there is just one fullword following. It is set to 1 to request that User Control be enabled for the newly-opened device.

You can place several option groups in the processing options list, each with an option code in its first word. A list of the possible option groups is provided in the *GDDM Base Application Programming Reference* book.

`1` The number of 8-byte names in the seventh and last parameter.

`NAME_LIST` An array of 8-byte names, identifying the physical device to be opened. The naming scheme used in the **name list** depends on the output family and the subsystem being used.

In most cases the name list can have only one element in it. The exceptions are:

1. Family-4 output under TSO.
2. Family-2, -3, and -4 output under CMS, in which case second and third elements can be used to specify a filetype and a filemode.
3. Auxiliary devices (usually plotters) for family-1 output under any subsystem. These have two-part names; the first part is the name of the family-1 terminal to which they are attached, the second part is the name of the auxiliary device itself.

The naming conventions for each subsystem and output family are described in the *GDDM Base Application Programming Reference* book.

In this example (on CMS), the single name in the name list has been set to “062.” This name is known to the subsystem. It is the virtual address of the device in question. On IMS the single name may be set to an “LTERM name.” On all the other subsystems (CICS, and TSO), you cannot open any display device other than the user’s console. This restriction does not apply to printers.

On all subsystems the device name may be allowed to default to the user console. There are two ways of specifying this action. You may omit the name list (by giving a length of 0), or you may set the name to *. A further possibility is to request a dummy device. (See “Opening and using a dummy device” on page 387.)

In the example, the call to DSOPEN makes known to GDDM a device with a subsystem name of 062 to be used for family-1 output. The DSOPEN call tells GDDM that the device is a local 3472-G graphics terminal with a mouse attached, and it assigns an identifier of 11 to the device for future reference in the program. It requests that the device be opened in a mode that accepts no input from the end user and processes only the program's output.

More examples using the DSOPEN call can be found in Chapter 20, “Sending output from an application to a printer” on page 399 and Chapter 21, “Sending output from an application to a plotter” on page 433. For a detailed description of the parameters of DSOPEN, refer to the *GDDM Base Application Programming Reference* book.

Coding a partial device definition for end users to change with nicknames

GDDM enables users to store device-definition information in special statements called **nicknames**.

If you code a nickname in the name-list parameter of a DSOPEN call, instead of specifying the name of a physical device, the nickname identifies a device definition supplied elsewhere. The “elsewhere” is typically either the GDDM external defaults module or an end user's defaults file (although you can also supply nicknames within the program on the ESSUDS or ESSEUDS calls).

When they customize GDDM after installation, the system-support personnel can include in the external defaults module system-wide nickname statements that contain standard device definitions for use by all users. In addition, under CMS and TSO, end users can supply nicknames in their own private defaults files and systems-support personnel can provide such files for general access by groups of users.

With nickname statements, end users can supplement and change the device definitions on a DSOPEN call **without having to change and recompile the program**.

It is often sufficient for programmers to code a simple DSOPEN call in their programs specifying a device-id, the output family, and a nickname, with default or null values for the other parameters. This leaves it up to the end users of the application or their system-support personnel to decide what actual device to use for that device name and family of output, and what processing options to use with it.

```
DCL PROCOPT_LIST(1) FIXED BIN(31);
DCL NAME_LIST(1) CHAR(8);
NAME_LIST(1) = 'OURPRT';

    /* DEVICE-ID  FAMILY  TOKEN  PROCESSING-OPTIONS  DEVICE-NAME*/
CALL DSOPEN(4,      2,     '*',   0,PROCOPT_LIST,    1,NAME_LIST);
```

When this call is executed, GDDM searches the nickname files for any statements that refer to the same output family and device name; in this case a device opened for family-2 output with the name OURPRT. The files would typically supply a device token and a set of processing options. If no match is found, the DSOPEN call applies unchanged and the output is sent to a print file called “OURPRINT

ADMPRINT”. Assume, however, that GDDM finds this nickname statement in the end user's defaults file:

```
ADMMNICK FAM=2,NAME=OURPRT,
         TOFAM=,TONAME=,
         DEVTOK=X4028A4,
         PROCOPT=((IPDSBIN,0,2)),
         DESC="LOCAL 4028 PRINTER"
```

The FAM and NAME parameters specify the output family and device name to which the statement applies. This nickname statement adds a device token and a processing option to the parameters supplied in the above DSOPEN call, as follows:

- The device token, supplied in the DEVTOK parameter, is X4028A4. This specifies that a print file be created for a 4028 printer loaded with A4 size paper.
- The processing option, supplied in the PROCOPT parameter, is IPDSBIN; it is equivalent to option group 40 in DSOPEN, and it specifies that the printer should print the main document on paper from the default bin and print the header page on paper from bin number 2.

  A list of all the nickname processing options and their DSOPEN option group equivalents is given in the *GDDM Base Application Programming Reference* book.

How GDDM compounds device-definition information for a conceptual device

When a DSOPEN call is issued in an application program, GDDM gathers together all the matching nickname information and compounds it into one single device definition.

In this process GDDM often encounters device-definition information that is contradictory; for example,

The nickname statement for the device in the external defaults module can have a different device token to the one on the DSOPEN call

or

The nickname statement in the user's defaults file can specify a processing option that makes the device operate in output-only mode while the processing option on an ESSUDS call in the program specifies that the device should operate in input/output mode.

GDDM follows these specific rules to resolve such contradictions:

- GDDM scans device-definition information in increasing order of priority.

  This means that, to find device-definition information that matches the DSOPEN call, GDDM scans the locations that can contain such information in this order:

  1. The external-defaults module
  2. The user's defaults file
  3. The SPIB control block passed on the initialization call SPINIT (if the system programmer interface is used)
  4. Any ESSUDS or ESEUDS calls in the application

5. The DSOPEN call itself.

- As GDDM finds processing options, it compounds them all together, if they do not contradict each other. If they do, the processing option in the higher-priority location is used.
- Any processing options that do not apply to the physical device are ignored.
- As GDDM finds device tokens, the token in the highest priority location is used.

  The processing options and device token in a nickname statement are, in effect, default values. Any explicit value on the DSOPEN call overrides them.
- Where values are specified in the TOFAM and TONAME parameters of a nickname statement, they override the values specified on the FAM and NAME parameters of the DSOPEN call.

  This aspect of nicknames is used by end users to redirect an application's output to a different device.

Offering end users a menu of devices available for output

Using the ESQUNLS and ESQUNS calls, you can query, for each output family, the nickname information that is in effect for an end user of your program, and store it in a buffer. Your application can then select device definitions that are suitable for the programs's output and present them to the end user in a menu. End users can then select the most convenient device, (without having to change their defaults files) and your application can use the chosen device name in a DSOPEN call.

Suppose you want to enable end users to choose, from a menu, a queued printer device on which to print the output of your application. This is example illustrates some of the calls you would use to do this.

```
DEVDISP: PROC OPTIONS(MAIN);
/*******************************************************************/
/* QUERY THE GDDM NICKNAMES, AND THEIR DESCRIPTIONS                */
/* AND DISPLAY THEM ON THE END USER'S SCREEN                       */
/*******************************************************************/
DCL BUFPTR PTR;                    /* START OF BUFFER- POINTER    */
DCL P      PTR;                    /* BUFFER OFFSET  - POINTER    */
DCL PA      FIXED BIN(31);         /*                - ARITHMETIC */
DCL BUFENDA FIXED BIN(31);         /* BUFFER END     - ARITHMETIC */
DCL A,B,C,I FIXED BIN(31);         /* WORKING VARIABLES           */

DCL (FAMILY,LEN,RETLEN) FIXED BIN(31);
DCL (FAM,NC,DC) FIXED BIN(31);

DCL  NAME CHAR(8),
     DESC CHAR(72);

DCL 1 BUFFER BASED(BUFPTR),                 /* NICKNAME BUFFER    */
     2 BUFLEN  FIXED BIN(31),               /*     LENGTH         */
     2 DATA CHAR(LEN REFER(BUFLEN));        /*     DATA           */
```

Figure 102 (Part 1 of 4). Program to display a menu of output devices for end users.

```
DCL 1 QARRAY(1) FIXED BIN(31);
DCL 1 FWORD FIXED BIN(31) BASED(P);
DCL 1 STRING CHAR(72) BASED(P);

DCL OS2PRT CHAR(85)
 INIT(' NICKNAME FAM=0,TOFAM=1,NAME=OS2PRT,TONAME=(*,ADMPMOP),
        DESC="DEFAULT OS/2 PRINTER"');

 FAMILY = 2;              /* Query nicknames for FAM=0 and FAM=2  */  A
 CALL FSINIT;
 /*****************************************************************/
 /* SET UP A NICKNAME USING THE ESSUDS API CALL                   */
 /*****************************************************************/
 CALL ESSUDS(85,OS2PRT);                                              B

 /*****************************************************************/
 /* QUERY LENGTH OF NICKNAME INFO                                 */
 /*****************************************************************/
 CALL ESQUNL(FAMILY,LEN);                                             C

 /********************************/
 /* ALLOCATE THE NICKNAME BUFFER */
 /********************************/
 ALLOCATE BUFFER;                                                     D

 /****************************************/
 /* GET THE NICKNAME INFO INTO THE BUFFER */
 /****************************************/
 CALL ESQUNS(FAMILY,LEN,BUFFER.DATA,BUFFER.BUFLEN);                   E

 /*****************************************************************/
 /* QUERY THE CURRENT DEVICE TO FIND OUT HOW MANY ROWS THERE ARE   */
 /* As this is the first call that relates to the device and a     */
 /* "DSOPEN" call has not been issued, GDDM does an implicit       */
 /* "DSOPEN" to the default device. (eg. On VM, the user console)  */
 /*****************************************************************/
 CALL FSQURY(0,3,1,QARRAY);
 /****************************/
 /* CALCULATE LOOP VARIABLES */
 /****************************/
 PA = UNSPEC(BUFPTR) + CSTG(BUFFER.BUFLEN);
 BUFENDA = PA + BUFFER.BUFLEN;

 /*****************************************************************/
 /* LOOP DOWN THE BUFFER RETURNED FROM ESQUNS                     */
 /* FOR EACH ENTRY GET THE FAM,NAME & DESC VALUES                 */
 /*****************************************************************/
 I=1;
 DO WHILE (PA < BUFENDA);
```

Figure 102 (Part 2 of 4). Program to display a menu of output devices for end users.

```
  UNSPEC(P) = UNSPEC(PA);
  FAM = FWORD;                     /* FAM */
  PA=PA+4;
  UNSPEC(P) = UNSPEC(PA);
  NC  = FWORD*4;
  PA=PA+4;
  UNSPEC(P) = UNSPEC(PA);
  NAME = SUBSTR(STRING,1,NC);      /* NAME */
  PA=PA+NC;
  UNSPEC(P) = UNSPEC(PA);
  DC  = FWORD*4;

  PA=PA+1*4;
  UNSPEC(P) = UNSPEC(PA);
  DESC = SUBSTR(STRING,1,DC);     /* DESCRIPTION PARAMETER        */

  PA=PA+DC;                       /*POINT TO START OF NEXT ENTRY   */
  /*************************************************************/
  /* Define NAME field and put name into it.                   */
  /*************************************************************/
  CALL ASDFLD(I,I,1,1,8,2);
  CALL ASCPUT(I,8,NAME);
  CALL ASFCOL(I,FAM+1);    /* Color of field depends on output     */  F
                           /* family of the nickname.              */

  /*************************************************************/
  /* Define DESC field and put desc in it.                     */
  /*************************************************************/
  CALL ASDFLD(I+100,I,10,1,70,2);
  CALL ASCPUT(I+100,70,DESC);

  /*************************************************************/
  /* UPDATE LINE COUNT. IF END-OF-SCREEN REACHED, PAUSE FOR ANY    */
  /* USER INPUT (EG. ENTER/PF KEY) THEN CLEAR PAGE READY           */
  /* FOR NEXT SCREEN FULL                                          */
  /*************************************************************/
  I=I+1;
  IF I > QARRAY(1) THEN
    DO;
      CALL ASREAD(A,B,C);                                           G
      CALL FSPCLR;
      I=1;
    END;
 END;
  /*********************************************/
  /* DISPLAY ANY LAST FEW ENTRIES ON THE PAGE  */
  /*********************************************/
  IF I>1 THEN
   CALL ASREAD(A,B,C);
```

Figure 102 (Part 3 of 4). Program to display a menu of output devices for end users.

```
 FREE BUFFER;                          /* Free the NICKNAME buffer */

 CALL FSTERM;                                    /* TERMINATE GDDM  */

/* INCLUDE NON-REENTRANT GDDM API CALL DEFINITIONS FILE  */

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINE;
%INCLUDE ADMUPINF;
END DEVDISP;
```

Figure 102 (Part 4 of 4). Program to display a menu of output devices for end users.

Points illustrated by the DEVDISP program

Querying the length of nickname information: Before querying the end user's nickname information, you need to know how much memory you require to store it. You can find the length of the nickname information by issuing an ESQUNL call like the one at **C**. The value returned by this call is the length of the buffer you need to create to store the nicknames.

Setting up a buffer to store nickname information: At **D** the ALLOCATE command is used to allocate storage to the buffer that will hold the nickname information.

Copying the nicknames for an output family into a buffer: The ESQUNS call at **E** queries the nickname information of devices available for the family of output specified and places it in the buffer. In this case the FAMILY parameter has been set to 2 at **A**, so all the valid nickname locations are searched for nicknames that have FAM=2 or FAM=0 (default device) specified. Whatever the value of the FAMILY parameter of the ESQUNS call, nicknames that have no family specified or have FAM set explicitly to 0 are also returned to the buffer.

The ESSUDS call at **B** sets up a nickname for the default print device of a workstation supported by GDDM-OS/2 Link. Because this is the last nickname defined before the ESQUNL and ESQUNS API calls, it is first to be displayed in the list output at **G**.

Displaying nickname information on the user's screen: For each nickname encountered, two alphanumeric fields are created. The name of the device is contained in one and the description parameter from the nickname is displayed in the field next to it.

Because GDDM encounters nickname information for devices of different output families in several different locations, the information in the buffer has information relating to each of the families in no particular order.

To help the end user distinguish between nicknames for different families, the ASFCOL call at **F** uses the FAM parameter of the nickname to color the field in which it is output. The names of devices that are opened for output of family 2 appear in pink and those defined for family-0 output appear in blue.

Additional programming for specific needs: The program in Figure 102 on page 376 just displays a list of valid devices on the end user's screen. With some extra programming, it could be used to enhance the usability of an application.

How a nickname can cause a device definition to be revised completely

"How GDDM compounds device-definition information for a conceptual device" on page 375 shows how, in most situations, the parameters of the DSOPEN call for a device take precedence over the parameters of nickname statements. There are two nickname parameters, however, that take precedence over parameters of the DSOPEN call, namely the TONAME and TOFAM parameters.

With the TONAME parameter on a nickname statement, an end user can cause a device with a different name to the one specified on the DSOPEN call to be opened for output from an application.

With the TOFAM parameter, a nickname statement can cause a device to be opened for a different family of output to that specified on the DSOPEN call.

GDDM accumulates TOFAM and TONAME values during the scan. These statements:

```
ADMMNICK NAME=DEV99,FAM=1,TOFAM=4,TONAME=P3800M3,DEVTOK=IMG240,
         PROCOPT=((IPDSROT,90))
ADMMNICK NAME=DEV99,FAM=1,TOFAM=3,TONAME=SYSPRT2,DEVTOK=S3800W8,
ADMMNICK NAME=DEV99,FAM=1,TOFAM=2,TONAME=A4028P5,DEVTOK=X4028A4,
         PROCOPT=((PRINTCTL,1,8)),DESC="4028 WITH A4 PAPER"
```

are equivalent to:

```
ADMMNICK NAME=DEV99,FAM=1,TOFAM=2,TONAME=A4028P5,DEVTOK=X4028A4,
         PROCOPT=((IPDSROT,90),(PRINTCTL,1,8)),
         DESC="4028 WITH A4 PAPER"
```

At the end of the scan, GDDM updates the DSOPEN parameter list with the latest TOFAM and TONAME values (in addition to the latest DEVTOK and PROCOPT values), and then starts a new scan of all the nickname statements, excluding any already matched. Notice that the parameter list is not updated during a scan: no change is made until all the nickname statements have been scanned, and then the latest values are taken.

Rescanning nicknames when TONAME or TOFAM is specified

A change of output family or name causes GDDM to rescan all nickname statements not already matched. During the rescan, GDDM searches for nickname statements that match the new output family and device name values. It accumulates data from matching statements in the same way as during the first scan; and the latest values still override any conflicting values found earlier in the rescan.

At the end of the rescan, the DSOPEN parameter list is updated again. A DEVTOK or a PROCOPT value that conflicts with a value established during the earlier scan is ignored; in other words, the value established during the earlier scan takes priority. Notice that the rule here is different from the one that applies to conflicting options **within** a scan: in that case, the later one applies.

GDDM performs further rescans of the nickname statements while matching statements continue to be found. Nickname processing ends when there is a complete rescan without a match.

Coding nickname statements within application programs

Most nicknames are coded by systems-support personnel and by end users. If a particular processing option or device token is important for the program, the application programmer can specify it explicitly on the DSOPEN call, which overrides any nickname statements. For this reason it is unlikely that an application programmer would need to code a nickname statement. However, application programmers can supply nickname statements within programs using one of three API calls.

You can supply one nickname statement at a time using the ESSUDS call. For example:

```
CALL ESSUDS(48, 'NICKNAME FAM=1,NAME=A4250,PROCOPT=((COLORMAS,100))');
```

The first parameter is the length of the nickname statement.

Nickname statements coded on the ESSUDS call have the same format as those in the user's defaults file, **source format**. On the other calls by which you can pass nicknames to the device definition, you must supply nicknames in **encoded format**.

Encoding nickname statements to improve application performance

If your application relies heavily on nicknames supplied on API calls, its performance can be enhanced if nicknames are supplied in **encoded format**. Supplying the nickname statements in this format saves processing resources. Using either the ESEUDS or the SPINIT call, you can supply nicknames in encoded format. Whereas you can only pass one nickname per ESSUDS call, with the ESEUDS call, you can pass several nickname statements to GDDM together.

If you use the system programmer interface, you must encode the nicknames you pass to GDDM on the SPIB. Information on creating encoded nicknames can be found in the *GDDM System Customization and Administration* book.

Syntax rules for coding source-format nickname statements

You can code the parameters of a nickname statement in any order. When you do not need to supply a value, you can omit the parameter entirely. Or you can code the keyword but omit the value, as in TOFAM and DEVTOK here:

```
CALL ESSUDS(51,'NICKNAME FAM=1,NAME=ADEV,TOFAM=,TONAME=SCR99,DEVTOK=');
```

Each processing option following the PROCOPT keyword must be enclosed in brackets, with the elements of each option separated by commas. The processing options must be separated by commas, and the complete list of options must be enclosed in another set of brackets. For example:

```
CALL ESSUDS(68,
'NICKNAME FAM=2,PROCOPT=((PRINTCTL,0,1,32,0,0,0,80,0),(INVKOPUV,YES)));
```

A multipart name (as of a CMS file) must be enclosed in brackets and the parts separated by commas:

```
CALL ESSUDS(57,
           'NICKNAME NAME=(OUT1,ADMPRINT,A),TONAME=(PRTFIL,ADMPRINT,G)');
```

In a multipart name specification, you can omit parts or specify them as *. Here is a rather extreme case:

```
CALL ESSUDS(57,'NICKNAME NAME=(,*,C),TONAME=063');
```

This matches a name with a blank as the first part, a * as the second part, and C as the third.

In general, matching multipart names is similar to matching single-part names.

A name-part in the NAME parameter can have a ? as the first or last character or both, meaning “match any characters in this position.” For example:

```
PRINT?    matches any name-part starting with PRINT
?3        matches any name-part ending with 3
?DEV?     matches any name-part containing DEV
```

Specifying device usage using the DSUSE call

In your application you can open several different devices, issuing a DSOPEN call for each one. Opening a device has, in itself, no effect on the program until GDDM is informed that the program needs to **use** a particular device.

DSUSE indicates which device should be used for future output. DSUSE also performs an implicit DSDROP (see “Discontinuing use of a device, using DSDROP” on page 383). This is the format for DSUSE:

```
CALL DSUSE(1,11);        /* Use device 11 as the primary device */
```

- The first parameter states whether the device should be used as a primary device or an alternate device.

  The **primary device** is usually a display screen; it is the main target device for the program's output. It is possible to request “snapshots” or copies of the primary device to be made. In that case, the copies are sent to an **alternate device**, usually a printer. The way in which these copies are made is addressed in “Copying graphics to a printer using call GSCOPY” on page 414.

  So, the first parameter is set to 1 if the device is to be used as a primary device. It is set to 2 to request usage as an alternate device.

- The second parameter is the device identifier–the number assigned to the device when DSOPEN was issued.

At any one time you can have one current primary device and (optionally) one current alternate device.

Discontinuing use of a device, using DSDROP

Issuing a DSUSE call for one primary device implicitly discontinues the usage of the previous primary device (if any). The same applies for alternate devices. You may not have more than one currently active device in each category. If you want to explicitly discontinue the use of a currently active device, this is the format of the call:

```
CALL DSDROP(1,11);    /* Discontinue primary usage of device 11 */
```

The parameters are as for DSUSE: 1 denotes primary usage (2 would be alternate usage), and 11 is the device identifier.

Note that the device is not closed. All its pages and their contained output are maintained. When you issue a DSUSE call to the device again, it is just as you left it–you can even leave a segment open, if you choose.

See “Reinitializing a device, using the DSRINIT call” on page 390 for information about another way of discontinuing use of a device.

Using the default primary device

If you do not issue an explicit DSOPEN for any device, but begin issuing graphics or alphanumerics calls (creating a page, defining an alphanumeric field, or opening a segment, for example), GDDM associates the objects created with the device current at that time.

Because there is none, GDDM issues an internal DSOPEN for the default device, the user console. It then requests, by means of an internal DSUSE, that this device be treated as the primary device.

GDDM uses the device identifier 0 for the default primary device. You should therefore avoid using this identifier yourself. You should use the 0 identifier only if you are sure that end users of your program will not need to use the invoking device once the program is running. The same goes for identifier 1, which may be used as the default alternate device.

If you send some output to the user console (allowing the device to default), and then issue a DSUSE call, to send some output to a different primary device, GDDM issues an internal DSDROP for the user console.

Sending output to a device other than the invoking device

Many applications, when invoked by one device, need to be capable of sending output to printers and plotters and sometimes even to other terminals. There are three different ways that a GDDM application program can do this.

1. When, for instance, the application generates output on the user console that the end user wishes to print, the application can issue a DSUSE call to make a print device the current alternate device.

   To send output to this device, the application just needs to issue a DSCOPY, FSCOPY, or GSCOPY call. There is an example of this use of an alternate device at “Copying graphics to a printer using call GSCOPY” on page 414.

2. The output your application presents to the user on the screen may not be ideal for output on a different type of device such as a printer or plotter. Most printers, for example, have fewer colors, more rows per page, and higher pixel density than graphics displays, and mixing colors in plotted output does not give predictable results.

   Where your application sends output to a print or plot device, it is advisable to issue a DSUSE call to make the printer or plotter the current primary device. Your application can then issue the same (or a similar) set of calls with changes in the parameters to produce output similar to that which appeared on the screen but in a form that suits the hard-copy medium. “Example: Program using two primary devices” shows how to do this. When all the calls have been executed again, an output call such as FSFRCE sends them as output to the new current primary device.

3. Users of your application may need to create output suitable for a device that is very different to the one on which they are running the program. GDDM enables the application to open and use a **dummy device** that can have the characteristics of a target device that is not present. Users can create output for the dummy device and store it.

Using more than one primary device

To send the same picture to two different primary devices, you must execute the graphics calls twice–once with the first device as the current primary and once with the other. With just two devices, it may be sufficient to make one device the alternate device, in which case a simple copy call does what is required.

Example: Program using two primary devices

This section contains a program example to illustrate using two devices. It draws a picture of a grapefruit on two different screens, and then redraws it on the first screen at a smaller size.

```
SCREEN2: PROC OPTIONS(MAIN);
DCL (TYPE,MOD,COUNT) FIXED BIN(31); /* ASREAD parameters         */
DCL PROCOPT_LIST(10) FIXED BIN(31); /* Processing options list   */
DCL NAME_LIST(1) CHAR(8);           /* Device-control name list  */

NAME_LIST(1)='061';                 /* CMS device address        */

CALL FSINIT;

/******************/
/* OPEN DEVICE 15 */
/******************/
CALL DSOPEN(15,1,'*',0,PROCOPT_LIST,1,NAME_LIST);/*Open device 15*/
CALL ASDFLD(2,3,8,1,29,2); /* Device 15 has been opened, but not */  A
                      /* yet specified for any usage. This      */
                      /* ASDFLD therefore causes an internal    */
                      /* DSOPEN of the user console (and a      */
                      /* matching DSUSE), not device 061.       */
                      /* Alphanumerics and graphics are         */
                      /* associated with the user-console's     */
                      /* default page                           */

CALL ASCPUT(2,29,'SAMPLE OUTPUT TO USER-CONSOLE');
CALL GSFLD(4,1,28,80);        /* Define 28-row graphics field    */
                              /* For the user-console            */
CALL GSPS(1.0,1.0);           /* Ensure square drawing area      */  B

CALL GSWIN(0.0,20.0,0.0,20.0);/* Choose coordinate system        */  B

CALL GSLSS(3,'ADMCOLSD',0);   /* Load GDDM 64-color pattern set  */  C

CALL GRAPE_FRUIT;             /* Call user subroutine to draw    */
                              /* a picture of a grapefruit       */
CALL ASREAD(TYPE,MOD,COUNT);  /* Send output to user-console     */
/*********************/
/* DROP USER-CONSOLE */
/*********************/
CALL DSDROP(1,0);             /* Drop the user-console from      */
                              /* primary usage, preparatory to   */
                              /* sending output to device 15.    */
/************************************/
/* MAKE DEVICE 15 THE CURRENT DEVICE */
/************************************/
CALL DSUSE(1,15);             /* Use device 15 as the primary    */
CALL ASDFLD(2,3,8,1,38,2);    /* This alpha field is assigned to */  A
                              /* the default page of device 15   */

CALL ASCPUT(2,38,'SAMPLE OUTPUT TO DEVICE AT ADDRESS 061');

CALL GSFLD(6,1,17,80);        /* Define 17-row graphics field    */
                              /* for device 15                   */
CALL GSPS(1.0,1.0);           /* Ensure square drawing area      */  B

CALL GSWIN(0.0,20.0,0.0,20.0);/* Choose coordinate system        */  B
CALL GSLSS(3,'ADMCOLSD',0);   /* Load GDDM 64-color pattern set  */  C
```

Figure 103 (Part 1 of 2). Program using two displays, each as the primary device

```
CALL GRAPE_FRUIT;             /* Reexecute subroutine to draw    */
                              /* a picture of a grapefruit       */
CALL ASREAD(TYPE,MOD,COUNT);  /* Send output to device 15        */

CALL DSDROP(1,15);            /* Temporarily drop device 15      */
/*****************************************************/
/* USER-CONSOLE AUTOMATICALLY MADE CURRENT DEVICE */
/*****************************************************/
CALL ASCPUT(2,29,'SECOND OUTPUT TO USER-CONSOLE');                 D
CALL GSCLR;                   /* Clear previous graphics -       */
                              /* remove the large grapefruit     */
CALL GSWIN(-10.0,30.0,-10.0,30.0);
                              /* Redefine the window             */ E


CALL GRAPE_FRUIT;             /* Draw much smaller grapefruit    */
                              /* in the center of the screen     */
CALL ASREAD(TYPE,MOD,COUNT);  /* Send output to user-console     */

CALL FSTERM;                  /* Terminate GDDM                  */
GRAPE_FRUIT: PROC;
CALL GSSEG(0);                /* Open graphics segment           */
CALL GSCOL(7);                /* Set color to neutral to enable  */
                              /* use of GDDM 64-color set        */
CALL GSPAT(121);              /* Grapefruit color                */
CALL GSMOVE(10.0,4.0);        /* Move to start of graphics area  */
CALL GSAREA(0);               /* Open a graphics area            */
CALL GSARC(10.0,10.0,360.0);  /* Draw outline of the grapefruit  */
CALL GSENDA;                  /* Close the graphics area         */
CALL GSPAT(0);                /* Reset shading pattern to solid  */
CALL GSCOL(6);                /* Set color to yellow for stalk   */
CALL GSMOVE(14.0,10.5);       /* Move to start of stalk          */
CALL GSAREA(1);               /* Start area with drawn boundary  */
CALL GSARC(14.0,14.0,91.67);  /* One edge of the stalk           */
CALL GSLINE(18.0,13.0);       /* End of the stalk                */
CALL GSARC(14.0,14.0,-91.67); /* Other edge of the stalk         */
CALL GSENDA;                  /* End area representing stalk     */
CALL GSSCLS;                  /* Close graphics segment          */
END GRAPE_FRUIT;              /* End user subroutine             */
%INCLUDE(ADMUPINA);           /* Include DCLs of GDDM entries    */
%INCLUDE(ADMUPIND);
%INCLUDE(ADMUPINF);
%INCLUDE(ADMUPING);
END SCREEN2;
```

Figure 103 (Part 2 of 2). Program using two displays, each as the primary device

Points illustrated by the SCREEN2 program

This example shows a number of considerations that are peculiar to a program using two different primary devices.

Duplicate identifiers: The statements marked **A** both define fields with an identifier of 2. This is not an error or conflict of any sort, because the fields belong to different pages (and also to different devices).

The device is at the head of the hierarchy. Each device has its own set of pages, each with their own graphics and alphanumerics.

The rules about not using the same identifier twice apply only within the next highest element in the hierarchy. For example, your first device can have a page with identifier 5; so can your second device. One of a device's pages may have an alphanumeric field with identifier 32; so may another such page.

Viewport matching window: To ensure that the grapefruit is circular, the aspect ratio of the window must match that of the viewport. This is done in the statements marked **B** by setting a square picture space (and therefore a square viewport), and by using a window of 20 units in each direction.

Default primary device: Just after the DSDROP of device 15, GDDM meets an ASCPUT call **D** . As there is no current primary device at that time, GDDM assumes that the default device should be used (as it did at the start of the program). The user console is already open, so GDDM issues just an internal DSUSE to make the user console the current primary device again.

Scope of symbol sets: The scope of a symbol set is the device. This means that the application program must load a separate symbol set for each device, even if the loads are of the GSLSS type. In the example, the 64-color pattern set has to be loaded twice (once for each device) at **C** .

You may load a vector symbol set for one device and give it an identifier of 194. You may then load a different vector symbol set for another device and give it the same identifier of 194. Although this is not an error, using a different identifier would make the program more easily understandable.

Enlarging window to shrink the graphics: The subroutine GRAPE_FRUIT draws the fruit within the coordinate ranges x:0 through 20, y:0 through 20. When the window itself has these ranges, the subroutine's output fills the viewport. If the subroutine is reexecuted under a larger window as defined at **E** , the output fills correspondingly less of the viewport. In the last section of the program, the grapefruit is redrawn in the central quarter of the viewport.

Opening and using a dummy device

When developing and testing a new program it may be convenient to do so without attaching it to the eventual target device. This is possible by requesting a **dummy device** at DSOPEN time. The “name” of the device must be set to `' '` (blank).

```
NAME_LIST(1)=' ';    /* Set blank name to indicate dummy device */

/*    DEVICE-ID  FAMILY  DEV_TOKEN  OPTIONS        WHICH DEVICE */
CALL DSOPEN(11,    1,    'L79A3',  0,PROCOPT_LIST,  1,NAME_LIST );
```

No output is sent to such a device. It is merely a convenience that would allow, for instance, the generation of output for a 3279 display at a 3472-G (nonPS) terminal.

You must specify a device token for a dummy device. You cannot specify a device token of `*` to get GDDM to query the device characteristics itself because the target device isn't really there. The only way GDDM can know the device characteristics of a dummy device is by the DSOPEN passing a device token. An explicit device token is therefore compulsory.

Example: Program using a dummy device to create a stored picture

Dummy devices are sometimes used in combination with an FSAVE call to save pictures on auxiliary storage. Such pictures, however, are only suitable for transmission to a device **of the same type** as the one used to save them.

This example, which could be run without a user console in batch mode, illustrates the situation. It creates two saved representations of the architect's design–one for later display on a 3472-G, one for later display on a PC with a high resolution 8514/A screen.

```
SAVE2: PROC OPTIONS(MAIN);
CALL FSINIT;

NAME_LIST(1)=' ';    /* Set blank name to indicate dummy device */

/*********************************************************************/
/* OPEN DUMMY DEVICE WITH CHARACTERISTICS OF 3472-G                  */
/*********************************************************************/
/*    DEVICE-ID  FAMILY  DEV_TOKEN  OPTIONS        WHICH DEVICE */
CALL DSOPEN(11,    1,   'L3472G',  0,PROCOPT_LIST,  1,NAME_LIST );
/*********************************************************************/
/* MAKE DUMMY DEVICE THE CURRENT DEVICE                              */
/*********************************************************************/
CALL DSUSE(1,11);                      /* Use dummy device with */
                                       /* 3472G characteristics */
CALL A_DRAWING; /* Call subroutine to create architect's drawing*/
                /*      (on the default page of device 11)      */
CALL FSSAVE('DIAG3472');   /* Save diagram for later FSSHOR-ing */
                           /* on a 3472-G display screen        */
/*********************************************************************/
/* OPEN DUMMY DEVICE WITH CHARACTERISTICS  OF A PC WITH 8514/A SCREEN */          */
/*********************************************************************/
/*    DEVICE-ID  FAMILY  DEV_TOKEN   OPTIONS       WHICH DEVICE */
CALL DSOPEN(12,    1,    'LPCC4',  0,PROCOPT_LIST,  1,NAME_LIST );
/*************************/
/* DROP DUMMY 3472-G     */
/*************************/
CALL DSDROP(1,11);              /* Must drop one primary device */
                                /* before using another         */
/*********************************************************************/
/* MAKE DUMMY PC (8514/A) THE CURRENT DEVICE                         */
/*********************************************************************/
CALL DSUSE(1,12);               /* Use dummy device with        */
                                /* 24-line 8514 characteristics */
CALL A_DRAWING; /* Call subroutine to create architect's drawing*/
                /*      (on the default page of device 12)      */
CALL FSSAVE('DIAG8514');   /* Save diagram for later FSSHOR-ing */
                           /* on a 24-line 8514 display screen  */
```

Figure 104 (Part 1 of 2). Program using a dummy device to create a stored picture

```
CALL FSTERM;                    /* Terminate GDDM              */
A_DRAWING: PROC;
CALL GSPS(1.0,44.5/117.0);        /* Match aspect ratio of plan */
CALL GSWIN(0.0,117.0,0.0,44.5);   /* Window in meter units      */
CALL GSSEG(0);                    /* Create graphics segment    */
CALL GSCOL(1);                    /* Set color to blue          */
CALL GSMOVE(102.4,35.0);          /* Start drawing diagram      */
    and so on.                    /* Continue drawing           */

CALL GSSCLS;                      /* Finish drawing             */
CALL ASDFLD(1,1,10,1,70);         /* Add alphanumeric data      */
    and so on.

END A_DRAWING;                    /* End of subroutine          */

%INCLUDE(ADMUPIND);               /* Include GDDM entry-points  */
%INCLUDE(ADMUPINF);
%INCLUDE(ADMUPING);

END SAVE2;
```

Figure 104 (Part 2 of 2). Program using a dummy device to create a stored picture

Closing a device using the DSCLS call

When you do not intend reusing a device, it should be explicitly closed to release the associated resources. This is the format of the close call:

```
CALL DSCLS(15,0);       /* Close device 15 and erase the screen */
```

All page contents and symbol sets for this device are now released. GDDM retains no memory of the device. Should a new device be opened with identifier 15, it need bear no relationship to the device now being closed.

The first parameter is the device identifier. The second is an option that may be set to values that determine how the device is treated when it is closed.

If the device is a terminal opened for family-1 output, you can specify whether or not to erase the screen and whether or not to lock the keyboard when the device is closed.

If the device is a printer opened for output other than family-1, you can use this parameter to determine whether the print file is to be saved or erased when the device is closed.

The effect of these options is subsystem-dependent. The variations in these effects are described in the *GDDM Base Application Programming Reference* book.

Reinitializing a device, using the DSRINIT call

If you want to discontinue the use of a device and also release resources, such as symbols sets, that have been allocated to it, you can issue a DSRINIT call in the program. This returns the device to the state it was in when first opened.

Chapter 19. Designing device-independent programs

The information in this section is aimed to help you write application programs that perform equally well on many different devices.

When you write a GDDM application program, you may have a very good idea of who the end users will be and even what kinds of terminal, workstation, printer, and plotter they use. It may be quite straightforward to write an application that functions perfectly with just those devices but the more specific your program is to one type of machine the less it is worth to you.

Programs that are written so that they are not specific to any one device or family of devices are of more use to more people and have a longer useful life span if they continue to function with new devices.

Device dependence in GDDM application programs

GDDM provides much of its function in a device-independent way. However, there are two ways that some GDDM functions are affected when programs invoke them for particular devices.

Device variation
An API call and its parameters are valid, but the current device does not give the same results as the general case usually described in this guide.

GDDM aims to exploit the capabilities of many different devices, so relatively few devices support the full range of parameter values for many calls. Using GSCOL to set the current color of an area to orange on an eight-color display is such a case.

Device dependence
The program issues an API call that is only valid for a particular device or group of devices. If the current device is not one of these, a GDDM error message is displayed.

Attempting to use alphanumerics on a plotter is a case in point.

The information in this section is aimed to make you aware of potential situations in the first category, and to help you avoid or cope with situations in the second.

Coping with device variation and dependence in your programs

You can do a great deal to make your programs less dependent on specific devices, if you remember these two tips when you write them:

- Use **default parameter settings** for API calls as much as possible.
  When a parameter is not specified explicitly on a GDDM call, the call usually takes the parameter setting from the standard defaults set by the systems support personnel who customize the system. By omitting an explicit parameter on a call, you give end users the freedom to choose parameters for the defaults that best suit their devices. As more devices are supported, users can change their defaults files, so your application doesn't need to be changed to work with them.

© Copyright IBM Corp. 1982, 1996

- Use the Base API **query calls** in your program to determine the characteristics of devices being used with the application.
  You can then include special routines for any device that deviates from the normal support. Some query calls are described in this section–see the *GDDM Base Application Programming Reference* book for a full list.

Avoiding dependencies when opening and using devices

When you open a device in an application program, GDDM needs to know the device's characteristics in order to process output to suit it. You can specify this information for GDDM in the DEVTOK (device token) parameter of the DSOPEN call or an associated NICKNAME statement. See Chapter 18, "Device support in application programs" on page 371. However, many programmers prefer not to supply a specific device token and instead code an "*" (asterisk) on the parameter. This forces GDDM to obtain the information itself, by querying the device.

Querying the conceptual device, using the DSQDEV call

You can issue a DSQDEV call in your program to query the DSOPEN information for any device that has been opened, explicitly or implicitly. This call returns information about the way a device has been defined to GDDM rather than about its physical characteristics.

If GDDM opens a primary or secondary device by default, you can issue a DSQUSE call to query its identifier. If you then use the identifier returned by the DSQUSE with a DSQDEV call, GDDM returns the device token and processing-option information that the systems support personnel or end user has specified as the defaults for that device.

The information returned includes the output family, the device name, the device token, and the processing options actually used for the DSOPEN, that is, the values after any nickname processing.

If any of these parameters could conflict with the processing of your program, you can call a routine that overrides them with an explicit DSOPEN or NICKNAME statement.

Querying characteristics of the physical device, using the FSQURY call

With the FSQURY call, you can query the characteristics of the physical device that is being used as the primary device (but not one being used as an alternate device). This call can return information about the general, graphics, partitions, image and plotter-related characteristics of the primary device.

Here is an example of how an application can use FSQURY to determine the default page size of the current primary device:

```
DCL FS_ROWS(1) FIXED BIN(31) INIT(0);
DCL DEPTH FIXED BIN(31);

/*        DEVICE  1st  COUNT    ARRAY                                  */
CALL FSQURY(0,     3,    2,   FS_ROWS);    /* Query number of lines on */
                                           /* screen of user console   */
DEPTH = FSROWS(1);

CALL ASDFLD(1, DEPTH, 1, 1, 36, 2);
CALL ASCPUT(1,23,'PF1=Help    PF4=Print'); /* Place PFkey information  */
                                           /* on last line of screen   */
```

If you know which GDDM functions can be subject to variation and with which devices variations occur, you can use these query calls to call routines in your program that take special actions.

Setting up a GDDM hierarchy that suits most devices

Because the device is the highest object in the GDDM hierarchy, it has implications for all the other objects. When you define any objects in the hierarchy, it is best to allow them to take default row and column values.

If you want to define a partition set or a GDDM page that doesn't cover the whole display screen or printer page, you can use the FSQURY call to determine how many rows and columns the device has and then define the object in terms of this. This ensures that the objects maintain their relative proportions on different types of device.

Even if you allow objects in the hierarchy to take the default values, you may still have device-dependency problems. The graphics field is defined in terms of rows and columns, which means that the aspect ratio of graphics drawn on it will vary across devices with different cell sizes. You can ensure that the aspect ratio of graphics drawn by your program is the same on all devices by defining the picture space on the graphics field with the GSPS call or by defining a uniform graphics window using the GSUWIN call.

Device considerations for graphics functions

Positioning graphics in the viewport: If your application sends output to a real or dummy device opened for family-4 output, graphics primitives that are drawn or loaded touching the right-hand edge of the viewport are subject to truncation.

This only becomes noticeable, if the primitives are scaled up by a segment transformation call such as GSSAGA or GSSTFM.

Because end user's installations may use nicknames to redirect output from your application to family-4 devices, it is best if you avoid drawing graphics that touch the right-hand edge of the viewport.

Specifying color attributes: When you use the GSCOL call in an application, remember that different devices support different numbers of colors. Some users of the program may have workstations that can display up to 256 different colors and others may be using monochrome displays and printers.

You can help users by specifying basic colors (those identified to GDDM by the numbers 1 through 8) in application programs. If you use more colors than the device supports, GDDM maps them back onto supported colors before displaying

the graphics. The way GDDM does this mapping for devices with different degrees of color support is shown in the *GDDM Base Application Programming Reference* book. However, the more unsupported colors used, the more likely it is that adjacent graphics primitives will be output with the same color, becoming indistinguishable.

To cope with this, you can use combinations of colors and shading patterns to shade different areas. Then if the current device only supports the basic eight colors, areas that are shaded orange in the program will be distinguished from those shaded red by the shading pattern used.

If you cannot avoid using colors that some devices cannot display, print, or plot, you should consider invoking GDDM's color/pattern translation function. This maps unsupported colors (and shading patterns) onto supported colored shading patterns so that colored areas can be distinguished. It is specified by the PATTRAN processing option when the device is opened.

An FSQURY call for the device can determine how many colors it supports. If there is a risk that your program will exceed the device's color support, you can check, using a DSQDEV call for the device, whether a translation table has already been specified by means of a NICKNAME statement in the user's defaults file. If it no translation table has been specified, or the table specified doesn’t meet your needs, you can call a routine containing a DSOPEN call or a NICKNAME for the device that specifies the PATTRAN processing option.

Specifying color-mixing attributes: In programs that draw different graphics primitives in the same part of the graphics window, you are advised to specify only overpaint mode for foreground color mixing. Not all devices support underpaint mode but all devices support overpaint mode. Using overpaint mode, you need to plan ahead, which primitives you want to overlap others in the program's output. You draw the lower primitives first and draw the primitives that are to be superimposed on them later in the program. It is also best to allow background mixing to use the default mode for that device. You can use FSQURY to determine which color-mix modes are supported for a given device.

Note: Even when a device supports a particular foreground mix mode and a background mix mode, it may not support a combination of the two. The mix-mode combinations supported by different devices are described in the *GDDM Base Application Programming Reference* book.

If the output device is a plotter, you should not specify mix mode in the foreground, unless you genuinely want the undefined color that results from mixing the pens' inks.

Specifying line-width attributes: Some devices can only transmit single-width lines as output. In your program, you can still specify lines of multiple thickness, (using GSLW or GSFLW), and lines of fractional thickness, (using GSFLW). In general, the support for lines of nonstandard thickness is better on printers than on displays. If the device supports nonstandard line widths specified in your program, the desired output will be generated. Otherwise, lines of standard thickness for that device are used. The line widths supported by different devices are described in the *GDDM Base Application Programming Reference* book.

Specifying shading patterns for graphics areas: If possible, limit the shading patterns used in your application to the 16 GDDM-supplied shading patterns.

If your application needs to load and use additional shading patterns, you are advised to specify the PATTRAN processing option on a NICKNAME statement for the output device. Unsupported shading patterns are then mapped onto supported patterns and given a different color.

Drawing graphics primitives: It is best if you **always open a graphics segment before issuing any calls to draw a primitive**. Different devices treat primitives drawn outside segments in different ways. Because applications discard such primitives once they have displayed them on the screen any local operation on the device, such as receipt of a system message, causes them to be lost. If another partition is opened (such as User Control), overlapping part or all of a primitive that is not in a segment, the obscured parts are not redrawn when the overlapping partition is closed.

Storing and loading graphics: If your application needs to load a saved graphics picture, make sure that it is in a graphics data format (ADMGDF) file, which you can load using the GSLOAD call. If your application needs to store a graphics picture, use the GSSAVE call, rather than the FSSAVE call. GSSAVE stores graphics in files of graphics data format, which can be displayed on any device. FSSAVE, however, saves graphics in a format that can be displayed only on the same type of device as the one used when it was created.

Device considerations for graphics-text functions

Positioning graphics text with respect to graphics: If you want to position text accurately with respect to graphics, it is best to use graphics text of mode 3. The other graphics text modes (1 and 2) use image symbols which may change position relative to graphics, if output on a different display or plotter.

If the program does use graphics text of mode 1 or mode 2, and uses a GSCOPY call to send the graphics to an alternate device, you can use the GSARCC call to maintain the relative positioning of text and graphics. GSARCC offers control either over the position of image symbols or over the aspect ratio of graphics.

Using the right code points for graphics text: An ESQEUD call in your program can determine whether a country extended code page (CECP) has been specified as a default for use with application programs. If none has been specified, you can invoke a routine that specifies one explicitly, see “Using GDDM to convert character code pages for international applications” on page 247.

Selecting symbol sets by device type: If your application program is to be used with different devices, it may be necessary to control symbol set loading on the basis of cell size. The GDDM symbol-set naming convention can help you in this task. The symbol-set name is specified as a parameter of the loading call. If the last character of the name is the period character “**.**”, GDDM replaces it with another character, specific to the cell size of the current device.

In this way, a symbol set with a cell-size definition that matches the device in use can be retrieved from auxiliary storage and loaded. In a particular application, if a display containing PS is to be printed, this function allows the selection of a symbol set specific to the printer when printing begins.

For information about which sets are loaded for a particular device cell size, see the *GDDM Base Application Programming Reference* book.

Device considerations for alphanumeric functions

Sending alphanumeric output to a device: Some devices, such as plotters, do not support alphanumeric output sent to them from programs. You can use FSQURY to query which alphanumeric functions, if any, are supported by a device.

Positioning alphanumeric text with respect to graphics: The symbols used for alphanumerics, like those for graphics text of modes 1 and 2, are image symbols. To prevent alphanumeric text from changing its position relative to graphics you can use GSCOPY and GSARCC in the same way as for graphics text of modes 1 and 2, see "Device considerations for graphics-text functions" on page 395.

Using the right code points for alphanumeric text: An ESQEUD call in your program can determine whether a country extended code page (CECP) has been specified as a default for use with application programs. If none has been specified, you can invoke a routine that specifies one explicitly, see "Using GDDM to convert character code pages for international applications" on page 247.

Mixing single- and double-byte characters in alphanumeric fields: If your application mixes double-byte and single byte characters in alphanumeric fields, you should use an FSQURY call to determine whether the device supports such mixing.

Device considerations for interactive-graphics functions

Programming for logical input devices: If you invoke any but the most basic interactive graphics functions, your program becomes specific to a limited number of devices. Most devices support programs that specify the alphanumeric cursor, the graphics cursor, the ENTER key, the function keys, or the mouse keys as logical input devices. But in order to exploit the higher level interactive-graphics functions properly, your application must become specific to a relatively small group of devices. The FSQURY call can determine which interactive-graphics functions are supported by a device. You can find information on the devices that support high level function in the *GDDM Base Application Programming Reference* book.

Device considerations for partitioning functions

You should avoid any dependency on hardware partitions, if possible. Hardware partitions are supported on a limited number of displays. Emulated partitions are supported on all displays.

Applications for devices that use programmed symbols

For a device such as the 3279, the number of PS cell definitions that must be transmitted to the terminal to create a picture depends on the picture's complexity. For example, a multicolored symbol requires three times as many bits for its definition as does the same symbol in monochrome. A complex chart with many lines, shaded areas, colors, and vector text needs much more PS information than a simple one using two or three colors and hardware text characters. Because the more complex picture requires more dynamic storage and takes longer to appear on the screen, you will generally need to achieve some balance between the picture requirements of your program and the operating environment.

Using PS with graphics: When GDDM is constructing a picture, the assumption is made that all PS stores in the device are available for use except those that have either been loaded with symbol sets, or explicitly reserved by the application program. Because the number of PS stores is limited, if an application program uses both additional PS character sets and graphics construction, special attention to PS allocations may be required. This is especially true for printers, because only one PS store can hold a multicolor symbol set.

In general, PS stores should be loaded with any additional symbol sets before graphics picture construction is started, because the PS stores are also used for picture display. An attempt to load a symbol set when graphics are displayed is usually rejected by GDDM. Only when all graphics items are deleted from all pages do the PS stores become released for loading symbol sets.

If the programmer anticipates the need to load a PS store while graphics data is present, the PSRSV call is available to **reserve** a PS store. This must be done before any graphics calls are issued. The specified PS store is not used for graphics data, and is explicitly referred to in the call statement to load the symbol set. When the symbol set is no longer needed, the symbol set can be released from the reserved PS store, and another symbol set can be loaded, or, the PS store itself can be released.

In a windowing environment, the PS stores are allocated in the following order:

1. For symbol sets in the active window
2. For graphics in the active window
3. For graphics for window borders (all windows)
4. Any remaining PS slots are allocated for symbol sets and graphics in non-active windows.

Checking the complexity of graphics output: If your application creates a complex graphics picture on the GDDM page, the picture may be subject to degradation (PS overflow) on a display that uses PS stores. (This is more likely if symbol sets have been loaded.)

The FSCHEK call enables the program to determine whether the next output operation would exceed the PS limits of the display. FSCHEK returns an error condition, if the picture would cause PS overflow.

To diagnose the error condition, the program can issue an FSQERR call. This call is described in “Querying the GDDM error record, using FSQERR” on page 133. This is an example of the code required:

```
DCL ERROR_PARM(2) FIXED BIN(31);

CALL FSCHEK;                  /* Check picture complexity       */
CALL FSQERR(8,ERROR_PARM);    /* Query the most recent error    */

/**A returned error code of 273 indicates overflow would occur **/

IF ERROR_PARM(2)=273 THEN DO; /* Overflow would occur on output */
CALL DSRNIT(9, 0);               /* Reinitialize device dropping   */
                                 /* all loaded symbol sets.        */
CALL DSUSE(9, 1);                /* Use device 9 as primary device */
  :
  :
END;
```

FSCHEK only checks the picture – it does not perform any output.

Keeping end users interested: Depending on system use, picture complexity, and other factors, several seconds may be required to complete a graphics display on the terminal. In designing interactive application programs that generate pictorial displays, the programmer should attempt to provide some response to the user as soon as possible after the last user action.

For example, if the application program must search a large data base or perform extensive calculations before the picture can be constructed, you might consider displaying a message, indicating that work is in progress.

As another example, if data is readily available and the user expects to see the picture, the program should force some information (such as chart axes or a title) onto the screen as soon as possible. The information presented should be something the user would like to see early in the picture generation.

You should note, however, that excessive forcing of partial graphics information can increase the total time needed to send the picture to the terminal, by increasing the number of transmissions.

Device considerations for image functions

For all display devices, other than the IBM 3193, that support the use of graphics, GDDM performs all image transforms and output by emulation. If you write an image processing application that is likely to be used on such devices, you need to give consideration to the reduced performance that this emulation entails.

Chapter 20. Sending output from an application to a printer

You can print output from an application by issuing one of the GDDM output calls while a printer is in use as the current primary or alternate device. The way this printing is done, however, is determined more by the DSOPEN call that opens the print device than by the call that sends the output to it. Most of this section is concerned with opening print devices prior to sending the output.

Overview of printing with GDDM

There are four distinct methods of sending output to a printer. They are summarized in Figure 105.

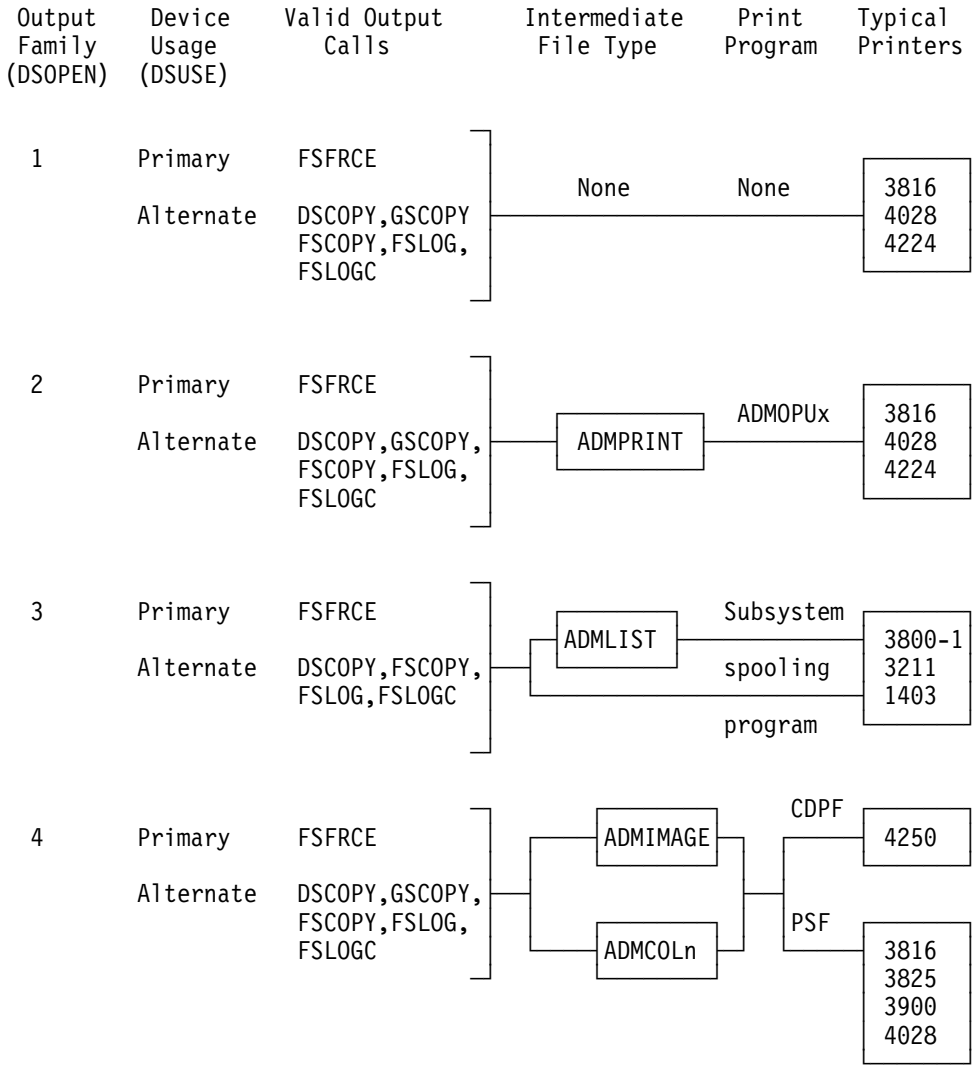


Figure 105. Overview of GDDM's support for printers

The printer that an application can use for output is determined by

- The type of data output; whether the output contains graphics, alphanumerics, image or a combination of these.

© Copyright IBM Corp. 1982, 1996

- The end user's printer installation; whether printers are attached directly or remotely and NICKNAMEs have been set to manage the spooling of print files.
- The end user's subsystem.

The family parameter of the DSOPEN call defines the output method, as follows:

Family-1 This sends output to a device attached directly to the end user's session. The end user has complete control over such devices.

This form of output can be used for printing on the CMS, CICS, and IMS subsystems. End users of GDDM-OS/2 Link and GDDM-PCLK with local printers attached to their workstations can avail of family-1 printing regardless of the subsystem on their host systems.

Family-2 This specifies **queued printing**. Your application program sends its output to a GDDM-created print file. The file is then printed by the GDDM Print Utility Program, ADMOPUx (where “x” depends on the subsystem), which is described in “Printing GDDM family-2 print files” on page 417. A print device that is opened for family-2 output is usually a printer associated with the subsystem and shared between several users.

Family-3 This instructs GDDM to send output to a **system printer** driven by a subsystem spooling program. Your program's output is stored in a GDDM-created file before being passed to the spooling program.

Family-4 This specifies **advanced function printing** for the application program's output.

The data from the program is saved in a GDDM-created file which, depending on the nature of the output data stream, is passed to different utility programs for printing

IBM provides two utility programs to print family-4 output. If the current device is an IBM 38xx, 3900, 4028, 4224, or 4234 printer opened for family-4 output, GDDM creates a print file that contains Advanced Function Presentation Data Stream, (AFPDS). Such print files are passed to the Print Services Facility (PSF) to be printed.

You can also specify family-4 output when your program opens an IBM 4250 printer. In this case, GDDM creates a print file containing Composed Document Presentation Data Stream (CDPS), which must be processed by the Composed Document Printing Facility (CDPF) before going to the printer.

Before printing, a number of text and graphics files may be combined in a page composition process.

Some printers used for family-4 output can, in other configurations, be used for family-1 or family-2 output.

Note: The family of output specified in the application program may not be the one created when the program is run. A nickname statement for family-2 output, for instance, could use the TOFAM=4 parameter to change the DSOPEN so that an application that is programmed to generate family-2 output creates family-4 output instead.

More information about each method of printing is given below.

Family-1 output: GDDM directly attached printers

You can treat a printer as an ordinary family-1 device, if it is attached directly to the end user's host session. With a printer opened for family-1 output as the current device you can use code to create your graphics and alphanumerics similar to that for a display device. The DSOPEN determines the destination of the FSFRCE output. In a CMS environment, the code to print family-1 output on an IBM 4224 printer would look like this:

```
DCL PROCOPT_LIST(10) FIXED BIN(31); /* Processing options list  */
DCL NAME_LIST(1) CHAR(8);           /* Device-control name list */

NAME_LIST(1)='061';                 /* CMS device address of    */
                                    /* 4224 color printer       */

/*    DEVICE-ID  FAMILY  DEV_TOKEN    OPTIONS       WHICH DEVICE*/
CALL DSOPEN(19,    1,      '*',    0,PROCOPT_LIST,  1,NAME_LIST );
                                    /* Open real printer at     */
                                    /* address X'061', using    */
                                    /* all the default values.  */

CALL DSUSE(1,19);                   /* Use device 19 (printer)  */
                                    /* as the primary device    */

CALL GSFLD(1,1,60,80);              /* Define graphics field    */
                                    /* 60 rows by 80 columns    */
CALL GSSEG(0);
CALL GSMOVE(24.0,70.0);             /* Start to draw graphics   */
      and so on...
CALL FSFRCE;                       /* Send 1st output to printer */
```

Figure 106. Example using an IBM 4224 printer for family-1 output.

Before the program is executed, the printer is attached to the user's VM machine at virtual address “061.”

The default page size is determined by the device token supplied in the DSOPEN call. If you do not specify an explicit device token for a printer opened for family-1 output, GDDM queries the printer to determine the page size. As always, the graphics field defaults to the page size.

For family-1 printing under CMS, the end user of your program needs exclusive control of a directly attached print device. Apart from users with printers attached to their workstations, it is unlikely that a printer would be devoted exclusively to one user's virtual machine. Installations can cope with this by spooling the output to RSCS (the Remote Spooling Communication System) Networking. You need to specify two processing options; the nicknames facility is generally the simplest way of doing this. Here is a suitable DSOPEN call:

```
NAME_LIST(1) = 'RSCSPRT1';
        /* ID  DEV-FAMILY  DEV-TOKEN  PROCESSING OPTIONS  DEV-NAME */
CALL DSOPEN(5,      1,        '*',    0,PROCOPT_LIST,   1,NAME_LIST);
```

and the required nickname statement:

```
ADMMNICK FAM=1,NAME=RSCSPRT1,TONAME=PUNCH,DEVTOK=X4224SS
        PROCOPT=((CPSPOOL,TO,RSCS),(CPTAG,REMPRT7,PRT=GRAF)
        DESC="4224 spooled printing")
```

The TONAME parameter sends the output to the virtual punch. The CPSPOOL processing option first spools the punch file to RSCS, and the CPTAG option tags it with the real printer name (REMPRT7) and an option indicating that the file is a GDDM graphics one. A device token for the real printer must be supplied in the DEVTOK parameter if there is not one on the DSOPEN. More information about nicknames is given in Chapter 18, “Device support in application programs” on page 371 and complete lists of device tokens and processing options are provided in the *GDDM Base Application Programming Reference* book.

A workstation running GDDM-OS/2 Link or GDDM-PCLK can have a printer or plotter attached to it as an **auxiliary device**. For family-1 output on a workstation's auxiliary printer, you need to open the device like this:

```
DECLARE PROCOPT_LIST(1)  FIXED BINARY(31);
DECLARE NAME_LIST(2)     CHARACTER(8);

NAME_LIST(1) = '*';
NAME_LIST(2) = 'ADMPMOP';  /* Default GDDM-OS/2 Link output device */
     or                   /* (printer or plotter)                 */
NAME_LIST(2) = 'ADMPCPRT'; /* Default GDDM-PC Link printer         */

  /* DEVICE-ID  FAMILY  DEVICE-TOKEN     OPTIONS          NAME     */
CALL DSOPEN(99,   1,       '*',       0,PROCOPT_LIST,  2,NAME_LIST);
```

Family-2 output: Print files for GDDM queued printers

The output to a queued printer is first written to a print file, passed to the GDDM Print Utility, and then sent to a printer. The exact mechanism varies according to the subsystem (see “Printing GDDM family-2 print files” on page 417).

When a printer is opened for family-2 output, various parameters may be set. They form a **print-control option group** within the processing options list (see the description of processing option 4 in the *GDDM Base Application Programming Reference* book).

Here is an example of code to open a queued printer:

```
DCL PROCOPT_LIST(10) FIXED BIN(31);/* Processing options list   */
DCL NAME_LIST(1) CHAR(8);          /* Device-control name list  */

PROCOPT_LIST(1)=4;                 /* Print control option code */
PROCOPT_LIST(2)=8;                 /* No. of fullwords following*/
                                   /* in  this  option  group   */
PROCOPT_LIST(3)=0;                 /* Do not print header page  */  A
PROCOPT_LIST(4)=2;                 /* No. of copies required    */  B
PROCOPT_LIST(5)=10;                /* Maximum depth for FSLOG   */  C
PROCOPT_LIST(6)=3;                 /* Depth of top margin       */
PROCOPT_LIST(7)=5;                 /* Width of left margin      */
PROCOPT_LIST(8)=0;                 /* Depth of bottom margin    */  C
PROCOPT_LIST(9)=80;                /* Maximum width for FSLOG   */  C
PROCOPT_LIST(10)=0;                /* Default code page mapping */  D
```

```
NAME_LIST(1)='PRINT65';            /* CMS file name            */ E

/*    DEVICE-ID  FAMILY  DEV_TOKEN   OPTIONS       WHICH DEVICE */
CALL DSOPEN(31,    2,     X3816L,  10,PROCOPT_LIST,  1,NAME_LIST );
                                   /* Open queued-printer device*/
```

The call requires some explanation:

- The name of the queued printer device, specified at the point marked **E** in the example, is subsystem-dependent. Briefly, it is a terminal name under CICS, an LTERM name under IMS, a VTAM LUname under TSO, or a file name under CMS (or, of course, a nickname on any of these subsystems).

  On CMS, you can specify a file type and file mode as the second and third elements of the name list. At **E**, the example supplies only the file name, in which case GDDM supplies a default file type of ADMPRINT and file mode of A1.
- The print control option that specifies the number of copies to be printed, at the point marked **B** is probably the option most commonly used.
- The printer may be receiving output from several different users by means of the GDDM Print Utility. It may therefore be convenient for each user's output to be preceded by a **header page**, giving the user's identification and the time the print file was created. The third word of the options group is set to 1 to request a header page (the default). At **A** in the example, a value of 0 is specified, to suppress the use of a header page.
- The fifth, eighth, and ninth words, specified at points marked **C** apply only to FSLOG and FSLOGC output. These are calls that permit alphanumeric logging data to be inserted between the copies of the primary device's output. They are described in “Sending a character string to a printer using call FSLOG” on page 414 and “Sending a character string with control character to printer using call FSLOGC” on page 415.
- The tenth word, at **D** is a rarely-used option used to override the default code-page translation specified for the printer. For more information on the translation of data from different code pages, see “Using GDDM to convert character code pages for international applications” on page 247.

Having opened a queued printer, you can use it like any other primary device. You issue:

```
CALL DSUSE(1,31);           /* Use device 31 (a queued printer) */
                            /* as the primary device            */
```

Subsequent statements (to create a page or a graphics field, for example) refer to the queued printer.

Alternatively, you can use the queued printer as an alternate device by issuing:

```
CALL DSUSE(2,31);           /* Use device 31 (a queued printer) */
                            /* as the alternate device          */
```

You can then issue an FSCOPY, GSCOPY, or DSCOPY command to copy the current page of the primary device to the queued printer.

Family-3 output: Print files for system printers

GDDM supports alphanumerics-only output to IBM 1403, 3211, and 3800 system printers. All GDDM alphanumeric calls can be used with these devices with the exception of those that use symbol-set functions. Field attributes such as blinking cannot be supported for printed alphanumeric output.

This example sends alphanumeric output to a system printer:

```
DCL PROCOPT_LIST(10) FIXED BIN(31);
DCL NAME_LIST(1) CHAR(8);

/*    DEVICE_ID  FAMILY  DEV_TOKEN  PROCESSING OPTIONS    DEVICE*/
CALL DSOPEN(17,    3,   'S3800N8',  0,PROCOPT_LIST,  0,NAME_LIST);   A
CALL DSUSE(1,17);       /* Use system printer as primary device */

CALL ASDFLD(1,3,14,1,25,2);        /* Define alphanumeric field */
CALL ASCPUT(1,25,'SALES REPORT, AUGUST 1992');
     and so on ...
CALL FSFRCE;                   /* Send output to system printer */
```

- The name parameter is subsystem-dependent. Briefly, it is a transient data destination on CICS, an LTERM name on IMS, a SYSOUT DDNAME on TSO, and a filename on CMS (the file type and file mode defaulting to ADMLIST and A1).

  In this example the name of the device was defaulted. Under CMS, this would result in the output being sent to the virtual printer (device “00E” by default).
- At **A** in the example, the device-token, “S3800N8,” specifies a 3800 printer with page size of 80 lines by 85 columns and line spacing of 8 lines to the inch.

Family-4 output: Print files for PostScript and PSF- and CDPF-attached printers

Family-4 output enables your application to send its output to IBM 38xx, 4028, 4224, and 4234 advanced-function printers, to IBM 4250 page printers, and to PostScript printers.

When the current device is a printer that has been opened for family-4 output, your application writes its output to a file when an FSFRCE (or a DSFRCE on TSO) call is issued. This file is then passed to another program that sends it to the printer. IBM provides two such programs; the Print Services Facility (PSF) for output on advanced function printers and the Composed Document Printing Facility (CDPF) for output on the IBM 4250 printer.

The code in Figure 107 on page 405 shows a typical DSOPEN for an IBM 3825, together with its parameter values and the necessary DSUSE call. The only change required to send output to a different page printer would be to select a suitable device token. (For a complete list, see the *GDDM Base Application Programming Reference* book.)

```
DCL PROCOPT(6) FIXED BIN(31);

    PROCOPT(1) = 8;                      /* Usable area size   */ A

    PROCOPT(2) = 80;                     /* 8 inches wide      */ B
    PROCOPT(3) = 100;                    /* 10 inches deep     */ C
    PROCOPT(4) = 0;                      /* 1/10 inch units    */ D

DCL NAMELIST(1) CHAR(8);

    NAMELIST(1) = 'SALES92';             /* File name          */ E

     /* DEVICE_ID   FAMILY   TOKEN    PROC_OPTIONS   NAME_LIST */
CALL DSOPEN (12,      4,    'A3825Q',  6,PROCOPT,    1,NAMELIST); F

CALL DSUSE (1,12);               /* Use 12 as the current      */ G
    :     :                      /* primary device             */
    :     :
    :     :                      /* Calls that put graphics,   */
    :     :                      /* alphanumerics, and image   */
    :     :                      /* onto the GDDM page         */
    :     :
CALL FSFRCE;                                                     H
```

Figure 107. Opening a device for family-4 output

Defining the area of the paper you want the printer to use

On the DSOPEN call, you can specify the dimensions of the paper area on which the GDDM page is to be printed, using processing options 8 (HRIPSIZE). This area is known as the **printer's usable area**.

It is not essential to specify this processing option. If you omit it, GDDM uses the default usable area associated with the device token you supply on the DSOPEN call. If processing option 8 were omitted from the example, the device token, A3825Q, would set the usable area to 8.2 by 10.6 inches.

If you decide to specify the printer's usable area using processing option 8, the dimensions you give it should not exceed those in the device token.

You can find a complete list of device tokens in the *GDDM Base Application Programming Reference* book.

In the example in Figure 107, the usable area for the printer is defined by the statements marked by the letters **A** through **D**. Processing option 8 coded at **A** specifies that the printer's usable area is to be specified. The next fullword specified at **D** gives the area a width of 80 units and the fullword at **E** gives it a depth of 100 units. The last fullword of the group specifies the units: 0 means tenths of an inch, (1 specifies millimeters). Together, these statements specify a usable area for the printer of 8 inches by 10 inches.

Positioning graphics, image, and alphanumeric fields in the usable area

With advanced-function printers, the way you specify the position of graphics, image, and alphanumeric fields depends on the device token you specify on the DSOPEN call. The *GDDM Base Application Programming Reference* book. indicates which family-4 device tokens are cell-based and which are pel-based.

If you specify a cell-based device token on the DSOPEN call for an advanced-function printer, the positioning of graphics, image and alphanumerics on the GDDM page is the same as that on other cell-based devices such as display screens and GDDM-attached printers. The device token defines the dimensions of the device's usable area in cells (for example, 82 x 85 for the A3825Q token). With the GSFLD, ISFLD, and ASDFLD calls, you can specify the size and position of graphics, image, and alphanumeric fields in terms of these dimensions.

If you specify a pel-based device token on the DSOPEN call, the dimensions of the device's usable area default to pels. You must then specify the size and position of graphics and image fields in terms of pels on the GSFLD and ISFLD calls. Alphanumerics are not supported on devices opened using pel-based device tokens.

An alternative way of specifying the size and position of the graphics and image fields when the DSOPEN uses a pel-based token, is to use the FSPCRT call to define a grid of rows and columns. This may be more convenient than using pel dimensions.

Directing the program's output

You specify the name of the file in the name-list parameter of the DSOPEN call, as shown at **F** in the example. At **E** the first element of the name-list array is assigned the name “SALES92.” If there is no file with the specified name, GDDM creates one.

At **G**, the DSUSE call makes the printer the current primary device. This causes the FSFRCE call at **H** to send the output to a print file rather than to the terminal device running the program.

On CMS, you can specify a filetype and filemode as the second and third elements of the name-list. The example supplies only the filename, in which case GDDM supplies a default filetype of ADMIMAGE and filemode of A1.

Information about names under each of the supported subsystems is given in the *GDDM Base Application Programming Reference* book.

Alternatively, you can specify a nickname instead of the name of a print file and have the destination of the output determined by that nickname. This makes the application more flexible because its nicknames can be remapped to those set up by the user and the installation.

Many CMS installations have a special nickname set up called “PRINTER.” If you specify “PRINTER” as the only element of the name-list when you code the DSOPEN for an advanced function print device, the print file can be spooled to PSF without any action on the part of the program or the end user. Routing of the print file can be achieved using the CP SPOOL and CP TAG commands.

Specifying the format to be used for family-4 output

By specifying a different device token or different values on processing option 9 (OFFORMAT), you can create family-4 output to suit different printers and print utilities. Alternatively, you can specify that the data sent to the family-4 print file be an **unformatted data stream**, which is a **bitmap** (suitable for processing by other programs).

The default data stream sent to family-4 print files, where no device token is used, represents the contents of graphics and image fields as uncompressed rastered image.

Depending on the device token specified for the primary device, files with this format can be created for printing via PSF on advanced-function printers or via CDPF on the IBM 4250.

Creating formatted output including GOCA, IOCA, and PTOCA objects

If the advanced-function printers available to end users of the application are supported by a level of PSF later than version 2.0, you can send a data stream to the print file that is much shorter than the default rastered-image data stream and produces output of a higher quality.

When you open a printer for family-4 output, the device token you specify, affects whether:

- Graphics on the current GDDM page can be converted to GOCA graphic drawing orders
- Image data on the current page can be converted into IOCA compressed format objects
- Alphanumeric text can be converted into PTOCA objects

Depending on which of these formats the current advanced-function printer supports, different mixes of GOCA, IOCA, PTOCA, and rastered image are included in the family-4 print file.

Because many advanced-function printers can process data streams that contain these objects, the quality of the printed output is much better than with rastered-image data stream alone and the data stream is much shorter.

You should endeavor to specify a device token that exploits the capabilities of the printer to process mixed-object data streams. The device token information in the *GDDM Base Application Programming Reference* book, tells you the level of each format supported for each cell-based AFPDS printer.

Note: You can use processing option 9 to override the default format specified in the device token.

Reducing program storage when generating rastered image output

GDDM keeps a record of the graphics created by your program in graphics data format (GDF) GDF is an intermediate format between the API and rastered images.

When rastered images are produced from graphics, all graphics lines must be stored as areas, not just vectors, because they can vary in width, and be many pixels wide. This expansion of lines into area definitions can make the GDF

relatively large. To reduce main storage requirements, GDDM, by default, holds the GDF for page printers on external storage, in a **spill file**. You can specify, with processing option 6, that the GDF is to be held in main storage instead. If the fullword following option six specifies the value 1, no spill file is used.

Using a spill file saves main storage, but increases processing time because of the additional external storage I/O.

The rastered image generated for advanced function printers using this format can make the family-4 print file cumbersome. In such a case, you may need to use processing option 7 to write the data to the print-file in horizontal sections or equal size, called **swathes**. This avoids using too much main storage, but processing time tends to increase with the number of swathes used.

Specifying a data stream to suit the purposes of your family-4 output

When you open a device for family-4 printing, you need to specify that the data stream in the formatted print file is one that suits your intended use of the output. To do this use processing option 5 (OFDSTYPE).

Creating an integral document

If you intend your application's output to be a **document** in its own right, a **primary data stream**, you can use the default setting of option 5.

If you create a PostScript document, the PS file may need to be downloaded to be sent to a PostScript printer. The POSTPROC processing option enables you to specify a program that is to perform postprocessing on any family-4 output that is created. You can specify a program that performs the download using this procopt on the DSOPEN call.

If your document is in AFPDS format, you can send it to the printer via PSF. If your document is in CDPDS format, you can send it to the printer via CDPF.

```
DCL PROCOPT(6) FIXED BIN(31);

    PROCOPT(1) = 5;                      /* File type           */
    PROCOPT(2) = 0;                      /* PostScript document */

    PROCOPT(3) = 8;                      /* Size                */
    PROCOPT(4) = 80;                     /* 8 inches wide       */
    PROCOPT(5) = 10;                     /* 10 inches deep      */
    PROCOPT(6) = 0;                      /* 1/10 inch units     */

    PROCOPT(7) = 42;                     /* Initial Image       */
    PROCOPT(8) = 2;                      /* Background to reduce*/
                                         /* amount of image     */

DCL NAMELIST(1) CHAR(8);

    NAMELIST(1) = 'DOCUMENT';            /* File name           */

     /* DEVICE_ID   FAMILY   TOKEN    PROC_OPTIONS   NAME_LIST */
CALL DSOPEN (12,      4,    'PPS2MQ',  6,PROCOPT,    1,NAMELIST);
```

Creating an encapsulated PostScript file, a page segment, or an overlay

If you need to include the formatted output of your program in another document, your application must create the pictures in **secondary data stream**. This secondary data stream can be an encapsulated PostScript (EPS) file, (if you want to include it in a PostScript document) a page segment (PSEG), (if you need to include a picture at a particular point in a document) or it can be an overlay (if you want to include some constant graphics, image, or text data such as a running heading in your document).

The text is prepared by another means, such as the IBM Document Composition Facility. You can then use CDPF or PSF to merge the illustrations and text to create a complete document.

To create an encapsulated PostScript (EPS) file, (if you want to include it in a PostScript document) a page segment, or an overlay for inclusion in another print data stream, you need to specify your choice on processing option 5 of the DSOPEN for the current device.

```
DCL PROCOPT(6) FIXED BIN(31);

    PROCOPT(1) = 5;                     /* File type          */
    PROCOPT(2) = 1;                     /* Page Segment       */

    PROCOPT(3) = 8;                     /* Size               */
    PROCOPT(4) = 60;                    /* 6 inches wide      */
    PROCOPT(5) = 30;                    /* 3 inches deep      */
    PROCOPT(6) = 0;                     /* 1/10 inch units    */

DCL NAMELIST(1) CHAR(8);

    NAMELIST(1) = 'PICTURE';            /* File name          */

     /* DEVICE_ID   FAMILY   TOKEN    PROC_OPTIONS   NAME_LIST */
CALL DSOPEN (12,      4,    'A3820Q',  6,PROCOPT,    1,NAMELIST);
```

If a picture contains text that uses the 4250 fonts (see “Using typographic fonts on a family-4 4250 printer” on page 424) in addition to graphics, you would normally need to create a secondary data stream. This is to avoid exhausting CDPF program storage.

Retrieving family-4 output for the application

Suppose you have an application that creates text documents and uses GDDM to produce graphics to imbed in the documents. Your application can use the GDDM API calls to initialize GDDM, create the graphics, and then format them as family-4 GOCA output.

Prior to GDDM 3.1.1, your application would have to perform an I/O operation to imbed the family-4 output file into the text document. The three new calls FSGETS, FSGET, and FSGETE enable you to bypass this I/O by retrieving each record of the family-4 output into buffers in the application’s storage. The three calls must be issued in the order shown in the following example:

```
RETRIEVE: PROC OPTIONS(MAIN);
/* Example of the use of FSGETx calls within a PL/I program        */
/*                                                                  */
/* Required VM FILENAME definition                                  */
/*                                                                  */
/*   FI IMAGOUT DISK fn ft fm                                       */
/*                                                                  */
/* or TSO DD statement (assumes data set pre-allocated)             */
/*                                                                  */
/*   ALLOC F(IMAGOUT) DA(xxxxxxxx.outfile) REU                      */

/* set up the processing options for Family-4 output                */

DCL PROCOPTS(4) FIXED BIN(31) INIT
             (5,                  /* Option 5, datastream type    */
              1,                  /*    1 = PSEG: 0 = Document    */
              9,                  /* Option 9, output file format */
              3);                 /*    set to GRCIMAGE (GOCA)    */
DCL N FIXED BIN(31) INIT
             (4);                 /* PROCOPT count value          */
DCL NAME(1) CHAR(8) INIT
             ('        ');        /* Set the output device name   */
                                  /* to BLANKS, for a dummy device*/
                                  /* to prevent file output       */
DCL NN FIXED BIN(31) INIT(1);     /* Namelist count value         */
DCL TOKEN CHAR(8) INIT
             ('A3825Q  ');        /* Use a Fam-4 device token     */


                                  /* GSLOAD parameters            */
DCL OARR(2) FIXED BIN(31) INIT    /* Options array                */
             (1,                  /* 1 = Beginning segment no     */
              2);                 /* 2 = Fill picture space       */
DCL SEGCT FIXED BIN(31) INIT(0);  /* Segment count                */
DCL DESC CHAR(1);                 /* Dummy field for description  */

DCL B CHAR(8206) INIT(' ');       /* Buffer for AFPDS data        */
DCL REC_OUT CHAR(8206) VARYING;   /* Buffer for output records    */
DCL L FIXED BIN(31) INIT(0);      /* Buffer data size             */
DCL BDT CHAR(3) INIT('Lyy');      /* Begin Document Order,        */
                                  /* X'D3A8A8'                    */


DCL 1 EDT_REC,                    /* End Document Record          */
      2 FIVEA CHAR(1) INIT('!'),                        /*X'5A'    */
      2 LENGTH FIXED BIN(15) INIT(16),                  /*= 16     */
      2 EDT CHAR(3) INIT('Lzy'),                        /*X'D3A9A8'*/
      2 FLAG BIT(24) INIT('000000000000000000000000'B), /*X'000000'*/
      2 DOC_NAME CHAR(8);

DCL IMAGOUT FILE RECORD OUTPUT ENV(V RECSIZE(8210));

   CALL FSINIT;
                                  /* open the Family-4 device     */
   CALL DSOPEN(42,4,TOKEN,N,PROCOPTS,NN,NAME);
   CALL DSUSE(1,42);              /* use this device              */
   OPEN FILE(IMAGOUT);            /* Open output file             */
```

```
/* At this point your program will issue the GDDM                 */
/* calls required to create the page for the PSEG. In             */
/* this example the supplied GDF sample file is loaded.           */

   CALL GSLOAD('ERXMODEL',2,OARR,SEGCT,0,DESC);

   CALL FSGETS;                    /* Use instead of FSFRCE to     */
                                   /* create the family-4 output   */
   DO UNTIL(L = 0);                /* Loop getting the family-4    */
   CALL FSGET(B,L);                /* records into variable B      */
   IF L > 0 THEN
     DO;
                  /* At this point your program will process each  */
                  /* record passed by the fsget call.  In this     */
                  /* example the records are written to a CMS file.*/
       REC_OUT = SUBSTR(B,1,L);
                  /* Test for BDT rec: save document name for EDT. */
       IF PROCOPTS(2) = 0 THEN DO;
          IF SUBSTR(REC_OUT,4,3) = BDT THEN DO;
             DOC_NAME = SUBSTR(REC_OUT,10,8);
          END;
       END;
       WRITE FILE(IMAGOUT) FROM (REC_OUT);
     END;

     ELSE
       DO;                         /* FSGETS failed, so free any   */
         CALL FSGETE;              /* storage acquired for Family-4*/
       END;                        /* buffers by GDDM              */
   END;

   CALL DSDROP(1,42);
   CALL DSCLS(42,1);

   CALL FSTERM;

/* At this point if Document output was specified, the            */
/* program must write an AFPDS End Document record.               */
   IF PROCOPTS(2) = 0 THEN DO;
       WRITE FILE(IMAGOUT) FROM (EDT_REC);
   END;

/* Close the output file.                                         */
   CLOSE FILE(IMAGOUT);

   %INCLUDE ADMUPIND;
   %INCLUDE ADMUPINF;
   %INCLUDE ADMUPING;

   END RETRIEVE;
```

This example illustrates how to extract either primary or secondary AFPDS format data and place it in an application buffer. The output is simply written to a CMS file. In an application, the buffer data would be written inline in a text or other document type generated elsewhere in the application program.

Using a printer as an alternate device

GDDM enables you to send copies of the primary device's output to an alternate device. You can use a printer as an alternate device and thus, obtain a hard copy of the output to a display terminal. A program for doing this is shown in “Example: Copying screen output to a printer” on page 416.

The DSOPEN calls described in the earlier sections of this section apply equally to alternate devices and primary ones.

The print control processing options described in “Family-2 output: Print files for GDDM queued printers” on page 402 can be applied to any printer that is being used as an alternate device. Their main use is to set the margins around the printed area. The number-of-copies option (the fourth one in the list) is honored for family-2 printing only.

A device opened for family-4 output can only be used as alternate device if a cell-based device token is specified on the DSOPEN call.

After opening, you make the printer the alternate device using a DSUSE call:

```
CALL DSUSE(2,31);           /* Use device 31                    */
                            /* as an alternate device, to       */
                            /* receive copies of the primary    */
                            /* device's output.                 */
```

You can have only one alternate device in use at a time. A DSUSE call for a new alternate device implicitly drops the alternate device that was in use before the DSUSE.

Five calls, DSCOPY, FSCOPY, GSCOPY, FSLOG, and FSLOGC, send output to the alternate device. They are described below.

Copying a transformed picture to a printer, using call DSCOPY

The DSCOPY call enables your application to copy any graphics or image data on the current GDDM page to the current alternate device, and control the size, position and orientation of the output. You can choose to copy the graphics data or the image data or both to the alternate device. DSCOPY doesn’t copy alphanumeric fields to the alternate device. Here is an example of the call:

```
DCL OPTIONS(3) FIXED BIN(31)   INIT( 2, 0, 1);

/*          WIDTH   DEPTH  HORZ_OFFSET VERT_OFFSET   COUNT  OPT_ARRAY */
CALL DSCOPY( 250,    250,     15,         10,          3,    OPTIONS   );
```

This call copies both the graphics and image data on the current page to an alternate device, such as a printer. It increases both the width and depth of the picture by 150%. The values specified for the horizontal and vertical offsets would usually mean that the top left-hand corner of the picture is positioned 15% of the way across the printer page (from the left) and 10% of the way down the printer page. However, because the third element of the options array specifies that the picture be rotated through 90°, the horizontal and vertical offsets are interchanged.

Copying a page to a printer using call FSCOPY

This call copies the current page to the current alternate device. If the alternate device is opened for family-1, -2, or -4 output, the alphanumerics, graphics, and image data are copied. If the alternate device is a family-3 printer, only the alphanumerics are copied.

Family-1, -2, and -4 printing is subject to the considerations outlined in “Mixed graphics and alphanumerics” on page 422. Unsatisfactory results may occur if your application tries to copy a mixture of alphanumeric and graphics data. This is because the relative positions of the two types of data are subject to change across different devices. See also “Printing images” on page 366.

The principal use of FSCOPY is to copy pages of alphanumeric data. The format of the call is simply:

```
CALL FSCOPY;            /* Send copy of page (alpha & graphics) */
                        /* to the printer                       */
```

These factors apply to FSCOPY:

- The size of the printed-copy page is the same (in printer hardware cells) as that of the current page (in hardware cells of the primary device).
- By default, the aspect ratio of the graphics is maintained. The aspect ratio of the page is not, however, as the aspect ratio of a single cell varies from device to device. Therefore the graphics occupy a different portion of the page (compared with that on the primary device), and consequently are positioned differently in relation to any alphanumeric fields. More information is given in “Mixed graphics and alphanumerics” on page 422.
- Alphanumeric field and character attributes are retained on the printed copy whenever possible. Underscore is retained, for example, but blinking cannot be.
- Wherever symbol sets are used to create the original picture, they are used again to create the copy. This applies equally to pattern sets and marker sets. If a substitution character was used on the original symbol-set load (see “Loading symbol sets for alphanumeric text” on page 238), GDDM loads the appropriate version of that symbol set for the printer.
- If the original picture uses proportionally spaced symbols, you should ensure that either:

  The same symbol set is available when printing takes place. This applies particularly when copying to family-2. The symbol sets for the printer are accessed when the print file is processed, not during execution of your program.

  or

  Any different symbol set used by your program for printing (by using a substitution character, for instance) has the same spacing for all characters as the set used for the original display.

  If these conditions are not met, the length of the printed string is different from that of the original.
- Graphics primitives outside segments are not copied.

You can obtain multiple copies of a page by issuing multiple FSCOPY calls. On a family-2 device you can, instead, use the number-of-copies parameter of the print-control processing option.

Copying graphics to a printer using call GSCOPY

This call copies the contents of the current page's picture space to the current alternate device if it is family-1 or -2. It does not copy alphanumerics or image data. It permits you to specify how large the copy should be. This is the format of the call:

```
CALL GSCOPY(60,120);     /* Copy graphics to queued printer, using a */
                         /* printer page-size of 60 rows by 120 cols */
```

By default, the aspect ratio of the graphics is maintained. If you draw a square picture on the primary device, for example, and then issue a CALL GSCOPY(5,120), you do **not** get an elongated version of the picture stretching right across the page. Instead, you get a square picture, 5 rows deep, centered on the boundary of columns 60 and 61. In some cases it may be more important to fill the area specified in the GSCOPY than to preserve the aspect ratio of the graphics. This call makes that happen:

```
CALL GSARCC(1);          /* Do not preserve aspect ratio */
```

GSCOPY treats symbol sets in the same way as FSCOPY.

Graphics primitives outside segments are not copied.

You can obtain multiple copies of the graphics on a page by issuing multiple GSCOPY calls. On a family-2 device you can, instead, use the number-of-copies parameter of the print control processing option.

Sending a character string to a printer using call FSLOG

This call enables you to send character strings to the printer in between FSCOPY or GSCOPY calls, or in between both.

The first FSLOG call after a copy call moves the printer to a new page. Batches of FSLOG data appear on the same page. This is the format of the call:

```
CALL FSLOG(47,'NEXT PAGE SHOWS ILLUSTRATION FOR COMPANY REPORT');
```

The first parameter gives the length of the text. The second gives the text itself.

The maximum depth and width of the log data is determined by the processing options you specify when you open the printer (see “Family-2 output: Print files for GDDM queued printers” on page 402).

Sending a character string with control character to printer using call FSLOGC

This call is similar to FSLOG, but GDDM interprets the first character in the string as a carriage-control character:

```
CALL FSLOGC(14,'-END OF REPORT'); /*Skip 3 lines before printing*/
```

The first parameter of the call is the length of the character string **including** the carriage-control character. The valid control characters are shown in Table 6. The hexadecimal codes are the same as the CTLASA and CTL360 codes.

FSLOGC has the same purposes as FSLOG, and some additional ones, including:

- Printing existing sequential files that contain carriage-control characters.
- Printing alphanumeric text layouts when the facilities offered by the more complicated alphanumerics API are not required.

Table 6. Carriage-control codes for FSLOGC

Spacing action	**Relation between spacing action and printing**		
	Spacing before printing	Spacing after printing	Spacing without printing
Space 1 line	blank	X'09'	X'0B'
Space 2 line	0	X'11'	X'13'
Space 3 line	–	X'19'	X'1B'
Skip to new page	1	X'89'	X'8B'
None (print unspaced)	+	X'01'	X'03'

Note: Bold printing of characters is not possible using the “+” carriage control character. For more information on the processing of overstrike characters, see the *GDDM Base Application Programming Reference* book.

Example: Copying screen output to a printer

The programming example in Figure 108 illustrates the use of a primary device and two queued printers:

```
GUIDE: PROC OPTIONS(MAIN);
DCL PROCOPT_LIST(10) FIXED BIN(31); /* Processing options list  */
DCL NAME_LIST(1) CHAR(8);           /* Device-control name list */

CALL FSINIT;

CALL GSSEG(0);                  /* Open graphics segment for    */
                                /* default page of user-console */
CALL GSCOL(2);                  /* Start drawing map of deer    */
CALL GSPLNE(116,XA1,YA1);       /* Estate                       */
    ....
CALL GSCHAR(45.0,62.0,30,'Wishing well (XVIIIth century)');
CALL ASREAD(TYPE,MODE,COUNT);      /* Send map to user-console  */ A

CALL FSPCRT(2,0,0,1);              /* Open a 2nd page           */
CALL ASDFLD(7,1,15,1,50,2);        /* Define alpha field        */
CALL ASDFLD(8,4,1,16,68,2);        /* Define alpha field        */
CALL ASCPUT(7,50,'This pamphlet describes the Hiltingbury Deer Park.');
CALL ASCPUT(8,1088,
'    In 1675, the 4th Duke of Exeter married his second cousin, a    '||
'famous society beauty named Elizabeth Powys. Their first son died in'||
    ....
    ....
'not forget to visit the recently restored summer house by the lake. ');

CALL ASREAD(TYPE,MODE,COUNT);    /* Send guide text to console  */

PROCOPT_LIST(1)=4;               /* Print control option code   */
PROCOPT_LIST(2)=2;               /* No. of fullwords following  */
                                 /* in  this  option  group     */
PROCOPT_LIST(3)=0;               /* Do not print header page    */
PROCOPT_LIST(4)=50;              /* Number of copies required   */

NAME_LIST(1)='GUIDE';            /* CMS file name               */

/*    DEVICE-ID  FAMILY  DEV_TOKEN   OPTIONS       WHICH DEVICE */
CALL DSOPEN(11,    2,      '*',    4,PROCOPT_LIST,  1,NAME_LIST );
                         /* Open queued-printer device to print */
                         /* 50 copies of guide (text + map)     */

PROCOPT_LIST(4)=35;                /* Number of copies required */
NAME_LIST(1)='ONLYMAP';            /* CMS file name             */
```

Figure 108 (Part 1 of 2). Copying to printers

```
CALL DSOPEN(12,    2,      '*',    4,PROCOPT_LIST,  1,NAME_LIST );
                        /* Open queued-printer device to print */
                        /* 35 enlarged copies of just the map  */

CALL DSUSE(2,11);       /* Use guide queued printer first      */
CALL FSCOPY;            /* Copy alphanumeric text from page 2  */
CALL FSPSEL(0);         /* Reselect default page (with map)    */
CALL GSCOPY(40,80);     /* Copy DEERPARK map to 40 by 80 area  */
CALL DSCLS(11,1);       /* Close queued printer                */ B

CALL DSUSE(2,12);       /* Use only map queued printer now     */ C

CALL GSCOPY(70,120);    /* Copy DEERPARK map to 70 by 120 area */
CALL DSCLS(12,1);       /* Close queued printer                */

CALL FSTERM;            /* Terminate  GDDM                     */
%INCLUDE(ADMUPINA);     /*Include GDDM entry-point declarations*/
%INCLUDE(ADMUPIND);
%INCLUDE(ADMUPINF);
%INCLUDE(ADMUPING);
END GUIDE;
```

Figure 108 (Part 2 of 2). Copying to printers

Notes:

1. **Copy operates on the current page contents**. The copy part of the program would work equally well without the ASREAD **A** to the primary device. All copy commands reflect the current page contents, whether or not they have been transmitted to the primary device.
2. **Suppressing print-file creation**. The second DSCLS parameter, 1, in statement **B**, indicates that the creation of the print file should proceed. In other circumstances a program might detect an error condition and need to cancel the print-file creation. In that case a parameter setting of 0 would be made.

   If a queued printer is not explicitly closed with a DSCLS, GDDM closes it (and proceeds with creating the print file) when it executes the FSTERM.
3. **DSCLS implies a DSDROP**. Normally the DSUSE **C** would be preceded by a DSDROP(2,11) to drop the previous alternate device. It is not necessary here because the DSCLS of device 11 drops the device.

Under CMS, this program creates two print files on the user's A-disk. The user would normally invoke the GDDM Print Utility to print the two files. For other subsystems the alternate device's DSOPEN would be slightly different and the print files would be sent straight to the print utility.

Printing GDDM family-2 print files

If the DSOPEN statement for a printer specifies device-family-2, the GDDM output calls create files to be processed by the GDDM Print Utility. This section gives an overview of the GDDM Print Utility, which is described in more detail in the *GDDM Base Application Programming Reference* book.

On subsystems other than CMS, there may be several printers under the control of the print utility; the name you provide in the name-list parameter of DSOPEN determines which printer is to be used. The queued printer output of several different users may appear on one printer; the sets of output are separated by header pages (unless suppressed on the DSOPEN).

There is a different version of the utility for each subsystem.

On CICS, IMS, and TSO, print files are sent to the print utility when the program issues a DSCLS for the queued printer device.

Under CMS, you can arrange a similar facility by spooling the print file to RSCS. This requires two sets of DSOPEN processing options, one to invoke the print utility, and the other to spool the print utility's output to RSCS. The nicknames facility is generally the simplest way of specifying these options. Here is a suitable DSOPEN call:

```
NAME_LIST(1) = 'RSCSPRT1';
        /* ID  DEV-FAMILY  DEV-TOKEN  PROCESSING OPTIONS  DEV-NAME*/
CALL DSOPEN(7,     2,        '*',     0,PROCOPT_LIST,   1,NAME_LIST);
```

and here are the required nickname statements to send it to a printer called REMPRT7:

```
ADMMNICK FAM=2,NAME=RSCSPRT1,DEVTOK=X4224SE,
         PROCOPT=((INVKOPUV,YES))
ADMMNICK FAM=1,NAME=RSCSPRT1,TONAME=PUNCH,DEVTOK=X4224SE,
         PROCOPT=((CPSPOOL,TO,RSCS),
                  (CPTAG,REMPRT7,PRT=GRAF))
```

The INVKOPUV processing option on the first nickname statement automatically invokes the function of the CMS version of the print utility, ADMOPUV. The second statement applies to the output from the print utility. It is similar to the one described in “Family-1 output: GDDM directly attached printers” on page 401. The TONAME parameter sends the output to the virtual punch. The two processing options spool the punch file to RSCS and tag it.

Under CMS you can, instead, attach a printer to your own VM machine (using the CP MOUNT command) and invoke the print utility yourself. This is the statement required:

```
ADMOPUV fname ON 063 (DEV device-token
```

- fname is the name of the print file.
- ON 063 gives the virtual address of the printer. This option may be omitted, in which case a default address of 061 is used.
- (DEV device-token supplies a device token for the printer. It is required only when the printer is attached to the PUNCH address.

Another option under CMS is to create an EXEC procedure to process the file automatically, by, for instance, transferring it to another virtual machine for printing. If you name the procedure ADMQPOST EXEC, GDDM invokes it whenever your program completes the creation of a print file. For more information about this technique, see the *GDDM System Customization and Administration* book.

If, under TSO, you want to use a particular device token at print time and to associate it with a single LU, then you need to use two nicknames. The first nickname is used when you are creating the print file and takes this form:

```
ADMMNICK FAM=2,NAME=NICKNAME,PROCOPT=((STAGE2ID,LUNAME)),DEVTOK=TOKEN
```

You may direct the print to the nickname from the print/plot panel of the ICU or from a DSOPEN namelist parameter. The STAGE2ID procopt places the LUNAME in the print file for the print utility to use. The second nickname, which is resolved at print time, looks like this:

```
ADMMNICK FAM=1,NAME=LUNAME,DEVTOK=TOKEN
```

The print utility issues a DSOPEN call and the STAGE2ID, which was assigned the value of LUNAME by the first nickname, is used as the name-list. Since this matches the NAME parameter in the second nickname, the device token specified in DEVTOK=TOKEN is used.

Printing composite documents

Composite documents are ones which comprise both text and pictures. The pictures can be computer graphics or images scanned from paper originals.

Composite documents can be stored in one of two formats. Those created using the IBM DisplayWrite licensed program are stored in files of Composite Document Presentation Data Stream (CDPDS) format. Those created using the SCRIPT/VS formatter of the IBM Document Composition Facility (DCF) are stored in Advanced Function Presentation Data Stream (AFPDS) format.

The primary function of the Composite Document Print Utility (CDPU) is to print composite documents. The CDPU can also be used to display them on the user's screen but composite documents cannot be plotted.

You can print composite documents by invoking the utility from within an application program, using the CDPU call.

A ready-made program to print composite documents is supplied with GDDM. It is called ADM4CDUx (where x is subsystem-dependent).

If you issue the CDPU call with a display, rather than a printer as the primary device, the Composite Document Print Utility displays the composite document on the screen. A REXX exec that displays composite documents (ADMUBCDV) is also supplied with GDDM on the CMS subsystem. A CLIST that performs the same task (AMUBCDT) is supplied with GDDM on the TSO subsystem.

Information about coding the CDPU call is contained in the *GDDM Base Application Programming Reference* book.

Example: Program to print a composite document

The program shown creates an AFPDS file from a CDPDS file. By omitting the DSOPEN and DSUSE calls, you can use the program to view a document on the CICS, TSO, or CMS subsystems.

```
SAMPLE: PROCEDURE OPTIONS(MAIN);

    /* DECLARE GDDM ENTRY POINTS */
 %INCLUDE ADMUPIND;  /* NAMES BEGINNING D... */
 %INCLUDE ADMUPINF;  /* NAMES BEGINNING F... */
 %INCLUDE ADMUPINK;  /* NAMES BEGINNING CD.. */

    /* OTHER DECLARATIONS  */
 DCL DEVID  FIXED BIN(31) INIT(11);
 DCL FAMILY FIXED BIN(31) INIT(4);
 DCL DEVTOK CHAR(8) INIT('A4');
 DCL IN(1)  CHAR(8) INIT('CDPIN');
 DCL OUT(1) CHAR(8) INIT('CDPOUT');
 DCL NONE(1) FIXED BIN(31);   /* DUMMY ARRAY */

    /* INITIALIZE GDDM  */
 CALL FSINIT;

    /* OPEN THE DEVICE  */
 CALL DSOPEN( DEVID, FAMILY, DEVTOK, 0, NONE, 1, OUT);
 CALL DSUSE( 1, DEVID);

    /* PRINT THE DOCUMENT */
 CALL CDPU( 1, IN, 0, NONE);

    /* TERMINATE GDDM */
 CALL FSTERM;
 END SAMPLE;
```

Controlling how end users browse composite documents

If an application calls the CDPU with the view control parameter set to a nonzero value, the application can control how the document is browsed. The CDPU creates a GDDM page containing the specified document page, but does no input or output. The application must issue its own ASREAD (or other input/output call) and interpret the returned values. Additionally, the application can:

- Define a graphics field in which to show the document page. The default is a field covering the whole screen.
- Display instructions to the end user.
- Test for requests for document pages beyond the document end.

Specifying the device for CDPU output

The output from the CDPU goes to the primary device specified by the DSOPEN and DSUSE calls. End users' installations can use nickname statements to modify the DSOPEN specification without changing the application.

The device can be any printer or display opened for family-1, -2, or -4 output.

When the CDPU call is executed, the CDPU checks the type of primary device that the application program has opened, and generates the appropriate data stream.

The default primary device is the terminal, which is why the CDPU displays the document at the terminal if the DSOPEN and DSUSE calls are omitted.

If the device is a printer opened for family-2 of family-4 output, an intermediate print file is created. Its name is taken from the **name-list** parameter of the DSOPEN call. If a file with the same name already exists, it is deleted without warning.

Printing non-GDDM sequential files

Under CMS and TSO, you can use GDDM utilities to print ordinary sequential files on print devices that have been opened for GDDM family-1 output, such as the 3028 and the 4224. (These files can also be routed to devices opened fro family-4 output.)

Under CMS, you would use the GDDM Print Utility for this purpose. This is the command:

```
ADMOPUV fname ftype fmode ON 063 (NOCC DEV device-token
```

It has the same parameters as previously described, with the addition of the NOCC option. Both `ftype` and `fmode` can be omitted, as can the options delimited by the bracket, except that the DEV device-token option is required if the printer address following the ON keyword is PUNCH. If `ftype` is not specified, ADMPRINT is assumed, and if `fmode` is not specified, "*" is assumed, with the usual CMS meaning. On CMS, family-4 output can be routed to PSF directly, see "Directing the program's output" on page 406.

The NOCC option means that the records do not have carriage-control characters in the first byte; the default assumption is that they do. In the default case, the first byte of each record is interpreted according to Table 6 on page 415. You can specify a device token (see "Using DSOPEN to tell GDDM about a device you intend to use" on page 371) after the DEV option; the default is "*".

Under TSO, you would use the GDDM Sequential File Print Program. Its program name is ADMOPRT, and it is invoked like this:

```
CALL 'dsname(ADMOPRT)' 'filename ON printername (NOCC'
```

The dsname is the data set in which ADMOPRT has been installed; filename is the ddname of the data set to be printed or, if there is no such ddname, the data-set name, and printername is the device on which it is to be printed. The (NOCC option means that the file is to be printed on the assumption that it contains no carriage-control characters. If you omit this option, GDDM interprets the first byte according to Table 6 on page 415.

ADMOPRT converts the sequential file into a GDDM print file, which it queues for ADMOPUT, the TSO version of the GDDM Print Utility. This utility must be run to produce the output; the *GDDM Base Application Programming Reference* book describes how to do this.

Re-rastering when copying

For primary devices that use hardware cells to display graphics, such as the IBM 3279 terminal, GDDM creates the picture by rastering the graphics requests in your program. In other words, it converts the graphics primitives into programmed symbols that are subsequently loaded into the PS stores of the primary device.

When the same picture is copied to an alternate device, GDDM cannot simply copy the same programmed symbols to the new device, because the new device may have cells of a different size. For instance, the 3279 display unit has cells of 9 pixels by 12, whereas a typical alternate device, the 3287 printer, has cells of 10 by 8. All the graphics, therefore, have to be re-rastered.

Every call such as GSLINE and GSCHAR must be reprocessed to obtain a copy of the picture on the alternate device. That is why access is required to the symbol sets involved (or their equivalents, if substitution characters were used).

The re-rastering is performed by the GDDM Print Utility. The print file that is passed to the utility contains the various primitives expressed in Graphics Data Format (GDF) (which is introduced in “Modifying graphics pictures that have been loaded into your program” on page 193).

Mixed graphics and alphanumerics

Even if a graphics program is eventually to run against a printer, you may find it convenient to run against a display device while you are developing the program. In that case you should be aware that the appearance of a picture may vary considerably (and sometimes unexpectedly) from one device to another.

The most tricky situation arises when the output contains both graphics and alphanumerics. The relative positioning of alphanumerics and graphics may change.

When the printer is the primary device, these are the factors to bear in mind:

- Whether or not the graphics field is explicitly defined, its aspect ratio changes from device to device. A graphics field of 32 rows by 80 columns, for example, gives a different aspect ratio on a printer to the one produced on a 3472-G display unit.
- If the aspect ratio is explicitly set (by calling GSPS), the **position** of the picture space within the graphics field varies from device to device. This is no problem unless alphanumeric fields are present. The relative position of alphanumerics and graphics is then affected.
- The default page size varies from device to device. On a 3472-G it is 32 by 80; on a printer it is 80 by 132. The output from programs that use the default page-size differs therefore from device to device.
- Graphics primitives are positioned using window coordinates applied to the picture space (or the viewport, if specified); alphanumeric fields are positioned by hardware cell position. When you send the same picture to two different types of device in succession, the relative positioning of alphanumeric and graphic data is bound to change, unless you take these special precautions:
  - Specify the graphics field explicitly for each output device.
  - Allow the picture space and viewport to take the default values. Do not issue any GSPS or GSVIEW calls.

  If you do this, the alphanumerics and graphics maintain their relative positioning. The aspect ratio of the graphics does change, however. It is not possible to maintain both factors.

- If the program uses mode-3 graphics text rather than alphanumerics, there is no problem with relative positioning, when the character box (the character size) is explicitly set.

  For example, you may have a routine of graphics calls that draws a geographical map. These calls can be a combination of GSLINEs, GSAREAs, and mode-3 GSCHARs. Assume that the map has been produced and tested using a 3472-G display. When you are satisfied with the output, you may decide to run the same routine against a printer device, setting a much larger page-size. When the character box is allowed to default in both cases (to the hardware cell size), the text is too small relative to the graphics when run against the printer. If the character box is explicitly set (in terms of window coordinates, as usual), the same proportion can be maintained.

When the printer is an alternate device, you can choose between keeping the aspect ratio of your graphics the same as on the primary device, or preserving the relative positions of the graphics and the alphanumerics, using a GSARCC call:

```
CALL GSARCC(1); /* Preserve graphics/alphanumerics relationship */
```

A parameter value of 1 means that the relative positions of the alphanumeric fields and the graphics are preserved, but the aspect ratio of the graphics changes. A value of 0 (the default) means the reverse. The call must be executed for each page being copied before the FSCOPY call.

Using loadable symbol sets on family-3 3800 printer

The 3800 printer permits the loading of symbol sets. This loading is controlled by JCL when the printer is initiated. The symbol sets involved have no connection with GDDM symbol sets; they are associated with the hardware. It is possible to load up to 4 such hardware symbol sets (they are numbered 0, 1, 2, and 3).

These symbol sets may not be loaded by use of GDDM calls. They have predefined values (0, 1, 2, 3) within GDDM, and it is the user's responsibility to ensure that the printer is loaded with appropriate fonts corresponding to these numbers.

Access to these symbol sets is provided by using the ASFPSS and ASCSS calls (see “Specifying a symbol set for use in an alphanumeric field” on page 239).

These are typical calls:

```
CALL ASFPSS(22,3); /* Field 22 is displayed in the font of the  */
                   /* fourth loadable 3800 symbol set           */
DCL CHAR1 CHAR(1);
UNSPEC(CHAR1)='00000010'B;      /* Put X'02' into char variable */

CALL ASCSS(17,4,CHAR1||'  '||CHAR1);/* 1st and 4th characters of*/
                                    /* field 17 use the third   */
                                    /* loadable symbol set      */
```

Note the following points:

- The symbol-set parameter of the ASFPSS call can be set to 0, 1, 2, or 3 to indicate usage of the 1st, 2nd, 3rd, or 4th loadable symbol set respectively.

The last parameter of ASCSS can specify hexadecimal values of “01,” “02,” or “03” to access the second, third, or fourth fonts.

- As with all character-attribute calls, ASCSS requires its attributes as a string of 1-byte character values.
- A value of “ ” (blank) in ASCSS means “inherit the field attribute set by ASFPSS.”

Using typographic fonts on a family-4 4250 printer

You can use the 4250 printer's fonts for mode-1 and -2 graphics text, as a high-quality alternative to GDDM image and vector symbol sets.

You access these fonts by specifying 5 as the symbol-set type in a GSLSS call. The second parameter of the GSLSS call is the name of the file holding the 4250 font. The symbol set identifier in the third parameter must be different from any type-1 symbol set already loaded, and also from any other type-5 symbol set.

Then you set the character mode to either 1 or 2 with a GSCM call (or allow it to default to 1). Finally, you write the text with GSCHAR or GSCHAP calls.

Mode-1 and -2 differ in the amount of control you have over the appearance of the text, as explained in “Affecting the appearance of graphics text” on page 58. They also differ in character and line spacing. In mode-1, the spacings follow the width and depth definitions contained within the fonts. This means the text is proportionally spaced along the lines. In mode-2, the characters are spaced at the width and depth of the current character box, which means the spacing along the lines is constant.

Here is an example that uses two different 4250 fonts:

```
   .
   .
   .
       /* TYPE     NAME   IDENTIFIER */
CALL GSLSS( 5, 'AFT10025',    98 );  /* Load 4250 font AFT10025 */
CALL GSLSS( 5, 'AFT06008',    99 );  /* Load 4250 font AFT06008 */
   .
CALL GSCM(1);                        /* Set mode to 1           */
CALL GSCS(98);                /* Make AFT10025 current (note 2) */
CALL GSCHAR(1.0,10.0,42,'Example of 14 point Univers Bold Condensed');
CALL GSCS(99);                /* Make AFT06008 current (note 3) */
CALL GSCHAR(1.0,15.0,44,'Example of 10 point Monotype Times New Roman');
CALL FSFRCE;                  /* Send text to 4250 image file   */
   .
   .
   .
```

Notes:

1. As supplied on your IBM system, as opposed to by GDDM, fonts have file names of the form AFTxxxxx, but these names can be changed by a user after installation. Under CMS, the font files have a file type of FONT4250.
2. Font AFT10025 is 14 point Univers Bold Condensed.
3. Font AFT06008 is 10 point Monotype Times New Roman.

The fonts are illustrated in *IBM 4250 printer type font catalog*. You can get a listing of the fonts available on your system and their AFTxxxxx numbers by running an IBM program, the DCF Font Library Index Program (see the *Document Composition Facility: Script/VS Language Reference* manual.

When the Composed Document Printing Facility (CDPF) prints a primary data stream containing both typographic font data and rastered graphics data (either directly, or indirectly within any included secondary data streams), it issues a warning message (BFU629W) stating that structured fields were identified. This warning does not affect the appearance of the output, which is correct.

Code-page support for 4250 output

For most types of application, you need not be concerned with this topic. However, you may need to understand it if you use 4250 fonts to print a number of different national languages, or to print special symbols such as scientific ones or those used in APL.

A **code page** associates a set of symbols with a set of two-digit hexadecimal numbers (code points), each symbol being represented by a number. Code pages are variations on the standard set of EBCDIC associations. Most are designed for printing particular national languages. In most code pages, the basic alphabet and the numerals have the same code points as in EBCDIC – `X'C1'`, `X'C2'`, `X'C3'` for A, B, C, and `X'F1'`, `X'F2'`, `X'F3'` for 1, 2, 3, and so on.

The variations generally occur with the special symbols. For instance, the code page designed for U.K. English has `X'4B'` as the code point for the pound sign; in the U.S. and Canada English set `X'4B'` is the dollar sign; and in the Brazil set it is a C with a cedilla.

Code pages have a similar naming scheme to fonts. They have file names of the form AFTCxxxx, and, under CMS, a file type of FONT4250. The file names can be varied after installation.

You make a code page current by executing a GSCPG call:

```
CALL GSCPG(5,'AFTC0385');    /*AFTC0385 (Canada French) current codepage*/
```

The first parameter is the type of code page: it must be 5.

Ordering of font and code page calls

When a 4250 font is loaded using a GSLSS call, it is associated with the 4250 code page that is current. Therefore, to associate a particular code page with a particular font, you must issue the GSCPG call before the GSLSS that loads the font.

The symbols for all code points in every code page are illustrated in *IBM 4250 printer type font catalog*.

The GDDM default code page is AFTC0395 (U.S. and Canada English). Your application may override this (or any code page specified by the installation), and use a different code page for the conversion of code points. Instructions for doing this are given in “Converting code pages using API calls in the program” on page 250.

Example: Program using 4250 fonts

An example of how to use 4250 fonts is given in Figure 109, with the output in Figure 110 on page 427.

```
FONT: PROC OPTIONS(MAIN);
DCL PLIST(8) FIXED BIN(31);
DCL NLIST(3) CHAR(8);
DCL XA(5)    FLOAT DEC(6) INIT (1.0,99.0,99.0,1.0,0.0);
DCL YA(5)    FLOAT DEC(6) INIT (99.0,99.0,1.0,1.0,0.0);
CALL FSINIT;                   /* Initialize GDDM              */
PLIST(1) = 9;
PLIST(2) = 1;                  /* Formatted output             */
PLIST(3) = 5;
PLIST(4) = 0;                  /* Primary data stream          */
PLIST(5) = 8;
PLIST(6) = 60;                 /* Width                        */
PLIST(7) = 40;                 /* Depth                        */
PLIST(8) = 0;                  /* In tenths of inches          */
NLIST(1) = 'FONT';
NLIST(2) = 'SAMPLE';           /* Output file-id               */
NLIST(3) = 'A1';

CALL DSOPEN(11,4,'IMG600X',8,PLIST,3,NLIST);
CALL DSUSE (1,11);                  /* Make 4250 primary device */
CALL GSUWIN(0.0,100.0,0.0,100.0);   /* Define uniform window    */
CALL GSCPG (5,'AFTC0394');     /* Select U.K.-English code page */
CALL GSLSS (5,'AFT08004',77);  /* Load Helvetica 12pt MED      */
CALL GSCS (77);
CALL GSCM (2);                 /* Spacing controlled by GSCB   */
CALL GSCB (5.0,10.0);
CALL GSCHAR(1.0,80.0,16,'Helvetica - 12pt');

CALL GSMOVE(30.0,50.0);
CALL GSARC (50.0,50.0,360.0);  /* Include some ordinary graphics*/
CALL GSMOVE(1.0,1.0);
CALL GSPLNE(4,XA,YA);
CALL GSLSS (5,'AFT08008',66);  /* Load Times New Roman 12pt MED */
CALL GSCS (66);
CALL GSCM (1);                 /* Spacing controlled by font   */
CALL GSQTB (33,'Times New Roman - 12pt - centered',3,XA,YA);
CALL GSCHAR((100-XA(3))/2,20.0,33,'Times New Roman - 12pt - centered');
CALL FSFRCE;                   /* Generate output file         */
CALL FSTERM;                   /* Terminate GDDM               */
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END;
```

Figure 109. Example of using 4250 fonts

Figure 110. Output of 4250 font example

Color masters for publications

You can use the family-4 (page) printers to create color-separation masters for printing text and graphics in full color in publications. There is normally one monochrome master for each of three subtractive primary colors (yellow, magenta, and cyan), and a fourth for black. Each master records where ink of the color that it represents has to be deposited, as illustrated by Figure 111 on page 428. The images on the masters have to be transferred photographically to the printing plates.

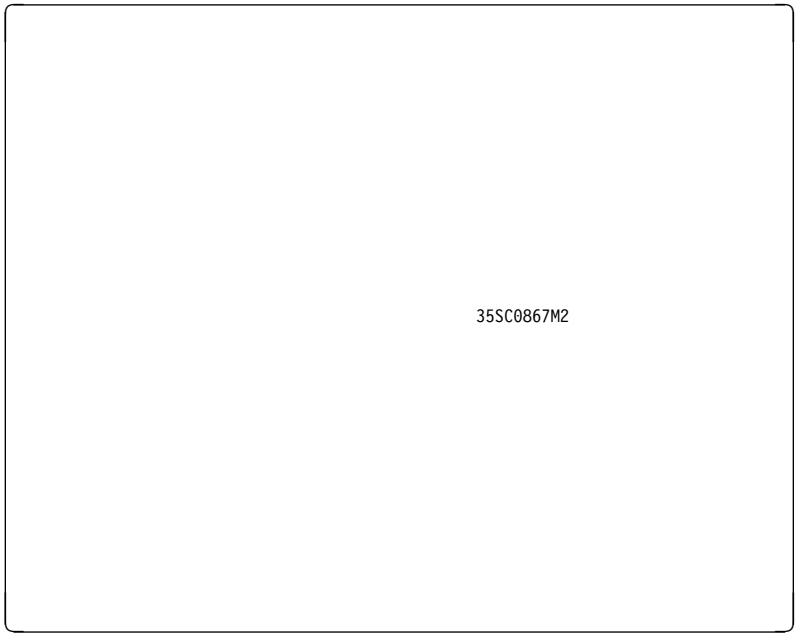


Figure 111. How a picture is changed into a number of color masters

In GDDM programs, you specify color attributes by numbers: 1 means blue, 2 means red, and so on. The numbers are listed in full in “Setting the current color, using GSCOL” on page 35. For example, 4 in a GSCOL call means green, so if you draw a line after executing this statement:

```
CALL GSCOL(4);
```

it appears in the publication as green. For this to happen, the line should be present on the yellow and cyan masters, but not on the magenta or black ones.

The method of defining how much of each color is on each of the plates is as follows. For each of GDDM's colors, a particular density is required on each plate. The amount is specified by means of a shading pattern and a color-master table entry. The pattern defines the density of the color. The table defines, for each master, which pattern is used to print each color.

The patterns belong to a pattern set created using the Image Symbol Editor. The patterns must be 32 pixels square, this being the notional cell size that GDDM uses for family-4 devices. Each pattern represents the density at which one of the four printing process colors should be printed so that it depicts a particular GDDM numbered color correctly. For instance, to print the correct shade of GDDM color 4, green, you may require a pattern for the yellow master in which 33% of the pixels are present, and another for the cyan master in which 50% of the pixels are present.

The GDDM-supplied symbol set, ADMDHIPK (see Figure 112 on page 429), gives an indication of what such a pattern set might be like.

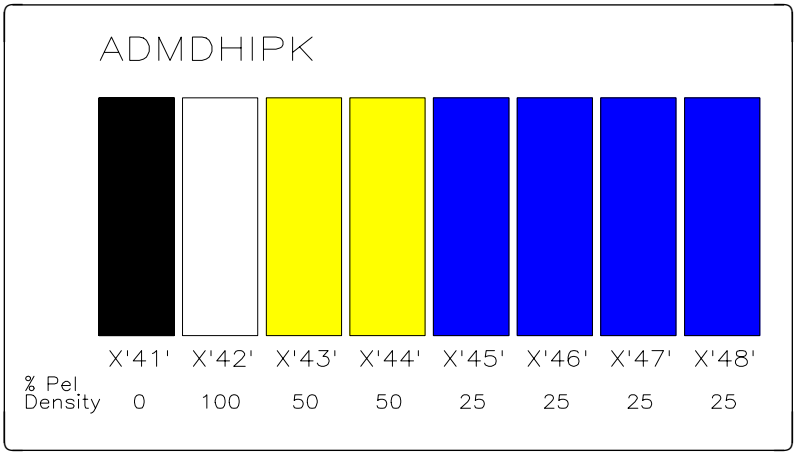


Figure 112. ADMDHIPK, the GDDM sample symbol set for color masters

To tell GDDM which patterns it must use on each master for each GDDM color, you code and assemble a macro, ADMMCOLT. The macro creates a **color-master table**. The macro must be assembled into a load module with the name ADMDJCOL. ADMMCOLT is described in the *GDDM Base Application Programming Reference* book. Here is an example of a color master table:

```
ADMDJCOL CSECT
         ADMMCOLT START,SETS=1
ADM00001 ADMMCOLT PATTERN=ADMDHIPK,COLORS=10,MASTERS=4,SETID=ADM00001
*
*                  YEL  MAG  CYN  BLK
*
DEFAULT  ADMMCOLT ( 41,  41,  41,  42)
BLUE     ADMMCOLT ( 41,  43,  44,  41)
RED      ADMMCOLT ( 43,  44,  41,  41)
PINK     ADMMCOLT ( 41,  42,  41,  41)
GREEN    ADMMCOLT ( 43,  41,  44,  41)
TURQSE   ADMMCOLT ( 41,  41,  42,  41)
YELLOW   ADMMCOLT ( 42,  41,  41,  41)
NEUTRAL  ADMMCOLT ( 41,  41,  41,  42)
BACKGRD  ADMMCOLT ( 41,  41,  41,  41)
ALLBLK   ADMMCOLT ( 42,  42,  42,  42)
         ADMMCOLT END
         END
```

In the first line of this macro, the number of sets being defined is specified. In the second, the pattern set from which the patterns are to be selected is specified as the GDDM-supplied one, ADMDHIPK. The number of GDDM colors is specified as 10. The number of masters is specified as four. The name of the color-master table is specified as ADM00001. Names have the form ADMnnnnn, where n is numeric.

The remaining lines specify the hexadecimal numbers of the patterns to be used. Each line represents a GDDM color and each column a master. The first line gives the four patterns for GDDM color 0, the second for color 1, and so on. For user-created patterns the numbers must be in the range 65 through 239 (X'41' through X'EF') in the same way as for user-defined patterns in the GSPAT call, which is described in “Setting the current shading pattern, using GSPAT” on page 37.

The example specifies that, for instance, color 4 (green) is to generate pattern X'43' for the first master, X'41' (that is, nothing) for the second, X'44' for the third, and X'41' (nothing again) for the fourth. Patterns X'43' and X'44' in ADMDHIPK have a pixel density of 50%. The pixels are arranged so that they do not overprint. The first master is used to make the yellow printing plate, the second for the magenta, the third for the cyan, and the fourth for the black.

The required patterns vary from one printing establishment to another, because of variations in inks, papers, printing technology, and so on. To get the required shade of green, for instance, you might need a 60% pattern for yellow, a 40% for cyan, and a 10% for black. Patterns can be determined only by trial and error. However, for many applications, such as printing business charts, it is not necessary to obtain precise shades, and the amount of experimentation required may be small.

GDDM supplies a number of sample color-master tables, based on the pattern set ADMDHIPK. GDDM also supplies some sample color-toning tables, based on another GDDM-supplied pattern-set ADMDHIPL. These are designed to show each input color as a different shade of gray. The definition and use of the color-toning tables are the same as for the color master tables, except that only one color master output file is created. The tables are contained in a GDDM-supplied module called ADMDJCOL.

When the program that creates the masters is executed, you must ensure that the file containing ADMDHIPK, or whatever pattern set you have specified, is available. Under CMS, for instance, you must ensure that a disk containing the pattern set has been accessed.

DSOPEN statement for color masters

You tell GDDM to create color-separation or color-toning masters in a processing option on the DSOPEN call. The option code is 3000. There is an example in Figure 113 on page 431. The fullword following the code must contain a number comprising one to five digits, corresponding to the numerical part of the required color table name. The full name of a color table has the form ADMnnnnn. GDDM expands the number in the option list to five digits, if necessary, by adding leading zeros, and adds “ADM” to the front, before searching for the color table. A parameter of 0 has the special meaning that monochrome output is required.

Note: For color masters to be created correctly, you must not specify any value other than 0 or 1 on processing option 9 (OFFORMAT). Allowing option 9 to default is sufficient but, if you cannot be sure that it won’t be set by a nickname statement, you should specify the value 1 on the DSOPEN call as in Figure 113 on page 431.

Each color master is created as a high-resolution image file of its own. Under CMS, GDDM uses the specified file name, and assigns a different file type to each master, of the form ADMCOLn, where n is a digit that ranges from 1 to the number of masters specified in the referenced color table. The *GDDM Base Application Programming Reference* book explains what to do under other subsystems.

```
DCL PROCOPT(12) FIXED BIN(31);

    PROCOPT(1) = 5;             /* Data-stream type   */
    PROCOPT(2) = 0;             /* Primary            */

    PROCOPT(3) = 9;             /* Output file format */
    PROCOPT(4) = 1;             /* CDPF               */

    PROCOPT(5) = 7;             /* Swathing           */
    PROCOPT(6) = 10;            /* 10  swathes        */

    PROCOPT(7) = 8;             /* Page size          */
    PROCOPT(8) = 85;            /* 8.5 inch wide      */
    PROCOPT(9) = 110;           /* 11 inch deep       */
    PROCOPT(10) = 0;            /* 1/10 inch measures */

    PROCOPT(11) = 3000;         /* Color masters      */
    PROCOPT(12) = 1;            /* Color-table ident  */

DCL NAMELIST(1) CHAR(8);

    NAMELIST(1) = 'COLMAST';    /* File name          */


      /* DEVICE_ID   FAMILY   TOKEN    PROC_OPTIONS    FILENAME */
CALL DSOPEN (11,       4,    'IMG85',   12,PROCOPT,   1,NAMELIST);

CALL DSUSE (1,11);

CALL FSPCRT(1,85,110,1);
CALL GSFLD(10,10,65,90);
   .
   .
   .
```

Figure 113. Creating color-separation masters

Chapter 21. Sending output from an application to a plotter

You can send graphics output to a plotter attached to a terminals such as the 3472-G. A typical use is for making hard copies of screen graphics.

GDDM applications can plot alphanumeric text, only if the plotter is attached to a PC or PS/2. On other configurations only the graphics field is sent to the plotter and alphanumeric calls are invalid. Graphics primitives outside segments are not plotted.

Plotters can be opened for family-1 or family-2 output. You tell GDDM that you intend to use a plotter by issuing a suitable DSOPEN call. It can be the primary or alternate device.

Nicknames can be used to send output originally created for a different device (for example, a family-2 printer) to a plotter.

DSOPEN for plotters

A DSOPEN call for a plotter requires a two-part name, identifying the workstation in the first part and the plotter in the second. Here is a simple example:

```
DECLARE PROCOPT_LIST(1)  FIXED BINARY(31);
DECLARE NAME_LIST(2)     CHARACTER(8);

NAME_LIST(1) = '*';
NAME_LIST(2) = 'ADMPLOT';

  /* DEVICE-ID  FAMILY  DEVICE-TOKEN     OPTIONS          NAME  */
CALL DSOPEN(99,   1,       '*',       0,PROCOPT_LIST,  2,NAME_LIST);
```

The two parts of the name are as follows:

- The * in the first element means the plotter is attached to the workstation from which the program was invoked. On CMS, you can send the output to a plotter on a different workstation by specifying the workstation's address.
- A workstation can have more than one plotter attached to it. All the plotters are given names when the workstation is customized. You can specify the name of a particular plotter in the second element of the name. The example uses the reserved name ADMPLOT. This tells GDDM to use the first (or only) plotter attached to the display or workstation.

GDDM can query the plotter, so rather than specify an explicit token name, you can place an asterisk (*) in the device-token parameter.

Note: If the workstation to which the plotter is attached is supported by GDDM-OS/2 Link, the DSOPEN for the plotter is the same as for a printer. For an example of plotting using GDDM-OS/2 Link, see "Family-1 output: GDDM directly attached printers" on page 401.

© Copyright IBM Corp. 1982, 1996

Processing options for plotters

A number of the physical characteristics of the plotter, such as the pen pressure and the plotting area, can be varied. Some characteristics can be set by the application program using processing options on the DSOPEN and some by the end user.

Here is an example of a DSOPEN call for a plotter that includes a set of processing options:

```
DECLARE PROCOPT_LIST(7)  FIXED BINARY(31);
DECLARE NAME_LIST(2)     CHARACTER(8);

PROCOPT_LIST(1) = 11;        /* Option group 11 = pen velocity  */
PROCOPT_LIST(2) = 50;        /* Set velocity to 50 cm/second    */

PROCOPT_LIST(3) = 14;        /* Option group 14 = plotting area */
PROCOPT_LIST(4) = 20;        /* x axis to run from 20% through  */
PROCOPT_LIST(5) = 70;        /* 70% of paper width              */
PROCOPT_LIST(6) = 10;        /* y axis to run from 10% through  */
PROCOPT_LIST(7) = 90;        /* 90% of paper depth              */

NAME_LIST(1) = '*';
NAME_LIST(2) = 'ADMPLOT';

   /* DEVICE-ID  FAMILY  DEVICE-TOKEN     OPTIONS         NAME */
CALL DSOPEN(3,     1,       '*',      7,PROCOPT_LIST,  2,NAME_LIST);
```

Controlling the velocity of plotter pens

By specifying processing option 11 (PLTPENV) on the DSOPEN for the plotter, your program can control the speed with which the pens draw the output (see “Optimum pen speed and pressure” on page 451).

If this option is allowed to default, the pen velocity that is set on the plotter takes effect. If you specify a value in the range 1 through 255 , this overrides the setting on the plotter. If the value you specify is more than the plotter's maximum, the maximum is used.

Specifying the width of plotter pen used

Depending on how the plotter has been set up, you can use processing option 12 to specify the width, in tenths of a millimeter, of the pens to be used.

The actual width of the pens used for plotting depends on which pens have been loaded into the plotter's pen holders. It is therefore outside the control of GDDM. GDDM uses the specified (or defaulted) value for shading areas, drawing double-width lines, drawing lines in the background color, and drawing images and image symbols. If these primitives are to be plotted correctly, the plotter operator must ensure that pens of the specified (or defaulted) width are loaded.

Controlling the pressure of plotter pens on paper

If you specify processing option 13 (PLTPENP) on the DSOPEN for the plotter, you can control the force of the pen on the plotter paper (see “Optimum pen speed and pressure” on page 451).

Some types of plotter do not have variable pen pressure, in which case the processing option is ignored.

If this processing option is allowed to default, the pen pressure that is set on the plotter takes effect. If you specify a value in the range 1 through 255, this overrides the setting on the plotter. If the value is more than the plotter's maximum, then the maximum is used.

Specifying the plotting area

By including processing option 14 (PLTAREA) on the DSOPEN for the plotter, you can specify on which part of the paper the picture is to be plotted. This is shown in Figure 114.

The plotting area is equivalent to the screen of a display device. The left- and right-hand edges are specified as percentages of the paper width, and the top and bottom edges as percentages of the paper depth. You can see an example of this in the procopt list of the DSOPEN call on page “Processing options for plotters” on page 434.

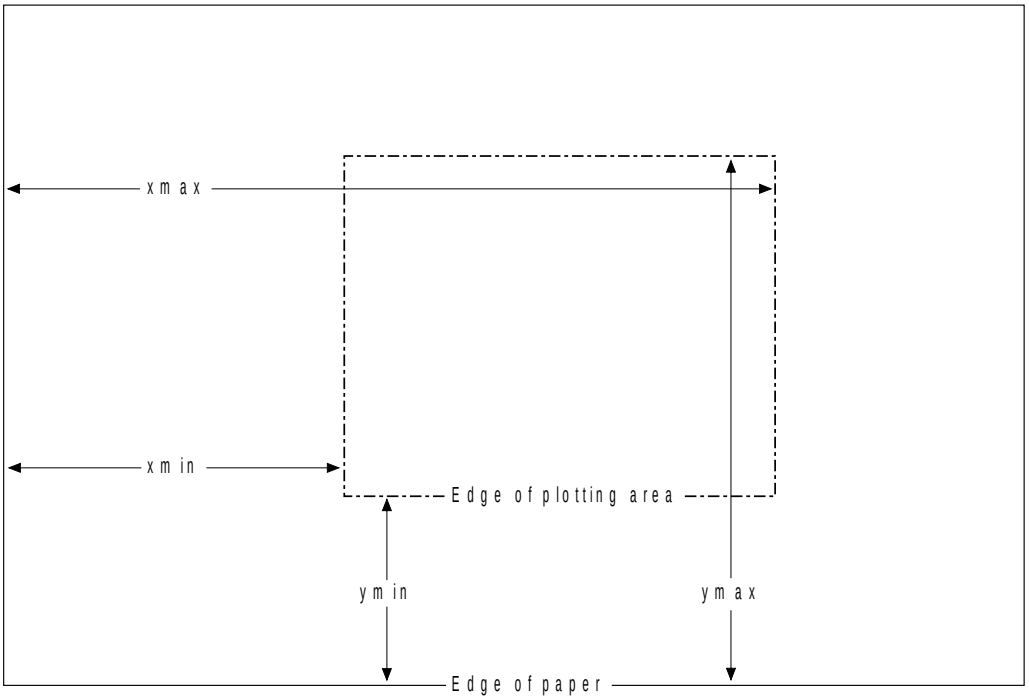


Figure 114. Plotting area

The default values are 0, 100, 0, and 100, meaning the whole of the paper. If 0,0,0,0 is specified, the value set on the plotter by the operator is used.

If you want to reflect the picture through the x or y axis, you can specify one or both of the maximum x and y values to be less that their corresponding minima.

Selecting the size of paper for the plotter

Depending on how the plotter has been set up, you can use processing option 15 (PLTPAPSZ) to specify the size of paper to be loaded into the plotter.

There are two sets of sizes–the International Organization for Standardization (ISO) series A4, A3, A2, A1, and A0, and the American National Standards Institute (ANSI) series A, B, C, D, and E. The third fullword of the group specifies which series is to be used, and the second fullword, which member of that series.

GDDM derives the usable plotting area from the paper size, the default plotting area being the whole paper area.

Some types of plotter can detect the size of paper with which they are loaded. Others require the operator to indicate it by setting switches. If both the paper size and the size type are set or defaulted to 0 in the processing option, GDDM queries the hardware for the paper size. If either is nonzero, GDDM takes the size from the processing option.

If a paper-size processing option is specified, GDDM assumes the specified value, irrespective of the actual paper size.

Some types of plotter have paper size switches. These must be set to match the actual paper size.

Long plots: On IBM 6186-2 and 6187-2 plotters with “roll-feed” paper, you can plot GDDM output from your applications up to a length of 11.86 meters or 38.9 feet. GDDM draws the long plot over a series of frames, the size of each frame being determined by the size of paper loaded.

You can determine, using the FSQURY call, whether the plotter has the necessary roll feeder attachment to facilitate long plots.

Long plotting is supported on three widths of roll paper and the maximum length and frame size you specify for the plotter is different for each width of paper.

For a paper roll that is 36 inches (92 cm) wide, you can specify a frame size of up to A0 or E on processing option 15 (PLTPAPSZ) of the DSOPEN call for the plotter. Then using the FSPCRT call, you define the size of the GDDM page with either the depth or the width greater than the default size of each for an A0 page. If you have specified rotated output on processing option 16 (PLTROTAT), you need to specify a depth for the GDDM page that exceeds the default depth for the A0 page.

The default size of the A0 frame is 320 cells wide by 128 cells deep. To output a plot three times as long as the normal A0 page, you need an FSPCRT call like this:

```
/*          ID  DEPTH   WIDTH                                    */
CALL FSPCRT( 1,   128,    960,     0);
CALL FSPSEL( 1 );
⋮
CALL FSFRCE;
```

You can also send output to a plotter defined as an alternate device:

```
CALL DSOPEN( 31,    1,     '*',    0,PROCOPTS,  2,NAME_LIST);
CALL DSUSE ( 2,  31 );                 /* Use as alternate device */
⋮
/*          WIDTH% DEPTH% H-OFF V-OFF CNT OPTS                    */
CALL DSCOPY( 300,   100,    0,    0,   3,  OPT);
```

Table 7. Plotter-page sizes available to plotters with roll-feed media.

Roll paper width	Default (unrotated) Depth	& Width	Maximum (unrotated) Depth	& Width	Maximum (rotated) Depth	& Width
36 in. 92 cm.	128	320	128	3200	1280	320
24 in. 61.5 cm.	90	226	90	2486	990	226
11 in. 28 cm.	45	113	45	1130	450	113

Depth values are in plotter rows and width values are in plotter columns.

Picture orientation

If a plotting area has been specified with option group 14, then by default GDDM plots the x axis parallel to the longer side of the paper (sometimes called landscape format).

You can specify a value on processing option 16 (PLTROTAT) on the DSOPEN for the plotter to rotate the plotted output by 90 degrees. It may be necessary to adjust the size of the page, if you do this. Alternatively, you can use the DSCOPY call and use the third element of the option array to rotate the picture through angles of 90, 180 or 270 degrees.

Saving plotted output in a file

You can enable end users of your application to save graphics output in a file instead of sending it directly to the plotter. Using processing option 46 (TOFILE) on the DSOPEN or nickname statement for the plotter, you can override the default action, which is to send output directly to the plotter.

When graphics output is stored in plot files it is converted into IBM-GL format, which is described in the *IBM-GL Programming Manual (Graphics Language)*. The NAME parameter of the DSOPEN or nickname statement for the plotter is used to name the GL file created. End users can use a plotter driver made available at their installations to process the IBM-GL plot files for output on plotters.

Although the plotted output is saved in a file, you must still specify family-1 output as is usual for plotters.

The POSTPROC processing option enables you to specify a program that is to perform postprocessing on the GL file created. You can specify a program that performs the download using this procopt on the DSOPEN call.

You can find an example of a program that creates a GL plot file in “A C/370 programming example” on page 505.

Setting up the plotter

The plotter operator can affect the appearance of a plot in a number of ways that GDDM cannot detect.

Some or all of the following characteristics (depending on the plotter model) are under operator control if your program does not set them with processing options:

- Pen velocity
- Pen pressure
- Plotting area

- Picture orientation.

The following characteristics are always under operator control even though you can specify them in processing options:

- Pen width
- Paper size.

The reason you can specify them in processing options is that GDDM needs to know their values to generate the correct picture.

The color of the pen in each of the plotter's pen holders depends on the operator - GDDM cannot control the colors or determine what they are. In the normal case, the operator should ensure that they correspond as closely as possible to GDDM's color numbering scheme (see “Colors” on page 446).

So there is considerable scope for wrong pictures resulting from a plotter set up with different characteristics from those which you assumed when you wrote your program. For any plotter application, therefore, you should consider displaying setup instructions on the screen of the workstation. After displaying the instructions, your program should wait for a response from the operator confirming that setup is complete before sending the picture to the plotter.

Terminating a plot

To terminate the plotting of a picture before it is complete, the operator can press the CLEAR key on the keyboard of the display or workstation to which the plotter is attached.

Cells, pixels, and plotter units

Some GDDM graphics functions require the current device to have cells (character boxes) and pixels. The GSFLD call and mode-1 graphics text, for instance, require a cell size, and images and image symbols require a pixel size. For devices such as plotters that do not have real cells and pixels, GDDM assumes notional ones.

The notional cell for a plotter is such that a GDDM-defined number of rows and columns can be fitted into the plotting area. The plotting area is analogous to the screen of a display device, and the GDDM-defined rows and columns are analogous to the rows and columns of hardware cells on a screen.

The numbers depend on the paper size. For American A and metric A4 paper, 32 rows of cells and 80 columns would fill the plotting area. Full details for all paper sizes are given in the *GDDM Base Application Programming Reference* book. Because the rows and columns are defined as fitting the plotting area, changing the area's dimensions changes the notional cell size. This is a simple way of changing the size of a plot.

You can discover the notional cell density of the current device using a DSQDEV call. The last parameter is an array. In the third and fourth elements of this array GDDM returns the default number of cell rows and columns in the plotting area. Here is an example:

```
DECLARE D_TOKEN CHARACTER(8);
DECLARE P_LIST(1) FIXED BINARY(31);
DECLARE N_LIST(1) CHARACTER(8);
DECLARE QDEV(4) FIXED BINARY(31);

DECLARE (ROWS,COLUMNS) FIXED BINARY(31);

       /* DEVICE-ID   TOKEN   PROC. OPTIONS     NAME   CHARACTERISTICS */
CALL DSQDEV( 11,     D_TOKEN,   0,P_LIST,     0,N_LIST,    4,QDEV );

ROWS        = QDEV(3);
COLUMNS     = QDEV(4);
```

The notional pixels are dots spaced at the width of the pen. GDDM detects the pen width from the processing options, or assumes 0.3 millimeters if no pen width is specified.

Plotter units are smaller than pixels. They are the smallest possible displacement of a pen. They represent the maximum accuracy of the plotter – its resolution.

You can query the plotter units using the last parameter of DSQDEV. In the fifth and sixth elements, GDDM returns the depth and width of each cell in plotter units. In the seventh and eighth elements, it returns the number of plotter units per meter vertically and horizontally. Here is an example:

```
DECLARE QDEV(8) FIXED BINARY(31);
DECLARE D_TOKEN CHARACTER(8);
DECLARE P_LIST(1) FIXED BINARY(31);
DECLARE N_LIST(1) CHARACTER(8);

DECLARE (CELL_DEPTH,CELL_HEIGHT,VERTL_RES,HORTL_RES) FIXED BINARY(31);

   /* DEVICE-ID  TOKEN   PROC. OPTIONS     NAME   CHARACTERISTICS */
CALL DSQDEV( 12, D_TOKEN,  0,P_LIST,     0,N_LIST,    8,QDEV );

CELL_DEPTH  = QDEV(5);
CELL_WIDTH  = QDEV(6);
VERTL_RES   = QDEV(7);
HORTL_RES   = QDEV(8);
```

A simple plotting program

The program in Figure 115 on page 440 uses the plotter as the primary device. It plots a picture created by another program and stored on a segment library. The picture, called ADMTEST, is retrieved with a GSLOAD call. The program could, instead, have drawn a picture using the ordinary primitive and attribute calls such as GSLINE, GSMOVE, GSAREA, GSCOL, and so on.

No processing options have been specified, so they all take their default values. The operator must ensure that the pen holders are loaded with pens of 0.3 millimeter width (for instance, the standard fiber-tipped pens), with the correct color in each holder. On all plotters, the plotting area is the whole paper. The pen velocity and pen pressure are as set by the operator (or the fixed hardware values on plotters that do not allow the operator to vary them).

The GDDM page size and graphics field are allowed to default. This means that they fill the plot area. When a plotter is used as the primary device, a page size or graphics field (or both) can be specified in terms of the notional cells described in “Cells, pixels, and plotter units” on page 438.

The program displays setup instructions for the IBM 7375 plotter, which can detect the size of paper loaded, and has adjustable pen velocity and pressure.

```
PLOT1: PROC OPTIONS(MAIN);

DECLARE (ATYPE,AVAL,ACOUNT) FIXED BINARY(31);
DECLARE PROCOPT_LIST(1) FIXED BIN(31);
DECLARE NAME_LIST(2) CHAR(8);
DECLARE CNTRL(2) FIXED BIN(31);
DECLARE COUNT FIXED BIN(31);
DECLARE DESC CHAR(50);

CALL FSINIT;
/*******************************************************************/
/*               DISPLAY PLOTTER SETUP INSTRUCTIONS                */
/*******************************************************************/
CALL GSSEG(0);
CALL GSCM(3);
CALL GSCOL(6);
CALL GSCHAR(25.0,95.0,40,'  HOW TO SET UP THE IBM 7375 PLOTTER    ');
CALL GSCOL(1);
CALL GSCHAR(25.0,90.0,40,'CHECK THERE IS A FIBER-TIPPED PEN IN    ');
CALL GSCHAR(25.0,85.0,40,'EACH HOLDER WITH THE FOLLOWING COLOR:   ');
CALL GSCOL(6);
CALL GSCHAR(25.0,80.0,40,'   PEN HOLDER        COLOR             ');
CALL GSCOL(1);
CALL GSCHAR(25.0,75.0,40,'       1             BLUE              ');
CALL GSCHAR(25.0,70.0,40,'       2             RED               ');
CALL GSCHAR(25.0,65.0,40,'       3             PINK              ');
CALL GSCHAR(25.0,60.0,40,'       4             GREEN             ');
CALL GSCHAR(25.0,55.0,40,'       5             TURQUOISE         ');
CALL GSCHAR(25.0,50.0,40,'       6             YELLOW            ');
CALL GSCHAR(25.0,45.0,40,'       7             BLACK             ');
CALL GSCHAR(25.0,40.0,40,'       8             GREEN             ');
CALL GSCHAR(25.0,35.0,40,'SET THE PEN SPEED AND FORCE TO SUITABLE ');
CALL GSCHAR(25.0,30.0,40,'VALUES (SEE PLOTTER OPERATING MANUAL).  ');
CALL GSCHAR(25.0,20.0,40,'LOAD THE PLOTTER WITH PAPER OF THE SIZE ');
CALL GSCHAR(25.0,15.0,40,'YOU REQUIRE.                            ');
CALL GSCOL(7);
CALL GSCHAR(25.0, 3.0,40,'    PRESS ENTER WHEN READY TO PLOT      ');
CALL GSSCLS;
CALL ASREAD(ATYPE,AVAL,ACOUNT); /* Send instructions to screen     */
IF ATYPE¬=0 THEN GO TO FIN;     /* Plot only if ENTER pressed      */
CALL DSDROP(1,0);               /* Drop screen as primary device   */
```

Figure 115 (Part 1 of 2). Program using plotter as primary device

```
/*****************************************************************/
/*               OPEN THE PLOTTER                                */
/*****************************************************************/
NAME_LIST(1)='*';              /* Is attached to invoking terminal*/
NAME_LIST(2)='ADMPLOT';          /* Special GDDM-defined name     */
       /* DEV ID  FAMILY  DEV TOKEN  PROCESSING OPTIONS  DEV NAME  */
CALL DSOPEN( 101,    1,     '*',      0,PROCOPT_LIST,  2,NAME_LIST );
CALL DSUSE(1,101);                /* Use as primary device        */
/*****************************************************************/
/*                LOAD A PICTURE                                 */
/*****************************************************************/
CNTRL(1) = 0;                     /* Keep original segment ids    */
CNTRL(2) = 2;                     /* Make as big as possible      */
       /* OBJECT-NAME ARRAY-CNT ARRAY  SEG-CNT DESCRIP-LEN DESCRIP */
CALL GSLOAD( 'ADMTEST',   2,    CNTRL,  COUNT,      50,     DESC);
/*****************************************************************/
/*                 SEND PICTURE TO PLOTTER                       */
/*****************************************************************/
CALL FSFRCE;

FIN:
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END PLOT1;
```

Figure 115 (Part 2 of 2). Program using plotter as primary device

Copying screen output to a plotter

Many programs first create pictures on the screen and then enable the end user to copy them to a plotter. In such applications, you need to use the plotter as an alternate device and issue a DSCOPY, FSCOPY, or GSCOPY call to copy the output to it.

The program in Figure 116 on page 442 is an example of a program that creates a picture on the screen and then copies it to the plotter. The picture is loaded in the block of code at **A**. It could be created in any of the usual ways, including primitive calls like GSLINE and GSCOL.

For the instructions to the plotter operator, a new GDDM page is created at **B**.

The plotter is opened at **C**, and its use as the alternate device is specified at **D**. The original page (the default one, page 0) is reselected at **E** for the DSCOPY call, **F**, to send the picture to the plotter.

GDDM maintains the aspect ratio of the graphics when copying to the plotter: the picture fills as much of the area specified on the DSCOPY as possible without distortion. The bottom left-hand corner of the graphics field is placed at the bottom left of the plotting area.

```
PLOT2: PROC OPTIONS(MAIN);

DCL PROCOPTS(1) FIXED BIN(31);
DCL NAME_LIST(2) CHAR(8);
DCL DEV_TOKEN CHAR(8);
DCL OPT_ARRAY(2) FIXED BIN(31)  INIT(0, 0, 1);   /* 90° rotation */
DCL QDEV(4) FIXED BINARY(31);
DCL (ROWS,COLUMNS) FIXED BIN(31);
DCL CNTRL(2) FIXED BIN(31);
DCL DESC CHAR(50);
DCL COUNT FIXED BIN(31);
DCL (ATYPE,AVAL,ACOUNT) FIXED BIN(31);
CALL FSINIT;

/*****************************************************************/
/*                LOAD A PICTURE                                 */  A
/*****************************************************************/
CNTRL(1) = 0;                     /* Keep original segment ids   */
CNTRL(2) = 2;                     /* Make as big as possible     */
CALL GSSEG(1);                    /* Begin new segment           */
       /* OBJECT-NAME ARRAY-CNT ARRAY  SEG-CNT DESCRIP-LEN DESCRIP   */
CALL GSLOAD( 'ADMTEST',   2,    CNTRL,  COUNT,      50,     DESC);
CALL GSSCLS;

/*****************************************************************/
/*                DISPLAY PLOTTER SETUP INSTRUCTIONS             */
/*****************************************************************/
CALL FSPCRT(1,0,0,0);             /* Create a new page           */  B
CALL GSSEG(0);
CALL GSCM(3);
CALL GSCOL(6);
CALL GSCHAR(25.0,95.0,40,'  HOW TO SET UP THE IBM 7375 PLOTTER    ');
/*   .                                                           */
/*   .                                                           */
/*   .                                                           */
CALL GSCHAR(25.0, 3.0,40,'    PRESS ENTER WHEN READY TO PLOT      ');
CALL GSSCLS;
CALL ASREAD(ATYPE,AVAL,ACOUNT);   /* Send instructions           */
IF ATYPE¬=0 THEN GO TO FIN;       /* Plot only if ENTER pressed  */

/*****************************************************************/
/*                OPEN THE PLOTTER                               */
/*****************************************************************/
NAME_LIST(1)='*';              /* Is attached to invoking term.  */
NAME_LIST(2)='ADMPLOT';        /* Special GDDM-defined name      */
        /*  DEV ID  FAM  DEV TOK  PROCESSING OPT  DEV NAME       */
CALL DSOPEN( 202,    1,     '*',    0,PROCOPTS,  2,NAME_LIST);      C
CALL DSUSE(2,202);                   /* Use as alternate device */  D
```

Figure 116 (Part 1 of 2). Program using plotter as alternate device

```
/*****************************************************************/
/*                SEND PICTURE TO PLOTTER                        */
/*****************************************************************/
CALL FSPSEL(0);                      /* Select page with picture */  E
/*          WIDTH DEPTH  HOR_OFF  VER_OFF   COUNT  OPT_ARRAY     */
CALL DSCOPY(100,   100,     0,       0,       3,    OPT_ARRAY);     F
FIN:
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END PLOT2;
```

Figure 116 (Part 2 of 2). Program using plotter as alternate device

Plotting to scale

You may need to plot your program's output at a specific size. You can do this by making each window (world-coordinate) unit represent a particular physical measurement in the plotted output. The programming example in Figure 117 on page 444 makes one window unit plot as one millimeter, and then draws a 100-millimeter square.

To define the window units as required, you need to query three things: the number of rows and columns of notional cells in the current graphics field, the number of plotter units per cell in each direction, and the density of plotter units (or resolution) in both directions. This information is obtained at **C** and **E**.

The subsequent statements show how to calculate the number of window units to make one unit equal to one millimeter when plotted. The calculations multiply the number of rows and columns by the depth and width of a cell in plotter units to obtain the depth and width of the graphics field in plotter units. This value is divided by the number of plotter units per meter, and multiplied by 1000 to convert meters to millimeters.

Before the device can be queried, it must be opened, as at **A**. It is then made current, at **B**. Before the graphics field can be queried, it must be created. The example forces the creation of a default graphics field by issuing a graphics primitive call (GSMOVE at **D**). Another way that the program could create the same graphics field is by explicitly using a GSFLD call, specifying the page size as returned in QDEV(3) and QDEV(4). The statements **D** and **E** would be replaced by the following lines:

```
ROWS = QDEV(3);
COLS = QDEV(4);
CALL GSFLD(1,1,ROWS,COLS);
```

After the required graphics window has been created at **F**, a square of 100 window units is drawn. When plotted following the FSFRCE call at **G**, it is 100-millimeters square.

```
PLOT3: PROC OPTIONS(MAIN);

DCL PROCOPT_LIST(1) FIXED BIN(31);
DCL NAME_LIST(2) CHAR(8);
DCL DEV_TOKEN CHAR(8);
DCL QDEV(8) FIXED BIN(31);
DCL (ROW_POS,COL_POS,ROWS,COLS) FIXED BINARY(31);
DCL (CELL_WIDTH,CELL_DEPTH,VERTCL_RESLN,HORZTL_RESLN,
     WINDOW_DEPTH,WINDOW_WIDTH)   FLOAT DEC(6);

CALL FSINIT;
/*************************************************************/
/*                   OPEN THE PLOTTER                        */
/*************************************************************/

NAME_LIST(1)='*';             /* Is attached to invoking term.*/
NAME_LIST(2)='ADMPLOT';       /* special GDDM-defined name    */

        /* DEV ID  FAMILY  TOKEN     OPTIONS           NAME   */
CALL DSOPEN(303,     1,     '*',  0,PROCOPT_LIST,  2,NAME_LIST);   A

CALL DSUSE(1,303);            /* Use as primary device        */   B

/*************************************************************/
/*      SET UP WINDOW TO GIVE 1 WINDOW UNIT = 1 MILLIMETER    */
/*************************************************************/

CALL DSQDEV(303,DEV_TOKEN,0,PROCOPT_LIST,0,NAME_LIST,8,QDEV);     C

CALL GSMOVE(0.0,0.0);         /* Force creation of default    */  D
                              /* graphics field               */
CALL GSQFLD(ROW_POS,COL_POS,ROWS,COLS);                           E

CELL_DEPTH = QDEV(5);         /* Cell depth in plotter units  */
CELL_WIDTH = QDEV(6);         /* Cell width in plotter units  */
VERTCL_RESLN = QDEV(7);      /* Plotter units/meter vertically*/
HORZTL_RESLN = QDEV(8);    /* Plotter units/meter horizontally*/
WINDOW_DEPTH =
  (CELL_DEPTH*ROWS/VERTCL_RESLN)*1000;/*Calculate required X..*/
WINDOW_WIDTH =
  (CELL_WIDTH*COLS/HORZTL_RESLN)*1000;/* ..and Y window units */

CALL GSUWIN(0.0,WINDOW_WIDTH,0.0,WINDOW_DEPTH);                  F
```

Figure 117 (Part 1 of 2). Scale plotting program

```
/*************************************************************/
/*            DRAW A SEGMENT  (A SQUARE)                     */
/*************************************************************/

CALL GSSEG(2);                /* Current position = 0,0       */
CALL GSLINE(100.0,0.0);
CALL GSLINE(100.0,100.0);
CALL GSLINE(0.0,100.0);
CALL GSLINE(0.0,0.0);
CALL GSSCLS;

CALL FSFRCE;                  /* Send to plotter              */ G

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

END PLOT3;
```

Figure 117 (Part 2 of 2). Scale plotting program

Using nicknames to direct and control plotted output

If you have a program that currently sends graphics to a printer, end users can have the output sent to a plotter instead by creating a suitable nickname file (see “Coding a partial device definition for end users to change with nicknames” on page 374).

Diverting a program’s output from a printer to a plotter

Here are some examples of nickname statements that divert output from printers to a plotter:

```
ADMMNICK FAM=1,NAME=061,
     TOFAM=1,TONAME=(*,ADMPLOT)
ADMMNICK FAM=2,NAME=PR2,
     TOFAM=1,TONAME=(*,ADMPLOT)
ADMMNICK FAM=4,NAME=PR4,
     TOFAM=1,TONAME=(*,ADMPLOT)
```

The NAME parameters specify the printer device names supplied in DSOPEN calls. These three statements redirect any output for family-1, -2, and -4 printers named as 061, PR2, and PR4, respectively. The output goes to the first plotter attached to the invoking terminal instead of the named printer. No example has been given for family-3 printers because these devices support alphanumerics and not graphics, whereas plotters support graphics and not alphanumerics.

Diverting a program's output from a plotter to a printer

You can also make the reverse change (diverting output from a plotter to a printer) using nicknames. For instance, this statement sends the output to a family-2 print file called PR2, instead of a plotter:

```
ADMMNICK FAM=1,NAME=(*,ADMPLOT),
     TOFAM=2,TONAME=PR2
```

You can set up a single nickname statement to ensure that all output for a device with a particular name goes to a plotter. The following statement sends all output for any device called PLOTTER, of whatever family, to a plotter attached to the invoking terminal.

```
ADMMNICK NAME=PLOTTER,
     TOFAM=1,TONAME=(*,ADMPLOT)
```

Diverting a program's output from a plotter to an IBM-GL file

If some end users of your application don't have immediate access to a plotter, you can provide them with the ability to format the output for a plotter and then save it in a file using a nickname statement such as this:

```
  ADMMNICK FAM=2,NAME=GLPLOT,TOFAM=1,DEVTOK=L6187,
     TONAME=(PLOTFILE,GL),PROCOPT=((TOFILE,YES,REP)),
     DESC='GL plot file'
```

The TOFILE PROCOPT is described in the *GDDM Base Application Programming Reference* book. Some of the parameters here are system-dependent. This nickname is suitable for use with the ICU on VM.

Supplying processing options

In all cases, you can supply other processing options for the plotter or printer by adding a PROCOPT parameter to the nickname statement. A full list of the parameters is given in *GDDM Base Application Programming Reference* book.

In any case where a processing option in a DSOPEN call conflicts with an option in a nickname statement, the DSOPEN specification takes precedence.

Special considerations for graphics on plotters

Colors

The numbers you specify for colors in calls such as GSCOL become pen numbers when the output goes to a plotter. On a display unit, this call:

```
CALL GSCOL(1);
```

means that subsequent primitives are to be displayed in blue. On a plotter, it means the primitives are to be plotted with pen number 1. Whether this is blue or some other color depends on what pen has been loaded into the pen holder in position 1. It is the plotter operator's responsibility to ensure that each pen-holder position has a pen of the required color. A suggested scheme is shown in Table 8 on page 447.

Table 8. Suggested color scheme for plotter pens

Pen number	Suggested color		
	2-pen plotter	**6-pen plotter**	**8-pen plotter**
1	Black	Blue	Blue
2	Red	Red	Red
3		Magenta	Magenta
4		Green	Green
5		Cyan	Cyan
6		Black	Orange
7			Black
8			Green

Complications arise because GDDM cannot determine the colors of pens in the holder, and because the number of pens varies from one type of plotter to another. GDDM's actions are summarized in Table 9 on page 448. In more detail, this is what happens:

- For the default number, 0, GDDM always uses the highest-numbered pen.
- For color 8, which is defined as the background color, GDDM uses no pen. It imitates a primitive drawn in the background color–the color of the paper. Such a primitive would be invisible, except where drawn on top of a primitive of a different color. Where this happens, GDDM clips the underlying primitive to leave a clear line or area representing the overlying primitive. In the case of overlying lines, the width of the clipped area is equal to the pen width as specified in the processing options, or 0.3 millimeters by default (see “Processing options for plotters” on page 434).
- Because color 8 (background) does not use any plotter pen, pen holder 8 on the plotter can be used to hold an extra color, which your application can use by specifying color 0 or 16.
- For color –2, defined as white, GDDM takes the same action as for color 8; this means, on all plotters, using the background. For color –1, defined as black, GDDM takes the same action as for color 7; if the suggested color scheme for the pens is followed, the black pen is used.
- If the color number is higher than the highest pen number, GDDM wraps around the set of numbers after the lowest power of 2 that is equal to or greater than the highest pen number. This means after 8 for a six-pen plotter or after 2 for a two-pen plotter. Numbers between the highest pen number and the next power of 2 use the highest pen number. So on a six-pen plotter, color 7, and also color 6, use pen 6 (color 8 is an exception–it always has the special meaning of “background”); color 9 uses pen 1, color 10 pen 2, and so on. And on a two-pen plotter, color 3 uses pen 1, color 4 pen 2, color 5 pen 1, and so on.
- Color 7, which is defined as neutral and displayed as white on a color screen, uses a pen (unlike color 8). GDDM selects pen 7 on eight-pen plotters, and follows the wrapping algorithm on the other plotters.

When designing an application in which plotter output is important, it is advisable to experiment with the colors. A usable and pleasing picture on the screen may not be so if it is plotted unchanged.

Table 9. Color and pen numbers on plotters

Color number	Meaning	Color on screen*	Pen number (and suggested color)		
			2-pen plotter	6-pen plotter	8-pen plotter
–2	White	White	No pen	No pen	No pen
–1	Black	Black	1 (black)	6 (black)	7 (black)
0	Default	Green	2 (red)	6 (black)	8 (green)
1	Blue	Blue	1 (black)	1 (blue)	1 (blue)
2	Red	Red	2 (red)	2 (red)	2 (red)
3	Magenta	Magenta	1 (black)	3 (magenta)	3 (magenta)
4	Green	Green	2 (red)	4 (green)	4 (green)
5	Cyan	Cyan	1 (black)	5 (cyan)	5 (cyan)
6	Yellow	Yellow	2 (red)	6 (black)	6 (orange)
7	Neutral	White	1 (black)	6 (black)	7 (black)
8	Background	Black	No pen	No pen	No pen
9	Dark blue	Dark blue	1 (black)	1 (blue)	1 (blue)
10	Orange	Orange	2 (red)	2 (red)	2 (red)
11	Purple	Purple	1 (black)	3 (magenta)	3 (magenta)
12	Dark green	Dark green	2 (red)	4 (green)	4 (green)
13	Turquoise	Turquoise	1 (black)	5 (cyan)	5 (cyan)
14	Mustard	Mustard	2 (red)	6 (black)	6 (orange)
15	Gray	Gray	1 (black)	6 (black)	7 (black)
16	Brown	Brown	2 (red)	6 (black)	8 (green)

* On 3270-PC/GX workstation. For a 3179-G, 3192-G, 3472-G, 3270-PC/G, 3279, and the 5550 family, only eight colors are available, and numbers in the range 9 through 15 wrap around to the colors blue through neutral (white), and 16 to the default color of green.

Color mixing

With two exceptions, overlying primitives are plotted on top of underlying ones. The resulting colors depend on the physical and chemical interactions of the inks.

The exceptions apply in underpaint or overpaint mode only. They are:

- Any primitive, other than an image or image symbol, that underlies a solid shaded area is clipped at the edge of the area.
- If the overlying primitive is:
  - In background color or explicit white (colors 8 and minus.2),
  - And is a line (or arc), a vector symbol (or marker), or a solid-shaded area,

  then any underlying primitives, other than images and image symbols, are clipped to allow background to show through, as explained in the section “Colors” on page 446.

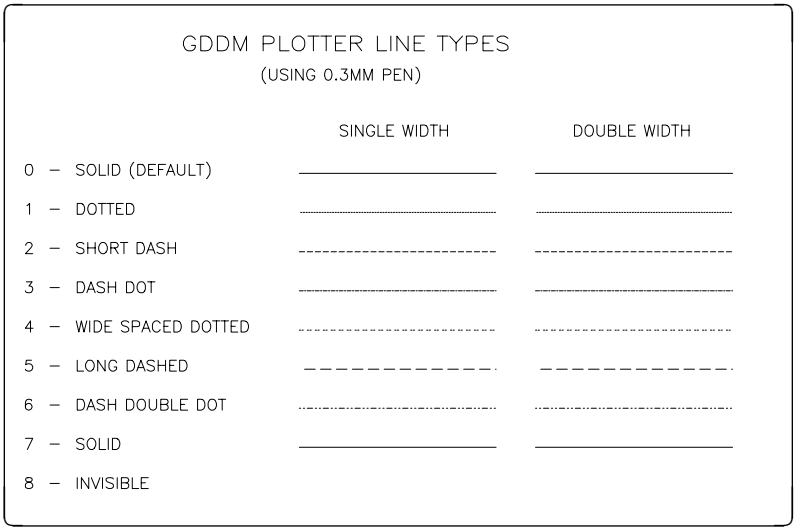


Figure 118. The eight GDDM line types for plotters

In summary, underlying primitives other than images and image symbols are clipped at the boundaries of all overlying solid-shaded areas. They are also clipped at overlying background-color vectors.

There is no such clipping in mix mode.

If you use underpaint mode for a picture that is displayed on the screen of a 3270-PC/G or /GX workstation that is also being plotted, the results differ. Underpaint mode is not supported on these displays; it is implemented as overpaint.

Performance considerations: Reverse clipping to give white graphics can use a lot of processing time in the host computer, depending on the complexity of the picture. In order to minimize the processing you are recommended to:

- Keep the number of lines and characters in colors –2 and 7 to a minimum.
- Avoid drawing lines, characters, or solid-shaded areas (especially complex ones) in colors –2 and 7, on top of solid shaded areas.

Graphics images and image symbols

These are always plotted, unless in background color, in which case they are, in effect, ignored.

They are clipped at the edge of the plot area. On a screen, they are clipped at the edge of the graphics field, so they may extend over a bigger area on the plot than on the screen.

Line types and widths

The line types for plotters are shown in the two available widths in Figure 118. The line type (1 through 8) is specified in the GSLT call, and the line width (1 or 2) in the GSLW or GSFLW call. Double-width lines are achieved by the plotter drawing two single-width lines next to each other. On long double-width dashed lines, the two lines can get out of synchronization. If this is a problem, you could specify a single-width line, and a particular color for these lines, but put a thicker pen in the plotter-pen stall for that color.

Shading patterns

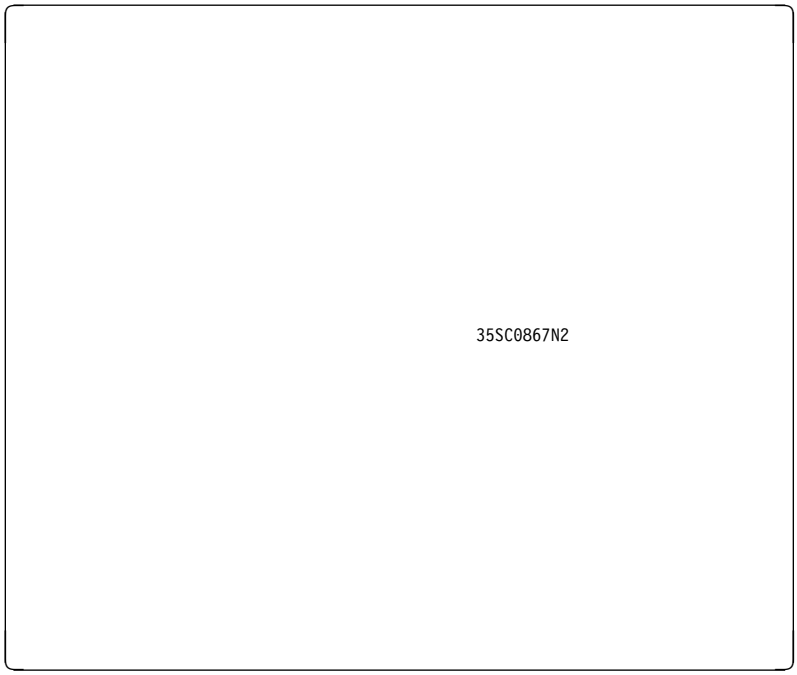


Figure 119. The 16 GDDM shading patterns for plotters

There are sixteen special GDDM-defined shading patterns for plotters. They are illustrated in Figure 119. The numbers are the ones that you would specify in a GSPAT call. No user-defined patterns can be specified for a plotter.

Shading can take a relatively long time on plotters. The single-hatched patterns (9 through 14) are quicker to plot than the cross-hatched ones (1 through 8). The solid pattern (0 or 16) is the slowest.

The separation of the shading lines depends on the pen width, as specified in the processing options, or as defaulted (see “Processing options for plotters” on page 434). If the specified or defaulted width differs from the actual width of the pen, the shading pattern may not be satisfactory. For instance, if the pen is actually narrower than specified, the “solid” pattern is not solid: gaps appear between the shading lines.

Symbol sets

The functions described in Chapter 12, “Using symbol sets” on page 233 apply to plotters as well as terminals. The symbols are drawn using the calls described in Chapter 4, “Creating graphics-text output in your application” on page 57.

Both image and vector symbols are supported on plotters. The image symbols are drawn using the notional pixels described in “Cells, pixels, and plotter units” on page 438. The size therefore depends on the specified or defaulted pen size.

Plotting pixels is relatively slow, and it quickly wears the pens. To alleviate these problems, GDDM plots all sets of contiguous pixels in the x direction as lines. Nevertheless, extensive use of image symbols is not advised on plotters.

The GDDM default symbol set for all modes of graphics text is the vector set ADMDVSS. To use an image set for mode-1 or -2, you must load it using a GSLSS call.

Optimum pen speed and pressure

The most suitable speed and pressure depend on the type of pen and the medium (paper, transparency foils, and so on) on which you are plotting.

In general, roller ball pens are the best at the highest speeds, and they may need the maximum force. Felt tips should ideally be used at somewhat below the highest speed and force. And drafting pens require a low speed and force. More detailed recommendations are given in the *GDDM Base Application Programming Reference* book and in the plotter operating manuals.

Chapter 22. Designing end-user interfaces for your applications

This section tells you about the GDDM calls that organize the screen of a display device into rectangular areas, using these different types of presentation structure:

- **Partitions** (application windows)
- **Operator windows**

Partitions and operator windows were first introduced in Chapter 7, “Hierarchy of GDDM concepts” on page 107. See that chapter for a brief description of the difference between them. Partitions are fully described in the first part of this chapter. Operator windows are fully described in “Using operator windows to write task-manager programs” on page 479 in the second part of this chapter.

Using partitions to divide up the screen

This section tells you how to create and use partitions. Partitions can be real or emulated. The IBM 3193 Display Station, 3290 Information Panel, and 8775 Display Terminal, have hardware facilities that enable application programs to create **real** partitions. Alternatively, GDDM can emulate partitions on all family-1 displays. A GDDM application program can create, position, size, scroll, and present partitions in a specified order with specified visibility. Some typical examples of the use of partitions are:

- A single GDDM application lets the terminal user enter a set of data in a **real** partition while it processes another set previously entered in another partition.

  Real or emulated partitions can also be used to present different functions of your application on the one screen, or split the screen into two or more partitions so that you can compare related files; for instance, a source file in one partition and a compiler listing in another.

  Partitions can be used to avoid screen redraws. For example, you could have an alphanumeric menu in one partition, and some graphics in another. Interactions between the terminal user and the application through the alphanumeric menu, that do not mean any changes to the graphics, can take place without the graphics partition being redrawn.

  Unlike operator windows, **partitions cannot be manipulated by the terminal user**, **and cannot be used to run several independent applications**
- To reserve an area of the screen, for PF key information to be displayed at the bottom of the screen all the time that an application is running
- Depending on terminal-user interaction, the application could “pop up” a partition to overlap part of whatever is currently on the screen. The partition could contain, for example, help information, or a picture, or a panel containing input fields.

To split the screen, you must tell GDDM the size and position of each partition. Partitions need not be contiguous (you can leave empty space between them as in the example). In addition, you can overlap emulated partitions.

© Copyright IBM Corp. 1982, 1996

A simple partitioning example

Figure 120 contains a listing of a data-entry program that divides the screen into two equal parts. If real partitions are available, the terminal user types data into one part of the screen while the application program processes data that was previously typed in the other. A screen formatted by the program is shown in Figure 121 on page 460.

```
PARTEX1: PROC OPTIONS(MAIN);

DCL PTS_ARRAY(3) FIXED BIN(31);
DCL PTN_ARRAY(4) FIXED BIN(31);
DCL (CUR_PTN(1),BAD_PTN) FIXED BIN(31);
DCL CHAR936 CHAR(936);
DCL FILE_NO CHAR(3);
DCL ERROR_FLAG CHAR(1) INIT('0');
DCL I PIC'ZZ9';
DCL (TYPE,ATVAL,COUNT) FIXED BIN(31);

CALL FSINIT;

/* Define partition set grid                                  */
PTS_ARRAY(1)=5;                /* 5 rows in partition set    */   A
PTS_ARRAY(2)=1;                /* 1 col  in partition set    */   A
PTS_ARRAY(3)=0;                /* Real partitions if possible*/   A
/*      P-SET ID   NO. OF PARMS    PARAMETER ARRAY            */
CALL PTSCRT(1,        3,             PTS_ARRAY);                  B

/* Create partition at top of screen                          */
PTN_ARRAY(1)=1;         /* Starts in row 1 (of 5-row PTN-SET) */   C
PTN_ARRAY(2)=1;         /* Starts in col 1 (of 1-col PTN-SET) */   C
PTN_ARRAY(3)=2;                /* Depth is 2 rows            */   C
PTN_ARRAY(4)=1;                /* Width is 1 column          */   C
/*       PTN ID    NO. OF PARMS   PARAMETER ARRAY             */
CALL PTNCRT(1,          4,           PTN_ARRAY);                  D

/* Create display in top partition                            */
CALL CREATE_FIELDS;
CALL ASCPUT(1,32,'DATA ENTRY PROGRAM.  PARTITION 1');

/* Create partition in bottom of screen                       */
PTN_ARRAY(1)=4;         /* Starts in row 4 (of 5-row PTN-SET) */   E
CALL PTNCRT(2,      4,       PTN_ARRAY);                           F
/* Create display in bottom partition                         */
CALL CREATE_FIELDS;
CALL ASCPUT(1,32,'DATA ENTRY PROGRAM.  PARTITION 2');
```

Figure 120 (Part 1 of 3). Example of a program using partitions to control data entry

```
/* Dialog with operator                                        */
DO I=1 TO 999 UNTIL (ATVAL=3);
  RETRY:;
  CALL ASFCUR(4,1,1);
  CALL ASREAD(TYPE,ATVAL,COUNT);  /* Read from 'active' partn.  */  G
  CALL PTNQRY(1,1,CUR_PTN);    /* Which partn. was 'active'? */     H

  /* If input not from partn. that was bad, re-prompt operator*/
  IF (ERROR_FLAG='1')&(CUR_PTN(1)¬=BAD_PTN) THEN DO;                J
    CALL PTNSEL(BAD_PTN);         /* Make bad partition current */  K
    CALL ASCPUT(2,46,
              'PLEASE CORRECT INPUT FROM THIS PARTITION FIRST');
    GOTO RETRY;
  END;

  /* Check input                                                 */
  ERROR_FLAG='0';
  CALL INPUT_PROCESS;                                               M
  IF ERROR_FLAG='1' THEN DO;    /* Input was faulty            */
    BAD_PTN=CUR_PTN(1);         /* Record id. of faulty partn.*/    N
    CALL ASCPUT(2,48,
           'INPUT FAULTY FROM THIS PARTITION. PLEASE CORRECT');
    CALL PTNSEL(-1);  /* Force current partition to be active */    O
  END;
  ELSE CALL ASCPUT(2,34,'INPUT '||I||' PROCESSED SATISFACTORILY');
END;
CALL FSTERM;

/* Subroutine to create the input menu for each partition     */

CREATE_FIELDS: PROC;
CALL FSPCRT(1,20,110,0);
CALL GSPAT(1);
CALL GSAREA(1);
CALL GSLINE(100.0,0.0);
CALL GSLINE(100.0,100.0);
CALL GSLINE(0.0,100.0);
CALL GSENDA;
```

Figure 120 (Part 2 of 3). Example of a program using partitions to control data entry

```
CALL ASDFLD(1,1,34,1,32,2);     /* Protected 32-char field    */
CALL ASFCOL(1,1);               /* .. with a color of blue    */
CALL ASDFLD(2,3,26,1,48,2);     /* Protected 48-char blue fld.*/
CALL ASFCOL(2,2);               /* Message field is red       */
CALL ASDFLD(3,5,20,1,15,2);
CALL ASCPUT(3,15,'FILE NUMBER IS=');
CALL ASDFLD(4,5,36,1,3,0);
CALL ASDFLD(5,7,17,12,78,0);    /* Unprotected field 12 X 78  */
END CREATE_FIELDS;


/* Subroutine to check and process operator input            */

INPUT_PROCESS: PROC;
CALL ASCGET(4,3,FILE_NO);
IF (FILE_NO<'200')|(FILE_NO>'490') THEN ERROR_FLAG='1';
ELSE DO;
  CALL ASCGET(5,936,CHAR936);

  /*  .                           Code to copy              */
  /*  .                           operator's input          */
  /*  .                           data to disk file         */

  CALL ASCPUT(4,3,'   ');       /* Reset file number to empty */
END;
END INPUT_PROCESS;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
%INCLUDE ADMUPINP;

END PARTEX1;
```

Figure 120 (Part 3 of 3). Example of a program using partitions to control data entry

The program in Figure 120 on page 454 illustrates some of the concepts of partitioning:

Setting up a grid in which to define partitions

Your application can create several **alternative** logical screens on a device. Each logical screen is called a partition set, and only one can be shown to the terminal user at any time.

All the partitions must belong to a partition set. You create a partition set with the PTSCRT call and then define the partitions within it using the PTNCRT call.

The example creates a partition set at **B**. The main purpose of the PTSCRT call is to fit a conceptual grid over the screen. You use this grid in the PTNCRT call to define the position and size of each partition. This conceptual grid, rather than the hardware rows and columns, is used to specify further divisions of the screen, see Chapter 7, “Hierarchy of GDDM concepts” on page 107.

The first parameter of the PTSCRT call is the partition set identifier. It must be greater than 0, which is reserved for the default partition set created by GDDM

when you issue no PTSCRT call. The last parameter is an array with zero through four elements, the number of elements being given in the second parameter.

The program in Figure 120 on page 454 sets the values of the array elements in the statements marked **A**. The first element is the number of rows in the partition set grid, and the second the number of columns. The third element of the array defines the type of partitioning.

At **A** the program creates a partition set grid with five rows and one column, specifies that GDDM is to use real hardware partitioning if the device has it, otherwise to use emulation, and that the partitions are not to overlap. If the partitions had overlapped, GDDM would emulate the partitions, because real partitions cannot overlap.

Creating partitions

The first partition is created at **D**. The PTNCRT call has three parameters, similar to those of PTSCRT. The first one is the partition identifier; the third is an array of data containing elements in the range four through six; and the second specifies the number of elements in this array.

This program in Figure 120 on page 454 uses a four-element array, setting the values of its elements in the statements marked **C**. The first two elements are the row and column position, on the partition-set grid, of the top left-hand corner of the partition. The other two are its depth and width, in partition set grid units. For information about the fifth parameter, see the *GDDM Base Application Programming Reference* book. The sixth parameter defines the visibility of the partition. A 0 means invisible, and a 1 means visible (the default). A use of the visibility parameter is examined later in this chapter.

The program places the top left-hand corner of the first partition in the top left-hand corner of the screen, and makes the partition as wide as the screen and two-fifths of its depth.

The second partition is created at **F**. It uses the same array of values as **D**, except that the top of the partition is positioned three-fifths of the way down the screen. The statement **E** alters the first element of the array parameter to specify this.

Once a partition has been created, you can treat it like the complete screen in a nonpartition application because a GDDM page occupies a complete partition.

Current partition sets, partitions, and pages

Sometimes you may need to create more than one partition set. If you do, the latest one becomes current when you create it. But you can make any partition set current with the PTSSEL call, for example:

```
CALL PTSSEL(2); /* Make partition set 2 current */
```

A partition belongs to the partition set that is current at the time of its creation.

As with partition sets, a partition is made current when it is created. Subsequently, the current partition can be changed, by device input, or by using the PTNSEL call. You can make current any existing partition within the current set, using PTNSEL:

```
CALL PTNSEL(3);  /* Make partition 3 current */
```

The program in Figure 120 on page 454 contains an example of this at **K**.

When you explicitly or implicitly create a GDDM page, it becomes associated with the partition then current. The example explicitly creates one page in each partition. A page becomes current when created. You can use FSPSEL to make a different page current within the partition.

Input/Output

When the screen is partitioned, it is particularly important to understand how the GDDM input/output calls such as ASREAD work.

When an input/output call is executed, GDDM sends the changes from the current pages in all the partitions in the current partition set to the terminal. It then waits. When the terminal user responds (for instance, by pressing the ENTER key) GDDM receives an interrupt together with input data. The wait is thereby satisfied and GDDM allows the application program to resume execution.

With hardware partitioning, the keyboard does not lock after the terminal user has responded. The terminal prevents the user from typing further data into the partition that the cursor was in when the user responded, but it allows typing in another partition. This means that you can enter data in one partition at the same time that the application is processing data entered in another.

In the typical case, the terminal user would complete the entry of data into one partition, press the ENTER key, and then start typing into the other partition. But although data entry can continue, nothing can be read in until the application executes a further ASREAD. GDDM ensures synchronization between the application program and the terminal user's actions by enforcing this sequence:

```
Program calls ASREAD and waits
Operator generates interrupt
   Then, normally, the program processes the input
   while the terminal user enters more data.
Program calls ASREAD and waits
Operator generates interrupt
   Then the program processes the new input
   while the terminal user enters more data.
Program calls ASREAD and waits
Operator generates interrupt
     .
     .
     .
```

If partitioning is being emulated, the keyboard is locked after each input transmission. You cannot therefore enter data until the application has finished processing your last input.

A GDDM input/output call updates all partitions in the display. In Figure 120 on page 454, the call to ASREAD at **G** creates two partitions on the screen, with a data-entry display in each one. But GDDM **updates** screens rather than rewriting them completely, so later executions of the ASREAD change only the data altered by the program. The rest of the screen remains unchanged. In the program in Figure 120 on page 454, each ASREAD reinitializes the partition from which the last error-free input was received by transmitting an empty menu.

For interactive graphics applications, logical input devices must be enabled for **each** partition.

Active and current partitions

When partitions are displayed on the screen of a device that supports real partitions, the one containing the cursor is said to be **active**. The terminal user can make a different partition active by moving the cursor into it. When real partitions are used (on the 3290, 8775, or 3193) this means using the partition-jump key.

When the terminal user causes an interrupt when using real partitions, the program receives data only from the active partition. When GDDM receives the input, it makes the partition from which it was received current. So if, for instance, the cursor is in partition 2 when the end user presses the ENTER key, partition 2 becomes the current partition after the ASREAD, even if partition 1 was current before the ASREAD.

You can discover which is the current partition at any time, and its size and position, by a PTNQRY call, like the one at **H**. In the first and second parameters, you tell GDDM which of five possible values are to be returned: the first parameter specifies which is the first value to be returned, and the second, how many are to be returned. The five possible values are: the identifier of the current partition; the row and column positions on the partition set grid of the current partition's top left-hand corner; and the depth and width of the current partition in partition-set grid units. The third parameter is an array in which GDDM returns the specified values. Statement **H** queries just the partition identifier.

Unless you force a different action, an ASREAD does **not** make the current partition become the active one. This would cause the cursor to jump to the current partition, which could inconvenience the terminal user, who might be typing into another partition. GDDM allows the partition with the cursor in it to remain active.

There are two ways of forcing a different action. Firstly, if you execute a PTNSEL call to select a partition other than the one that was current after the previous ASREAD, the new current partition becomes the active one at the next ASREAD:

```
CALL PTNSEL(3);  /* Make partition 3 current and force it to be */
                 /* active at next ASREAD                       */
```

In other words, if you change the current partition between ASREADs, GDDM makes the new current one become the new active one.

Secondly, you can force the partition that was current after the previous ASREAD to become the active one. This is a useful way of drawing the user's attention to faulty input. In this case, you should issue this specialized form of the PTNSEL call:

```
CALL PTNSEL(-1); /* At next ASREAD, force partition current at  */
                 /* that time to become active.                 */
```

In summary, partitions are recorded as current by GDDM, and as active by the terminal hardware. The terminal user can make a partition active just by moving the cursor into it, but GDDM only discovers which one is active when there is an interrupt. The active and current partitions are therefore not typically the same.

You should avoid assuming that a partition that was inactive at the time of the last ASREAD can be updated by your program. It may contain data entered by the terminal user either before or after that ASREAD. This input is lost if your program overwrites it.

Handling terminal-user errors

After the ASREAD at **G**, the program in Figure 120 on page 454 queries which partition was active and is therefore now current, at **H**. It then tests a flag to determine whether input to correct an earlier error was expected, and if so, whether the latest input is from the partition that contained the error. These tests are carried out by statement **J**.

If corrective input is expected, but the input in fact came from the other partition, then the bad partition is selected at **J**, and an error message is put into it. Because this statement causes a different partition to become current, GDDM causes this partition to become the active one at the next ASREAD.

If corrective input is not expected, the program calls a subroutine at **M** to check and process the input. If this subroutine finds an error, it sets the error flag. In this case, the program records which is the bad partition, at **N**, puts an error message into it, and, at **O**, forces it to be active after the next ASREAD.

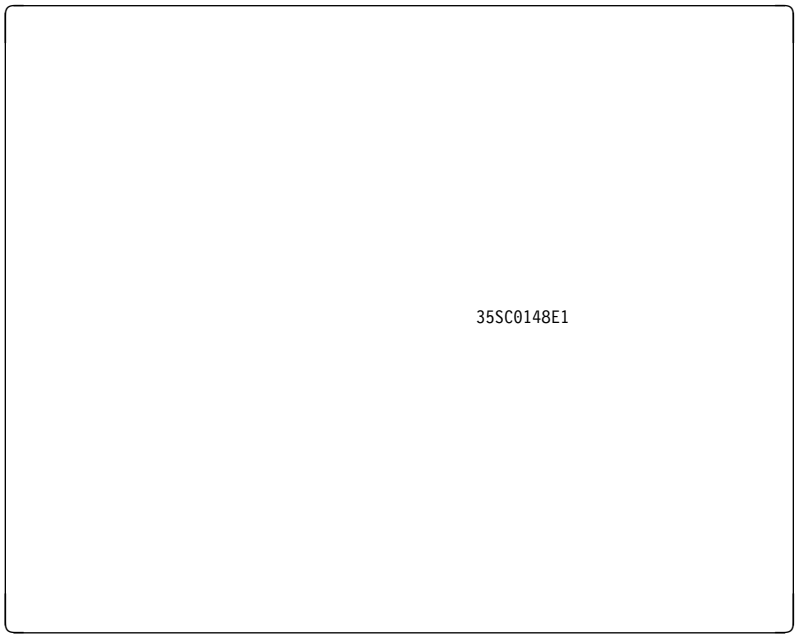


Figure 121. Screen formatted by the PARTEX1 partitioning program

Some other things you can do with partitions

The next two subsections cover some things that you can do with emulated partitions:

- Change their visibility
- Overlap them, and alter their viewing priority.

Visible and invisible partitions

The following example code is a program skeleton that illustrates how you can use visible and invisible partitions to organize screen layout, for example, for the data-entry panel for an ICU-like program.

```
PARTVIS: PROC OPTIONS(MAIN);

DCL (TYPE,MOD,COUNT) FIXED BIN(31);

/* Partition set parameters -      rows columns control overlap */
DCL SET_ARRAY(4) FIXED BIN(31) INIT(10,    16,     1,      1);

/* Partition parameters  -  row column depth width dev visibility */
DCL P1(6) FIXED BIN(31) INIT(1,   1,     2,   16,   -1,    1);
DCL P2(6) FIXED BIN(31) INIT(9,   1,     2,   16,   -1,    1);
DCL P3(6) FIXED BIN(31) INIT(3,   1,     6,    4,   -1,    1);
DCL P4(6) FIXED BIN(31) INIT(3,   5,     6,    4,   -1,    1);
DCL P5(6) FIXED BIN(31) INIT(3,   9,     6,    4,   -1,    1);
DCL P6(6) FIXED BIN(31) INIT(3,  13,     6,    4,   -1,    1);
DCL P7(6) FIXED BIN(31) INIT(3,   1,     6,    4,   -1,    0);
DCL P8(6) FIXED BIN(31) INIT(3,   1,     6,    4,   -1,    0);
DCL P9(6) FIXED BIN(31) INIT(3,   1,     6,    4,   -1,    0);
```

Figure 122 (Part 1 of 3). Skeleton of program changing visibility of partitions to control data entry

```
CALL FSINIT;

CALL PTSCRT(1,4,SET_ARRAY);                                          A

CALL PTNCRT(1,6,P1); /* Partition 1 - heading & message area */      B
/*     .
/*     .
CALL PTNCRT(2,6,P2); /* Partition 2 - PF key area           */      B
/*     .
/*     .
CALL PTNCRT(3,6,P3); /* Partition 3 - command area          */      B
/*     .
/*     .
CALL PTNCRT(4,6,P4); /* Partition 4 - X values              */      B
/*     .
/*     .
CALL PTNCRT(5,6,P5); /* Partition 5 - Y1 data               */      B
/*     .
/*     .
CALL PTNCRT(6,6,P6); /* Partition 6 - Y2 data               */      B
/*     .
/*     .
CALL PTNCRT(7,6,P7); /* Partition 7 - Y3 data               */      B
/*     .
/*     .
CALL PTNCRT(8,6,P8); /* Partition 8 - Y4 data               */      B
/*     .
/*     .
CALL PTNCRT(9,6,P9); /* Partition 9 - Y5 data               */      B
/*     .
/*     .

CALL ASREAD(TYPE,MOD,COUNT);          /* Display first panel */      C

CALL PTNSEL(3);                       /* Select partition 3  */      D
P3(6) = 0;                            /* Set to invisible    */      D
CALL PTNMOD(6,1,P3);                  /* Modify partition    */      D

CALL PTNSEL(4);                       /* Select partition 4  */      E
P4(2) = 1;                            /* New column position */      E
CALL PTNMOD(6,1,P4);                  /* Modify partition    */      E

CALL PTNSEL(5);                       /* Select partition 5  */      F
P5(6) = 0;                            /* Set to invisible    */      F
CALL PTNMOD(6,1,P5);                  /* Modify partition    */      F
```

Figure 122 (Part 2 of 3). Skeleton of program changing visibility of partitions to control data entry

```
CALL PTNSEL(6);                      /* Select partition 6  */   G
P6(6) = 0;                           /* Set to invisible    */   G
CALL PTNMOD(6,1,P6);                 /* Modify partition    */   G

CALL PTNSEL(7);                      /* Select partition 7  */   H
P7(2) = 5;                           /* New column position */   H
P7(6) = 1;                           /* Set to visible      */   H
CALL PTNMOD(6,1,P7);                 /* Modify partition    */   H

CALL PTNSEL(8);                      /* Select partition 8  */   I
P8(2) = 9;                           /* New column position */   I
P8(6) = 1;                           /* Set to visible      */   I
CALL PTNMOD(6,1,P8);                 /* Modify partition    */   I

CALL PTNSEL(9);                      /* Select partition 9  */   J
P9(2) = 13;                          /* New column position */   J
P9(6) = 1;                           /* Set to visible      */   J
CALL PTNMOD(6,1,P9);                 /* Modify partition    */   J

CALL ASREAD(TYPE,MOD,COUNT);         /* Display other panel */   K

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
%INCLUDE ADMUPINP;
CALL FSTERM;
END;
```

Figure 122 (Part 3 of 3). Skeleton of program changing visibility of partitions to control data entry

A program based on the skeleton in Figure 122 on page 461 produced the panels in Figure 123 on page 465 and Figure 124 on page 465. All that is added to the program skeleton to produce the screen layouts in the two figures is the code to label each partition and draw a line around its border. In practice, for an ICU-like application, the partitions would contain procedural or mapped alphanumerics.

The example in Figure 122 on page 461 creates nine partitions, each to contain a logical area of the screen. Only five of the nine are initially defined as visible, and are used to produce the first panel. The visibility of some of the nine partitions is then altered, and the results displayed as the second panel.

The PTSCRT call at **A** defines the partition-set grid as being 10 rows by 16 columns, using the parameters in SET_ARRAY. The fourth parameter of the array specifies that the partitions in the partition set can overlap. However, you do not see any overlapping partitions in the output displayed by the program. This is because, where partitions overlap, only one of them is specified as visible.

At the PTNCRT calls marked **B**, the program creates nine partitions for a heading and message area, a PF key area, a command area, an x data area, and five y-data entry areas. The PTNCRT calls use the parameters in arrays P1 through P9. The sixth parameter of each array specifies the initial visibility of each partition. A value of 1 makes it visible, while a value of 0 makes it invisible. Partitions 1 through 6 are initially defined as visible, while partitions 7 through 9 are initially

invisible. This is assuming that partitions 1 through 6 are the only ones that you want to be seen in the first panel displayed using the ASREAD at **C**.

Assume that in the second panel, you want to display partitions 1, 2, 4, 7, 8, and 9. You do not have to do anything to partitions 1 and 2 for them to be displayed again.

At **D**, as you no longer want partition 3 to be shown, you must first make it current, using the PTNSEL call, and then set its visibility parameter to 0 (invisible). Then modify the current partition using a PTNMOD call. The call has three parameters:

- The number of the first element in the third parameter. It must be in the range 1 through 6.
- The number of elements in the third parameter.
- An array of up to six elements, containing the attributes for the current partition. The example here uses the array that was originally used to create the partition.

Using PTNSEL and PTNMOD, you can alter the attributes of the remaining partitions as follows:

- Partition 4 is to be displayed in the second panel, but with its top-left-hand corner in column 1.
- Partitions 5 and 6 are not to appear in the second panel, so their visibility attribute is set to 0.
- Partitions 7, 8, and 9 were originally defined as invisible, with their top-left-hand corners in row 3 and column 1. At **H**, **I**, and **J** their positions are changed to 5, 9, and 13 respectively, and they are made visible.

The advantages of constructing panels using visible and invisible partitions are:

- You can easily produce several variations of the same panel
- You can take advantage of the whole screen area.

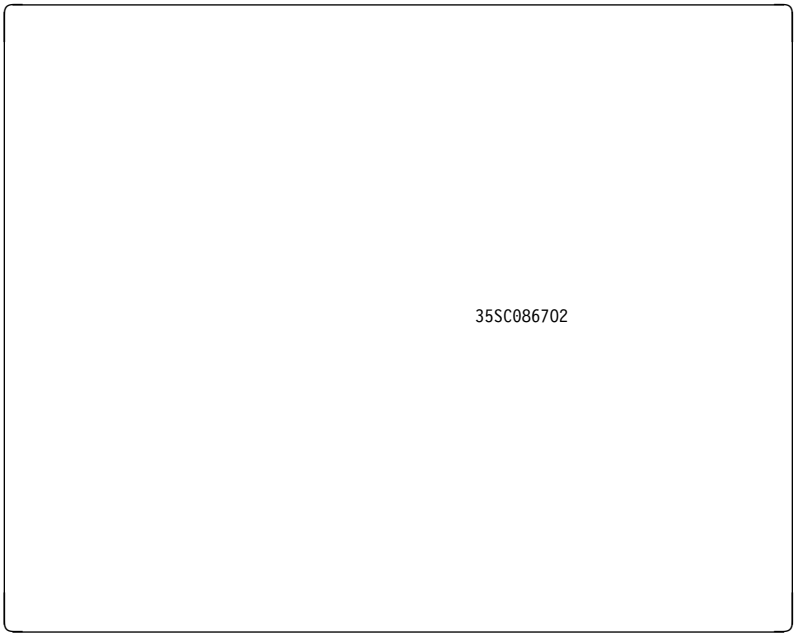


Figure 123. First panel using visible and invisible partitions

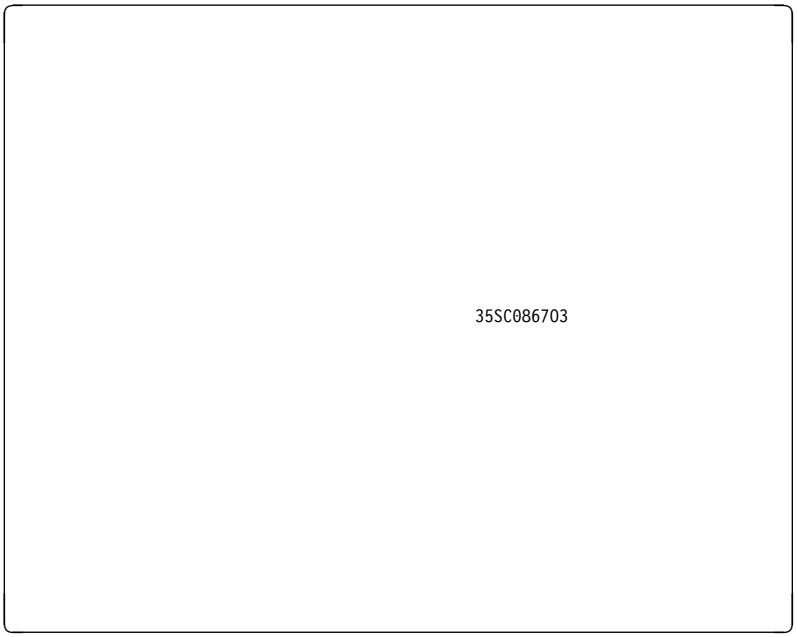


Figure 124. Second panel using visible and invisible partitions

Overlapping partitions

You can overlap partitions. Partitions are opaque, so the part of a partition that is overlapped by another partition is completely obscured by the top partition.

The next example contains the skeleton code to produce a partition that **overlaps** another partition:

```
PARTLAP: PROC OPTIONS(MAIN);

DCL (TYPE,MOD,COUNT) FIXED BIN(31);

/*  Partition set parameters -     rows columns control overlap  */
DCL SET_ARRAY(4) FIXED BIN(31) INIT(10,    16,     1,     1);

/*  Partition parameters - row column depth width dev visibility */
DCL P1(6) FIXED BIN(31) INIT(1,   1,    10,   16,  -1,    1);

DCL P2(6) FIXED BIN(31) INIT(5,   3,     6,   11,  -1,    1);

CALL FSINIT;

CALL PTSCRT(1,4,SET_ARRAY);                                        A

CALL PTNCRT(1,6,P1);                                               B
        .
        .
        .

CALL PTNCRT(2,6,P2);                                               C
        .
        .
        .

CALL ASREAD(TYPE,MOD,COUNT);

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
%INCLUDE ADMUPINP;
CALL FSTERM;
END PARTLAP;
```

A program based on the above skeleton program produced the screen output shown in Figure 125 on page 467. All that is added to the program is the alphanumeric code for the panel in partition 1, the graphics calls to produce the chart in partition 2, and the code to draw a line around the border of each partition.

The PTSCRT call at **A** defines the partition-set grid, using the parameters in SET_ARRAY.

The PTNCRT call at **B** creates partition 1, using the parameters in P1_ARRAY. This partition fills the screen.

The PTNCRT call at **C** creates partition 2, using the parameters in P1_ARRAY. These parameters place the top-left-hand corner of partition 2 in row 5 and column 3.

The advantages of overlapping partitions are:

- You can show a number of partitions on the screen, at the same time, but highlight one or more partitions by placing them on top of the others.
- You can show more of the underlying partitions than is possible with nonoverlapping partitions.

An example of the use of overlapping partitions is to associate each partition with each logical function of your program. The program would change the viewing order to let the terminal user access the partition associated with the function that is wanted. The next section tells you how your program can alter the viewing order.

If you specify on the PTSCRT call that partitions can overlap, you always get emulated partitions (even when the partitions do not actually overlap) on all devices including those that support real partitions.

Partitions are also always emulated when user control or operator windows are available.

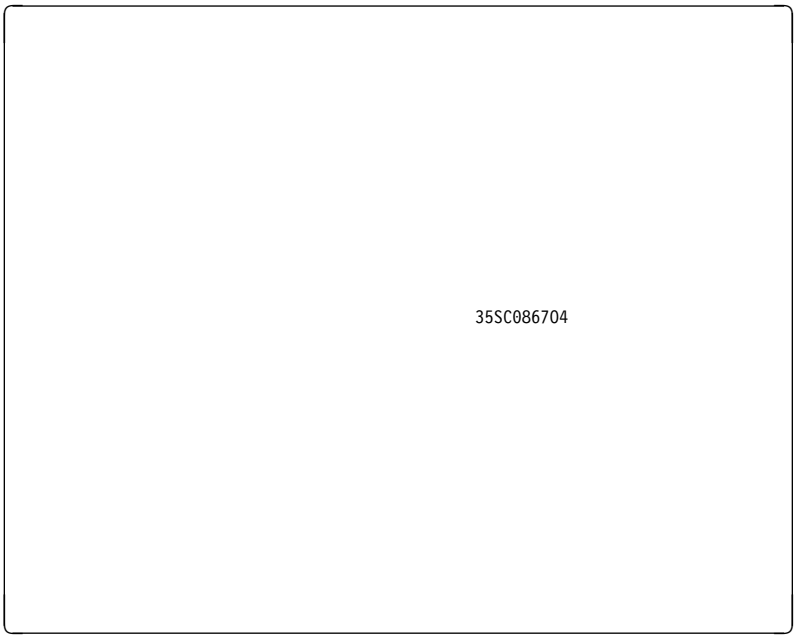


Figure 125. Output of program with overlapping partitions

Prioritizing partitions

When you first create a number of overlapping partitions, the viewing order depends on the order in which you create the partitions. The partition that you create first is at the bottom of the viewing order, and the partition that you create last is at the top. On the display screen, each partition appears on top of the partitions that are below it in the viewing order. Some partitions may be hidden behind other partitions, or may have their visibility attribute set to invisible, but they are still present in the viewing order.

You can change the priority of some or all of the partitions in the viewing order, using the call PTSSPP. This call lets you specify an array of identifiers of partitions whose priorities are to be adjusted by placing them as neighbors to one of the other partitions in the viewing order.

For example, suppose a partition set has the following seven partitions in descending order:

TOP 7, 6, 5, 4, 3, 2, 1 BOTTOM

If you wanted to take partitions 7, 2, and 1, and change their order of viewing so that they are after partition 5 and before partition 4, like this:

TOP 6, 5, **1, 7, 2,** 4, 3 BOTTOM

you would issue the following call:

```
DCL PRI_ARRAY (3) FIXED BIN(31) INIT(2,7,1);

CALL PTSSPP(1,4,3,PRI_ARRAY); /* Change partition viewing order */
```

The parameters are as follows:

- The first parameter specifies whether the partitions in the array in the final parameter are to be placed in descending or ascending order from the reference partition. (The reference partition is the reference point in the viewing order about which the reordering of the partitions is to take place. It is specified in the second parameter. In the example, it is partition 4.)

  The first parameter can have these values:

  -1 Descending order. The partitions in the array are placed behind the reference partition.

  1 Ascending order (as in the example). The partitions in the array are placed in front of the reference partition.

- The second parameter contains the identifier of the reference partition relative to which the reordering is to take place. It can have a value of –1, the effect of which depends on whether you set the first parameter to ascending or descending order:

  Descending The first partition in the array becomes the top partition in the viewing order, and the rest of the partitions in the array are placed behind it.

  Ascending The first partition in the array becomes the bottom partition in the viewing order, and the rest of the partitions in the array are placed in front of it.

- The third parameter contains the number of elements in the array in the final parameter
- The final parameter is an array of identifiers of partitions whose priorities are to be adjusted relative to the reference partition. Any element of the array can contain a value of –1, which causes all further elements to be ignored.

The reordering process takes the first partition in the array and places it above or below the reference partition in the viewing order, depending on the order specified in the first parameter. It then takes the second partition in the array and places it above or below the first partition, and so on, until all the elements of the array parameter have been processed, or until a value of –1 is found in the array.

The following programming example creates five overlapping partitions. Each partition is filled with a shading pattern, and some alphanumerics. The initial output displayed by the program is shown in Figure 127 on page 471. Initially, the cursor is displayed in partition 5. Partition 5 overlaps partition 4, partition 4 overlaps partition 3, and so on. If the terminal user moves the cursor into the visible part of, for example, partition 3, and presses the ENTER key (or some other interrupt-generating key) the program uses the PTSSPP call to bring that partition

to the top of the viewing order. If the user then moves the cursor into, for example, partition 5, and presses the ENTER key, partition 3 is replaced behind partition 2 and in front of partition 4, and partition 5 is brought to the top. Pressing the PF12 key terminates the application.

```
FOLDERS: PROC OPTIONS(MAIN);
DCL PARMS(4) FIXED BIN(31) INIT (0,0,1,1);
DCL PARMS1(4) FIXED BIN(31) INIT (1,1,15,40);
DCL PRIORITY(5) FIXED BIN(31) INIT(5,4,3,2,1);
DCL (TYPE,MOD,COUNT) FIXED BIN(31);
DCL COLOR FIXED BIN(31) INIT(0);
DCL PATTERN FIXED BIN (31) INIT(0);
CALL FSINIT;
CALL PTSCRT(1,4,PARMS);   /* Emulate partitions - they overlap */
CALL PTNCRT(1,4,PARMS1);  /* Top left partition */
COLOR=1;
PATTERN=1;
CALL COLOR_FOLDER;
CALL ASCPUT(1,8,'Folder 1');
CALL ASCPUT(2,79,(79)'A');

PARMS1(1)=5;
PARMS1(2)=11;
CALL PTNCRT(2,4,PARMS1);
COLOR=2;
PATTERN=2;
CALL COLOR_FOLDER;
CALL ASCPUT(1,8,'Folder 2');
CALL ASCPUT(2,79,(79)'B');

PARMS1(1)=9;
PARMS1(2)=21;
CALL PTNCRT(3,4,PARMS1);
COLOR=3;
PATTERN=3;
CALL COLOR_FOLDER;
CALL ASCPUT(1,8,'Folder 3');
CALL ASCPUT(2,79,(79)'C');
```

Figure 126 (Part 1 of 3). Example of program with controlled viewing order of partitions

```
PARMS1(1)=13;
PARMS1(2)=31;
CALL PTNCRT(4,4,PARMS1);
COLOR=4;
PATTERN=4;
CALL COLOR_FOLDER;
CALL ASCPUT(1,8,'Folder 4');
CALL ASCPUT(2,79,(79)'D');

PARMS1(1)=17;
PARMS1(2)=41;
CALL PTNCRT(5,4,PARMS1);            /* Bottom right partition */
COLOR=5;
PATTERN=5;
CALL COLOR_FOLDER;
CALL ASCPUT(1,8,'Folder 5');
CALL ASCPUT(2,79,(79)'E');

DO I=1 TO 99;
CALL ASREAD(TYPE,MOD,COUNT);
CALL PTNQRY(1,1,PARMS);
IF MOD>11 THEN GOTO ENDIT;
CALL PTSSPP(-1,-1,5,PRIORITY); /* Restore original order       */
CALL PTSSPP(-1,-1,1,PARMS);    /* Put selected partition at top*/
END;

COLOR_FOLDER: PROC;
CALL ASDFLD(1,1,2,1,8,0);
CALL ASDFLD(2,3,2,2,39,0);
CALL GSSEG(0);
CALL GSCOL(COLOR);
CALL GSPAT(PATTERN);
CALL GSMOVE(0,0);
CALL GSAREA(1);
CALL GSLINE(0,100);
CALL GSLINE(100,100);
CALL GSLINE(100,0);
CALL GSLINE(0,0);
CALL GSENDA;
END COLOR_FOLDER;
```

Figure 126 (Part 2 of 3). Example of program with controlled viewing order of partitions

```
ENDIT:;
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINP;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END FOLDERS;
```

Figure 126 (Part 3 of 3). Example of program with controlled viewing order of partitions

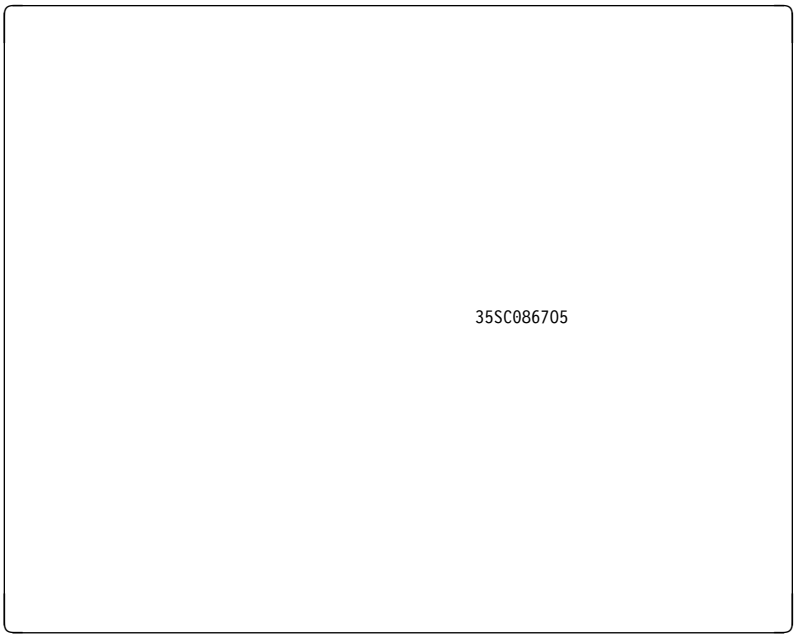


Figure 127. Output from program with prioritized partition viewing

Querying the priority of overlapping partitions

There are two calls that you can use to query the priority of partitions in the current partition set.

In the last section, the PTSSPP call was used to change the viewing priority of a specified array of partition identifiers relative to a specified reference partition identifier. The corresponding query call PTSQPP returns an array of partition identifiers relative to a specified reference partition identifier. For example, here is a typical call, that returns the identifiers of the three partitions that are above partition 5 in the viewing order:

```
DCL PRI_ARRAY (3) FIXED BIN(31);

CALL PTSQPP(1,5,3,PRI_ARRAY); /* Query partition viewing order  */
```

The parameters are as follows:

- The first parameter specifies whether the array in the final parameter is to return the identifiers of partitions in descending or ascending order from the reference partition. The possible values are:

-1 Descending order

1 Ascending order.

- The second parameter specifies the identifier of the reference partition that the query relates to. It can have a value of –1, the effect of which depends on whether you set the first parameter to ascending or descending order:

  Descending The first partition in the array is the top partition in the viewing order, and the rest of the partitions in the array are those that are behind it.

  Ascending The first partition in the array is the bottom partition in the viewing order, and the rest of the partitions in the array are those that are in front of it.

- The third parameter contains the number of elements to be returned in the array in the final parameter.
- The final parameter is an array that holds the returned identifiers of partitions that descend or ascend from the reference partition.

There is another query call, PTSQPI, that returns the identifiers of either all partitions in the current partition set, or just the invisible ones. Here is a typical call:

```
DCL PRI_ARRAY (7) FIXED BIN(31);

CALL PTSQPI(1,7,PRI_ARRAY);  /* Query all partition identifiers */
```

The parameters are as follows:

- The first parameter specifies the type of partition that you want information returned about:

  1 All partitions

  2 All invisible partitions.

- The second parameter specifies the number of elements in the array in the final parameter.
- The third parameter is the name of an array in which GDDM returns the requested information.

Other calls that operate on partitions and partition sets

PTSDEL Deletes a specified partition set.

PTSQPN Returns the total number of partitions (all or just the invisible ones).

PTSQRY Queries the attributes of the current partition set.

PTSQUN Returns a unique unused partition set identifier.

PTNDEL Deletes a partition

PTNMOD Modifies the attributes of the current partition.

PTNQRY Queries the attributes of the current partition.

PTNQUN Returns a unique unused partition identifier for use with a subsequent PTNCRT.

For more details of the above calls, see the *GDDM Base Application Programming Reference* book.

Large and small pages

This section tells you how to display amounts of data that are larger than the screen, and how to fill up the screen with small amounts of data. Some of the techniques it describes need hardware function that is available only on specific types of terminal, but others can be used on all terminals.

If you create a page that is too deep to be displayed all at once, scrolling may help. This technique treats the screen, or a rectangular area of the screen, like a window that moves up and down, or from side to side, in front of the page. (Or, as the screen does not actually move, it may help you to think of the page as moving up and down, or from side to side, behind the screen.) Data that falls within this **page window** is displayed, and other data can be displayed by repositioning the page window.

Another possible solution, applicable for alphanumeric data when the page is too wide or too deep, is to use smaller characters than normal. This is a possibility only on the 3290, because this is the only supported terminal with a variable cell size.

If you have the converse problem of displaying only a small amount of data, you can increase the cell size on the 3290 to help fill the screen.

Scrolling

Some types of terminal have their own scrolling function. Your program can send a large page to the terminal, and the user can use special scrolling keys to select the part to be displayed. The page can contain alphanumeric and graphics data. The amount of data on the page is limited by the storage capacity of the terminal, rather than the size of the screen.

In addition, GDDM provides a software scrolling function that lets your program select which part of the current page is to be displayed. This is supported on all types of display terminal.

The 3193 Display Station has vertical and horizontal hardware scrolling. The other terminals with their own scrolling function allow vertical scrolling only. GDDM software scrolling, however, allows both vertical and horizontal scrolling: the page window can be moved up and down and from side to side.

Your program positions the page window with the FSPWIN call. For example:

```
CALL FSPWIN(20,1,-1,-1);  /* Put row 20 at top of page window */
```

The FSPWIN call has two functions, of which scrolling is one. The other is to set the depth and width of the page window, and is explained in “Variable character size” on page 474. The parameters are:

- The row that GDDM is to position at the top of the page window–row 20 in the example.
- The column that is to be at the left-hand edge of the page window–column 1 in the example.
- The depth and width of the page window. When scrolling, these are both set to –1.

You can use FSPWIN to provide scrolling on terminals that do not have it as a hardware function. In a typical application, specific user actions, usually the pressing of PF keys, are defined as commands that mean scroll up or down or from side to side by a certain amount, or scroll to the top, bottom, or side of the page. The application would use FSPWIN to implement these commands.

Another use for FSPWIN is to place the window in a particular position over the page, independently of any action by the terminal user. For instance, the application might need to draw the user's attention to a particular line by putting it at the top of the page window. FSPWIN is useful for this purpose on terminals with hardware scrolling, and on those without.

If the terminal user uses the hardware scrolling function, the position of the page window changes without any indication being given to your program. Suppose that your program sends a page to the terminal after positioning the window at line 1. This line appears at the top of the page window. Suppose, then, that the user moves line 20 to the top of the page window with the hardware scrolling keys, and presses the ENTER key to send the page back to your program. Because the program is not notified of the change, it still considers the position of the page window to be line 1. If it resends the page to the terminal, line 1 and not line 20 is then at the top of the page window.

Putting a specified row at the top of the page window would sometimes result in the bottom row of the page being above the bottom of the page window. The page window space below the last row of the page would then be wasted. In these cases, the hardware (or if scrolling is being emulated, GDDM), positions the page window to use this space. Usually, this means arranging for the bottom row of the page to be on the bottom line of the page window. The row that you specified in the first parameter of FSPWIN is then displayed some way down the page window rather than on the top line.

Similarly, if you try to put the top line of the page some way down the page window using hardware scrolling, the hardware actually positions it at the top of the page window.

The alphanumeric cursor must always be within the page window. If you move the page window to a position that leaves the cursor outside, GDDM moves the cursor to within the window. Conversely, if you move the cursor to outside the window, GDDM repositions the window to re-include it. In these cases, GDDM generally arranges for the cursor to appear on the top line of the page window.

Variable character size

GDDM varies the cell size on the IBM 3290 Information Panel according to the number of rows and columns to be displayed. Within limits, it selects the cell width that best fits the number of columns to the screen width, and the cell depth that best fits the number of rows to the screen depth.

You can specify the number of rows and columns in a page in the FSPCRT or MSPCRT call, as explained in “The page and page window” on page 111. For example:

```
CALL FSPCRT(1,60,80,0);
```

creates a page 60 rows deep by 80 columns wide. For MSPCRT, GDDM takes the page size from the mapgroup if you do not specify it explicitly.

GDDM tries to fit the page onto the screen, by choosing the largest cell depth that allows 60 rows to be displayed and the largest cell width that allows 80 columns to be displayed, unless you specify a page window of smaller size than the page.

While the page is still empty, you can execute an FSPWIN call that specifies one or both of the window depth and width sizes in the third and fourth parameters. If you do, then GDDM chooses the cell size that best fits the window, rather than the complete page, to the screen. For instance, this call:

```
        /* Row  Column  Depth  Width */
CALL FSPWIN(1,     1,    30,    -1);
```

causes GDDM to select the cell depth that best fits 30 rows onto the screen, while leaving the cell width unchanged. The first two parameters still specify the page window position, so in this example, the window is positioned at row 1.

If the two example calls were executed one after the other, GDDM would create a page 60 rows deep and 80 columns wide, and a window that displays 30 of the rows, and all the columns, and is initially positioned over the top half of the page.

As hardware scrolling is being used on a 3290, there is no lateral scrolling, so the width of the window must be not less than the width of the page. For the same reason, the number of columns to be displayed must be no greater than what would all fit onto the screen if the smallest cell width were used. This restriction applies whether the number of columns is the width of the page, as defined by FSPCRT (or MSPCRT), or the width of the page window, as defined by FSPWIN.

Once the cell size for a page has been fixed, it cannot be altered. It is fixed in one of two ways: by executing an FSPWIN call that specifies the page depth or width or both, or by putting some data into the page.

If you put some data into a page without executing an FSPWIN call, GDDM attempts to fit the complete page onto the screen, using as large a cell size as possible. If it cannot display the complete page, it selects the minimum cell size, and displays as many rows as possible, using a page window positioned at row 1 of the page.

If you do not specify the number of rows or columns in a page, GDDM assumes device-dependent numbers. These are such that if you do not specify a page window depth or width, the resulting cell size width or depth, or both, is the default for the device. This means that if you specify neither a page size nor a page window size, GDDM uses the default cell size for the device.

You can still execute FSPWIN calls after the cell size has been fixed, but the page depth and width must both be specified as –1. The call's function then is just scrolling.

Cell sizes of the 3290: The minimum, default, and maximum width and depth of cells on the 3290 are shown below. The loadable cell size, that is, the size that the terminal uses for programmed symbols is also shown.

Cell size in pixels	Width	Depth
Minimum	6	12
Default	6	12
Loadable	9	16
Maximum	16	31

The terminal scales its own hardware characters to fit cells the width and depth of which are no less than the minimum size and no more than the loadable size. Cells that are wider or deeper, or both, than the loadable cell size may contain characters that are 9 pixels deep or 16 pixels wide, or both, with the rest of the cell empty.

Image symbol sets of any size up to 9 pixels by 16 may be loaded into a 3290, using the PSLSS call. However, if the screen cell size is less than the symbol size, only part of each symbol can be seen.

If your program uses graphics on a 3290, the cell size must not exceed 9 pixels by 16. Therefore, cell sizes that are between 9 pixels by 16 and 16 pixels by 31 are for the use of alphanumerics programs only.

The largest cell size that you can get on an IBM 3290 with an FSPCRT call is the loadable size. To get a cell size between this and the maximum, you need to create a window with FSPWIN.

You may be wondering how to ensure that your program uses cells of a valid size. The answer is to use the FSQURY command, as described in the *GDDM Base Application Programming Reference* book. This supplies much information about the device, including the depth and width of the screen in pixels, and the number of rows and columns it can display at the minimum and maximum cell sizes.

Effects on graphics of scrolling and variable cell size

When a page is scrolled by the terminal user with the terminal's hardware facility, graphics are scrolled along with the alphanumerics. The GDDM software function scrolls graphics similarly on all terminals.

Partitioning with scrolling and variable cell size

The program shown in Figure 128 on page 477 combines partitioning with some of the functions described in “Large and small pages” on page 473. It is intended to run on the IBM 3290 Information Panel. It creates two partitions, both scrollable, with a different cell size in each. The considerations described in “Variable character size” on page 474 apply to each partition.

It is intended to display two data sets, one containing a program's source code as entered by the programmer, and the other a compiler listing of the program. It handles a total of 80 lines of source code and 35 lines of listing, using a 25-line scrollable page window for each. Typical output is shown in Figure 129 on page 479.

The partition set is created at **A**, with a grid one column wide and nine rows deep. The top four rows are used for the source file partition, and the bottom four for the listing file partition. The two partitions are created at **B** and **G**.

The page for the source file display is created at **C**, with 83 rows and 82 columns. A window 25 rows deep is placed over this page at **D**. GDDM selects a cell width and depth that best fills the window.

The page for the listing file display is created at **H**, with 38 rows and 123 columns. A 25-row window is placed over this page at **I**. GDDM again selects a cell width and depth to best fill the window.

The user can use the IBM 3290 Information Panel's hardware scrolling facility to move the two page windows. No provision is made for software scrolling.

Up to 80 source records are read in statements **E** to **F** and stored on the GDDM page corresponding to the first partition. Up to 35 listing records are read in statements **J** to **K** and stored on the page corresponding to the second partition. Both partitions are sent to the terminal at **L**.

```
PARTEX2: PROC OPTIONS(MAIN);

DCL PTS_ARRAY(3) FIXED BIN(31); /* Partition-set parameters   */
DCL PTN_ARRAY(4) FIXED BIN(31); /* Partition parameters       */
DCL (TYPE,ATVAL,COUNT,LINE_COUNT) FIXED BIN(31);
DCL SOURCE FILE RECORD INPUT;   /* Program source file        */
DCL LISTING FILE RECORD INPUT;  /* Program listing file       */
DCL END01 BIT(1);               /* End-of-file flag           */
DCL BLOCK80 CHAR(80);           /* Input rec. from source file*/
DCL BLOCK121 CHAR(121);         /* Input rec. from listing file*/

CALL FSINIT;
/* Define partition-set grid                                  */
PTS_ARRAY(1)=9;                 /* 9 rows in partition set    */
PTS_ARRAY(2)=1;                 /* 1 column in partition set  */
PTS_ARRAY(3)=0;                 /* Use real partitions        */
/*      P-SET ID    NO. OF PARMS    PARAMETER ARRAY           */
CALL PTSCRT(1,           3,            PTS_ARRAY);                A

/* Create partition in top four-ninths of screen              */
PTN_ARRAY(1)=1;     /* Starts in row 1 (of the 9-row PTN-SET) */
PTN_ARRAY(2)=1;     /* Starts in col 1 (of the 1-col PTN-SET) */
PTN_ARRAY(3)=4;                 /* Depth is 4 rows            */
PTN_ARRAY(4)=1;                 /* Width is 1 column          */
/*       PTN ID    NO. OF PARMS    PARAMETER ARRAY            */
CALL PTNCRT(1,          4,          PTN_ARRAY);                  B
CALL FSPCRT(1,83,82,0); /* Create scrollable page 83 rows deep*/  C
CALL FSPWIN(1,1,25,82); /* .. of which 25 rows show at a time */  D
CALL ASDFLD(1000,1,34,1,14,2);
CALL ASCPUT(1000,14,'PROGRAM SOURCE');
```

Figure 128 (Part 1 of 3). Program using scrollable partitions and two cell sizes

```
OPEN FILE(SOURCE);        /* Open file holding program source */   E
ON ENDFILE(SOURCE) END01='1'B;  /* Set flag at end-of-file    */
END01='0'B;                     /* Initialize the flag        */
LINE_COUNT=0;                   /* Initialize the line count  */

READ FILE(SOURCE) INTO (BLOCK80);/* Read first source record  */
DO WHILE (END01='0'B);          /* Read up to 80 source recs. */
  LINE_COUNT=LINE_COUNT+1;      /* Bump line count            */
  IF LINE_COUNT>80 THEN DO;     /* Set limit of 80 lines      */
    CALL ASDFLD(1001,2,18,1,44,2);
    CALL ASCPUT(1001,44,
                 'SOURCE FILE TOO BIG. 1ST 80 LINES DISPLAYED');
    GOTO PART2;                 /* Go to process listing file */
  END;                          /* End of '>80' DO-group      */

  READ FILE(SOURCE) INTO (BLOCK80);  /* Next source record    */
  CALL ASDFLD(LINE_COUNT,LINE_COUNT+2,2,1,80,2);
  CALL ASCPUT(LINE_COUNT,80,BLOCK80);
END;                          /* End of source records DO-loop */   F
PART2:   /* Create second partition for listing file          */
PTN_ARRAY(1)=6; /* PTN starts in row 6 (of the 9-row PTN-set) */
/*       PTN ID    NO. OF PARMS    PARAMETER ARRAY             */
CALL PTNCRT(2,          4,           PTN_ARRAY);                  G
CALL FSPCRT(1,38,123,0);   /* Create page 123 cols by 35 rows */   H
CALL FSPWIN(1,1,25,123);   /* .. of which 25 show at a time   */   I
CALL ASDFLD(1000,1,54,1,15,2);  /* Alpha field for title      */
CALL ASCPUT(1000,15,'PROGRAM LISTING');

OPEN FILE(LISTING);      /* Open file holding program listing */   J
ON ENDFILE(LISTING) END01='1'B; /* Set flag at end-of-file    */
END01='0'B;                     /* Initialize the flag        */
LINE_COUNT=0;                   /* Initialize the line count  */
BLOCK121=' ';                   /* Clear record               */
READ FILE(LISTING) INTO (BLOCK121); /* Read 1st listing record*/
DO WHILE (END01='0'B);           /*Read up to 35 listing recs.*/
  LINE_COUNT=LINE_COUNT+1;       /* Bump line count            */
  IF LINE_COUNT>35 THEN DO;      /* Set limit of 35 lines      */
    CALL ASDFLD(1001,2,38,1,45,2);
    CALL ASCPUT(1001,45,
                'LISTING FILE TOO BIG. 1ST 35 LINES DISPLAYED');
    GOTO ENDIT;                  /* Send output to display    */
  END;                           /* End of '>35' DO-group     */
  CALL ASDFLD(LINE_COUNT,LINE_COUNT+2,2,1,121,2);
  CALL ASCPUT(LINE_COUNT,121,BLOCK121);
  BLOCK121=' ';                  /* Clear record              */
  READ FILE(LISTING) INTO (BLOCK121); /* Next listing record  */
END;                             /* End of listing DO-loop    */   K
```

Figure 128 (Part 2 of 3). Program using scrollable partitions and two cell sizes

```
ENDIT:

CALL ASREAD(TYPE,ATVAL,COUNT);/* Send 2 partitions to display */   L

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
%INCLUDE ADMUPINP;

END PARTEX2;
```

Figure 128 (Part 3 of 3). Program using scrollable partitions and two cell sizes

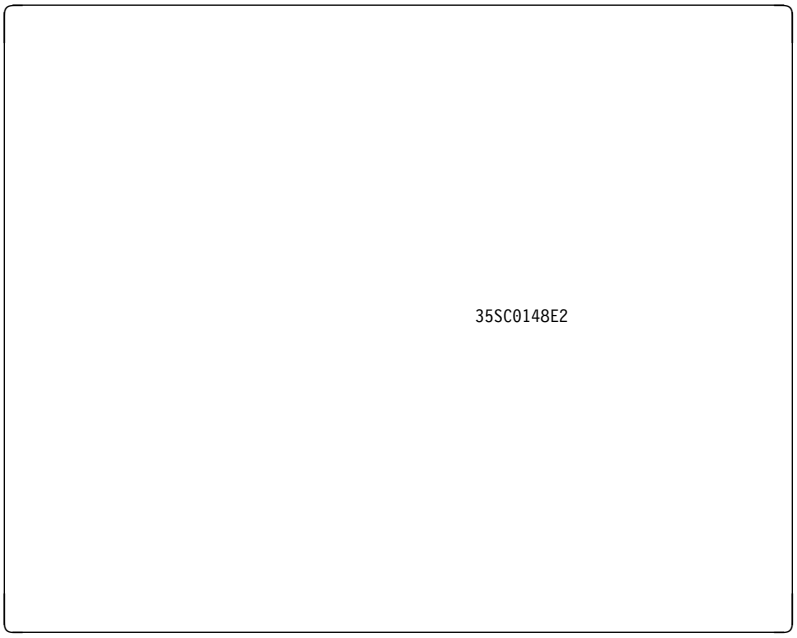


Figure 129. Screen with two cell sizes

Using operator windows to write task-manager programs

A task manager is a program that controls the initiation, termination, and execution priority of several applications that are running concurrently on the same display device, and it usually has an end-user dialog for this purpose.

Under TSO and CMS, and on all GDDM-supported display devices except the 5080 and 6090, a task manager can use a GDDM instance to enable the end user to run several GDDM applications concurrently. (In this context, a GDDM application means any program that uses GDDM to handle its terminal input/output.)

Typically, the end user accesses each application, and the end-user dialog with the task manager, through rectangular subdivisions of the display called operator

windows. The end user always interacts with the highest priority window, which is called the **active operator window**. GDDM highlights the active operator window with a special border, and always ensures that it is visually the topmost.

In a task manager, when the user satisfies the read outstanding in the active operator window, the application associated with the window runs until it either executes another input/output call or terminates. Meanwhile, the applications in all the other windows wait, because they have unsatisfied reads outstanding. In this way, the GDDM windowing functions support concurrent execution of several different GDDM applications. They do not support windowing of programs that use nonGDDM function for terminal input/output. Such nonGDDM programs would take over the whole screen until they terminated.

The GDDM instance controls the concurrent sharing of the device by several applications, using a **coordination exit routine**. Task management is described more fully at page 494.

In a subset of the function, available under CICS as well as TSO and CMS, a single GDDM application can use windowing support in its dialog with the end user, to present the separate functions of the single application, each in an operator window. In this case, you do not need a coordination exit. This is described in "Example: Program using one operator window" on page 481.

Whether a GDDM instance is being used by a task manager, or by a single application, the basics of a windowing program are the same:

- The first DSOPEN in a GDDM program opens the real display device with the (WINDOW,YES) processing option. This automatically creates a default operator window, and associates the real display device with it.
- You then divide the screen of the real display device into one or more operator windows.
- Subsequent DSOPEN calls open one or more virtual display devices and associate each with an operator window. (Under a task manager, the subsequent DSOPENs would be in each application.) In the *GDDM Base Application Programming Reference* book, you can find descriptions of processing options for a virtual device that are overridden by the processing options for the real device.
- Each application (under a task manager) or each function (under a single application) then communicates with the terminal user through an operator window conceptually situated in front of a virtual screen, and can behave as if it had complete control of a real screen.

A virtual device can itself be opened with the (WINDOW,YES) processing option. Operator windows created for this virtual device are further subdivisions of the **real** screen. So, although you can conceptually define hierarchies of operator windows, they **do not** appear inside each other. Rather, they are displayed as peers, according to their priorities.

A real or virtual device that is opened for windowing is called a **coordinating device** to denote that it coordinates the sharing of the device.

The association of the real device, operator windows and virtual devices in a single application is shown in Figure 130 on page 481.

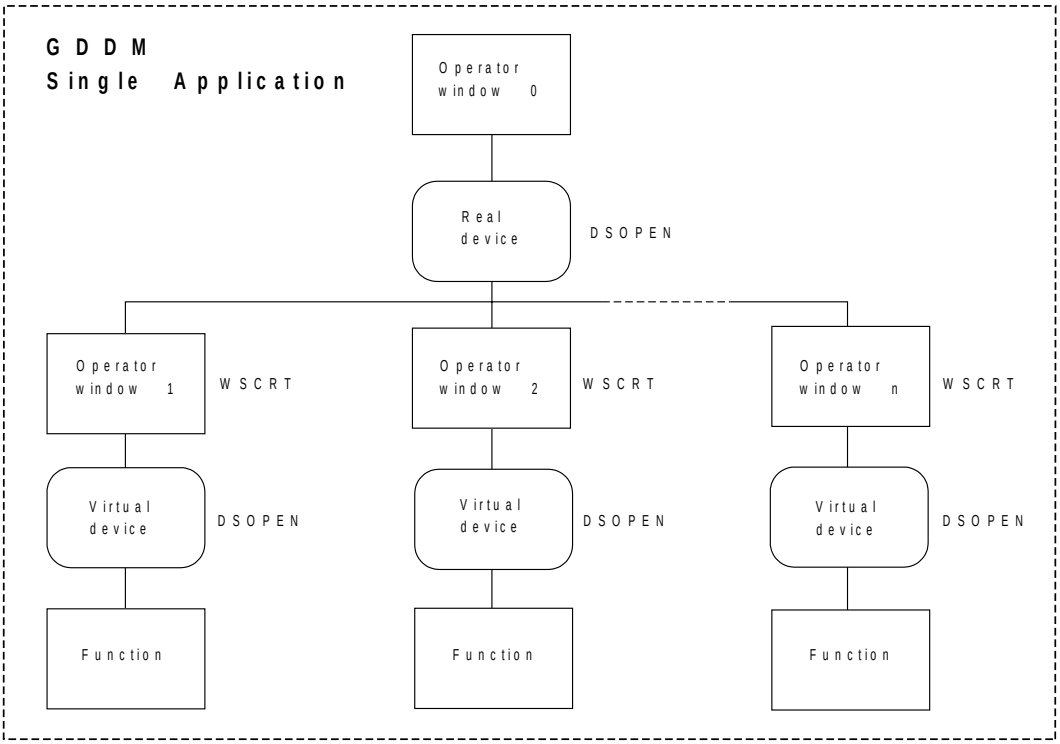


Figure 130. Hierarchy of devices and windows in a single application

You can use GDDM calls in your program to set or change the size, position, and viewing priority of operator windows, which can overlap.

In addition, whenever the application or function in the active operator window is waiting for input, the terminal user can select a different operator window to have top priority in the viewing order, and therefore become the active operator window, or change the size and position of operator windows, using the GDDM user control functions. All of the above manipulations of the operator window by the user can be done **without** interaction with the application program.

A virtual screen can be larger than an operator window, and larger than the real screen; user control provides horizontal and vertical scrolling. GDDM user control functions for the terminal user are covered in the *GDDM User's Guide*.

You should not confuse operator windows with partitions. Partitions (sometimes called application windows) can only be controlled by the application to which they belong. Partitions cannot be controlled by the terminal user, or used to run several application programs. When you use both types of presentation structure at the same time, partitions appear as subdivisions of the real or virtual screen, viewed through an operator window.

Example: Program using one operator window

The programming example below shows how a single GDDM application can use an operator window and an associated virtual device in its dialog with the terminal user:

```
OPWIN1: PROC OPTIONS(MAIN);
 DCL (TYPE,MOD,COUNT) FIXED BIN(31);
 DCL (VALID,FINISH) BIT(1) INIT('0'B);
 DCL PROCOPTS(7) FIXED BIN(31) INIT(24,1,28,1,29,1,1);
 DCL NAMES(1) CHAR(8),                /* DSOPEN dummy namelist */
 ( START INIT(0),                     /* Parameter to SHOWGDF  */
   READ INIT(1),                      /* Parameter to SHOWGDF  */
   WINDOW,                            /* Window to run next    */
   WSARR(10) INIT((10)0)              /* WSCRT parameter array */
 ) FIXED BINARY(31);

/*****************************************************************/
/* Open real device for windowing                                */
/*****************************************************************/
 CALL FSINIT;
 CALL DSOPEN(9,1,'*',7,PROCOPTS,0,NAMES); /* Open real device   */   A
 CALL DSUSE(1,9);                         /* Use real device    */

/*****************************************************************/
/* Create window for virtual device                              */
/*****************************************************************/
 WSARR(3) = 32;                     /* 32 row virtual screen    */
 WSARR(4) = 80;                     /* 80 column virtual screen */
 CALL WSCRT(1,4,WSARR,8,'WINDOW 1');/* Create window 1          */   B
 CALL SHOWGDF(START);               /* Initialize window 1      */   C

/*****************************************************************/
/* Perform I/O on virtual device                                 */
/*****************************************************************/
  DO UNTIL (FINISH = '1'B);
    DO UNTIL (VALID = '1'B);
     CALL SHOWGDF(READ);              /* Process transaction    */   D
    END;
    VALID = '0'B;
  END;
 CALL WSDEL(1);     /* Delete window 1 and close virtual device */   E
 CALL FSTERM;
 RETURN;

/*****************************************************************/
/* Transaction processing routine                                */
/*****************************************************************/
SHOWGDF: PROC(ACTION);
 DCL  (ACTION,SEG_COUNT,OPT_ARRAY(2) INIT(0,2)) FIXED BIN (31),
       NAME CHAR(8), DESCRIPTION CHAR(1);
 SELECT(ACTION);
```

Figure 131 (Part 1 of 2). The "OPWIN1" program

```
/*****************************************************************/
/* Initialization of screen                                      */
/*****************************************************************/
 WHEN(START)
  DO;
   CALL DSOPEN(1,1,'*',0,PROCOPTS,0,NAMES);/*Open virtual device*/   F
   CALL DSUSE(1,1);                        /*Use virtual device */
   CALL GSFLD(1,1,30,80);
   CALL ASDFLD(1,31,2,1,44,2);
   CALL ASFCOL(1,1);
   CALL ASCPUT(1,44,'Enter the name of a picture to be displayed:');
   CALL ASDFLD(2,31,47,1,8,0);
   CALL ASFCOL(2,4);
   CALL ASDFLD(3,32,2,1,35,2);
   CALL ASFCOL(3,1);
   CALL ASCPUT(3,35,'PF1=User Control 2=Show       3=End');
   CALL ASFCUR(2,1,1);
  END;
/*****************************************************************/
/* Input transaction - validate                                  */
/*****************************************************************/
 WHEN(READ)
  DO;
    CALL ASREAD(TYPE,MODE,COUNT);
    CALL ASFCUR(2,1,1);
    IF TYPE = 1 & ((MOD = 2)      /* If (PF key 2 pressed       */
      & (COUNT > 0))              /* and a picture name entered)*/
     | MOD = 3 THEN               /* or (PF key 3 pressed)      */
     DO;                          /* then perform action        */
      VALID = '1'B;
      IF MOD = 2 THEN
       DO;
        CALL GSCLR;
        CALL ASCGET(2,8,NAME);
        CALL GSLOAD(NAME,2,OPT_ARRAY,SEG_COUNT,0,DESCRIPTION);
       END;
      IF MOD = 3 THEN
       FINISH = '1'B;
     END;
    ELSE
     CALL FSALRM;
  END;
 OTHERWISE;
 END;
 RETURN;
 END SHOWGDF;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
%INCLUDE ADMUPINW;
END OPWIN1;
```

Figure 131 (Part 2 of 2). The "OPWIN1" program

The programming example, OPWIN1, creates an operator window, associates a virtual device with it, and displays it. The window contains some procedural

alphanumerics prompting the terminal user to enter in an input field the name of a picture to be displayed. The picture must be of the ADMGDF format, having previously been saved by a GSSAVE call. (See Chapter 10, “Storing and retrieving graphics pictures” on page 173 and “Modifying graphics pictures that have been loaded into your program” on page 193 for details.) After the terminal user has entered the name of a picture, PF keys 1 through 3 have the following effect:

PF key 1 takes the user into user control.

PF key 2 causes the program to load and display the requested picture.

PF key 3 ends the application.

The program illustrates how to convert a real device into a virtual device. A useful function of the program is that it opens a virtual device with screen dimensions of 32 rows by 80 columns regardless of the screen size of the real device. The program forms the basis of OPWIN2 which shows how to write a transaction processor for two virtual devices.

The DSOPEN at **A** opens the real device, and specifies that the device is to be windowed, that user control is available, and that the PF1 key invokes user control. It does this using the processing option groups 24, 28, and 29, respectively, in PROCOPTS. Another way is to use nicknames. See “Coding a partial device definition for end users to change with nicknames” on page 374.

Specifying that a device is to be windowed creates a default operator window, with an identifier of 0, and associates the device with the window. You do not have to use this window. Instead, you may prefer to use only windows that you explicitly create, as in the example.

The WSCRT call at **B** creates an operator window. Creating an operator window, either explicitly or by default, makes it the **current operator window**. For any GDDM device, you can create several operator windows, but only one of them can be the current operator window. The current operator window is the one whose attributes can be modified by a WSMOD call, described in “Modifying the attributes of an operator window, using call WSMOD” on page 490. An operator window can be made current by creating it using call WSCRT, or selecting it using call WSSEL, or by an input/output call WSIO. WSSEL and WSIO are described in the next section. The operator window made current by the most recently executed WSCRT, WSSEL, or WSIO call, is also the **candidate operator window**. The candidate operator window is the window with which the next virtual device to be opened is associated. There is only one candidate operator window, no matter how many devices or applications there may be. This is further explained in the next section.

As only one operator window is explicitly created in the example, it is also the active operator window.

The WSCRT call has the following parameters:

- The identifier of the new operator window.
- The number of elements of the array in the third parameter.
- An array containing the attributes for the new operator window. If any attribute is not specified, or is specified as 0, the default attribute value is used. The ten attributes corresponding to the elements are as follows:

1. The coordination address exit address. The default value is zero.
2. An exit token to be passed to the coordination exit. The default value is zero.

The previous two elements are used when the windowing application is being used by a task manager that is running several applications. Their use is covered in “Task management” on page 494.

3. The number of rows in the virtual screen of the virtual device opened in any subsequent DSOPEN. The default is the real screen depth.
4. The number of columns in the virtual screen of the virtual device opened in any subsequent DSOPEN. The default is the real screen width.
5. The row position of the top-left-hand corner of the operator window on the real screen. The default position is row 1.
6. The column position of the top-left-hand corner of the operator window on the real screen. The default position is column 1.

The above row and column attributes relate to the position of the top-left-hand corner of the window contents, not the position of the top-left-hand corner of the window frame.

7. The number of rows in the operator window. This does not include any rows occupied by the window frame.
8. The number of columns in the operator window. This does not include any columns occupied by the window frame.
9. The row position of the top-left-hand corner of the operator window on the virtual screen.
10. The column position of the top-left-hand corner of the operator window on the virtual screen.

- The length in bytes of the string in the final parameter.
- A string containing the title to be incorporated into the frame of the operator window.

The call at **C** calls the routine SHOWGDF to initialize the window with the procedural alphanumerics. The DSOPEN call at **F** opens a virtual device. When the DSOPEN is executed, GDDM automatically associates the virtual device with the candidate operator window that was created at **B** .

The call to SHOWGDF at **D** shows the requested picture.

When the program ends, the WSDEL call at **E** deletes the operator window, and also closes the virtual device associated with it.

Example: Program using two operator windows

The following program extends OPWIN1 to create and display two operator windows, each with its own virtual device. A picture can be displayed in each window, so that they can be visually compared.

Whenever the function in the active operator window is waiting for input, the terminal user can select another operator window to have top priority in the viewing order, and therefore to be active. This can be done with the implicit user control

function by either moving the graphics cursor (if displayed) into the required window, and selecting it using any of the following:

- ENTER key
- Button 1 on a mouse or puck
- Stylus tip.

or, if the graphics cursor is not displayed, the window can be selected by moving the alphanumeric cursor into the required window and pressing the ENTER key. The terminal user can, instead, press the PF1 key to explicitly enter user control mode. Using this function, the size, position, and viewing priority of operator windows can subsequently be changed. All of the above manipulations of the operator window by the user can be carried out **without** interaction with the application program.

```
OPWIN2: PROC OPTIONS(MAIN);
  DCL (TYPE,MOD,COUNT) FIXED BIN(31);
  DCL (VALID,FINISH) BIT(1) INIT('0'B);
  DCL PROCOPTS(7) FIXED BIN(31) INIT(24,1,28,1,29,1,1);
  DCL NAMES(1) CHAR(8),                /* DSOPEN dummy namelist */
  ( START INIT(0),                     /* Parameter to SHOWGDF  */
    READ INIT(1),                      /* Parameter to SHOWGDF  */
    WINDOW,                            /* Window to run next    */
    WSARR(10) INIT((10)0)              /* WSCRT parameter array */
  ) FIXED BINARY(31);

/*****************************************************************/
/* Open real device for windowing                                */
/*****************************************************************/
 CALL FSINIT;
 CALL DSOPEN(9,1,'*',7,PROCOPTS,0,NAMES);/* Open real device    */  A
 CALL DSUSE(1,9);                        /* Use real device     */

/*****************************************************************/
/* Create two operator windows                                   */
/*****************************************************************/
 WSARR(3) = 32;                   /* 32 row virtual screen      */
 WSARR(4) = 80;                   /* 80 column virtual screen   */
 WSARR(5) = 2;        /* Top-left corner row window on glass    */
 WSARR(6) = 3;     /* Top-left corner column window on glass    */
 WSARR(7) = 20;                   /* # Rows of window on glass  */
 WSARR(8) = 60;                   /* # cols of window on glass  */
 WSARR(9) = 13;/* Top-left corner row window on virtual screen  */
 WSARR(10) = 1;/* Top-left corner col window on virtual screen  */
 CALL WSCRT(2,10,WSARR,35,'GDDM Programming Example - Window 2');   B
                                  /* Create window 2            */
```

Figure 132 (Part 1 of 3). The "OPWIN2" program

```
 WSARR(3) = 32;                  /* 32 row virtual screen      */
 WSARR(4) = 80;                  /* 80 column virtual screen   */
 WSARR(5) = 10;       /* Top-left corner row window on glass    */
 WSARR(6) = 20;       /* Top-left corner col window on glass    */
 WSARR(7) = 20;                  /*   Rows of window on glass  */
 WSARR(8) = 59;                  /*   Cols of window on glass  */
 WSARR(9) = 13; /* Top-left corner row window on virtual screen */
 WSARR(10) = 1; /* Top-left corner col window on virtual screen */
 CALL WSCRT(1,10,WSARR,35,'GDDM Programming Example - Window 1');   C
                                 /* Create window 1            */

 WINDOW=1;
 CALL SHOWGDF(START);            /* Initialize window 1        */   D
 CALL DSUSE(1,9);                /* Reuse real device          */
 CALL WSSEL(2);                  /* Select operator window 2   */   E
 WINDOW=2;
 CALL SHOWGDF(START);            /* Initialize window 2        */   F

/*****************************************************************/
/* Perform real i/o and select transaction processor            */
/*****************************************************************/
  DO UNTIL (FINISH = '1'B);
    DO UNTIL (VALID = '1'B);
     CALL DSUSE(1,9);            /* Reuse real device          */   G
     CALL WSIO(WINDOW);          /* Real I/O                   */   H
     CALL SHOWGDF(READ);         /* Process transaction        */   I
    END;
   VALID = '0'B;
  END;
 CALL DSUSE(1,9);                /* Reuse real device          */   G
 CALL WSDEL(1);    /* Delete window 1, close its virtual device */
 CALL WSDEL(2);    /* Delete window 2, close its virtual device */
 CALL FSTERM;
 RETURN;
/*****************************************************************/
/* Transaction processing routine                               */
/*****************************************************************/
SHOWGDF: PROC(action);
 DCL (ACTION,SEG_COUNT,OPT_ARRAY(2) INIT(0,2)) FIXED BINARY(31),
        NAME CHAR(8), DESCRIPTION CHAR(1);
```

Figure 132 (Part 2 of 3). The "OPWIN2" program

```
SELECT(ACTION);
  WHEN(START)                     /* Initialization of screen   */
   DO;
    CALL DSOPEN(WINDOW,1,'*',0,PROCOPTS,0,NAMES);                  J
                                  /* Open a virtual device      */
                                  /* Device id = window id      */
                                  /* for simplicity             */
    CALL DSUSE(1,WINDOW);         /* Use virtual device         */  K
    CALL GSFLD(1,1,30,80);
    CALL ASDFLD(1,31,2,1,44,2);
    CALL ASFCOL(1,1);
    CALL
      ASCPUT(1,44,'Enter the name of a picture to be displayed:');
    CALL ASDFLD(2,31,47,1,8,1);
    CALL ASFCOL(2,4);
    CALL ASDFLD(3,32,2,1,35,2);
    CALL ASFCOL(3,1);
    CALL ASCPUT(3,35,'PF1=User Control 2=Show       3=End');
    CALL ASFCUR(2,1,1);
   END;
/*****************************************************************/
/* Input transaction - validate                                  */
/*****************************************************************/
  WHEN(READ)
   DO;
    CALL DSUSE(1,WINDOW);         /* Use virtual device         */  L
    CALL ASREAD(TYPE,MOD,COUNT);                                   M
    CALL ASFCUR(2,1,1);
    IF TYPE = 1 & ((MOD = 2)      /* If (PF key 2 pressed       */
      & (COUNT > 0))              /* and a picture name entered)*/
     | MOD = 3 THEN               /* OR (PF key 3 pressed       */
     DO;                          /* then perform action        */
      VALID = '1'B;
      IF MOD = 2 THEN
       DO;
        CALL GSCLR;
        CALL ASCGET(2,8,name);
        CALL GSLOAD(NAME,2,OPT_ARRAY,SEG_COUNT,0,DESCRIPTION);
       END;
      IF MOD = 3 THEN
        FINISH = '1'B;
     END;
    ELSE                          /* else ...                   */
     CALL FSALRM;                 /* sound alarm                */
   END;
  OTHERWISE;
END;
RETURN;
END SHOWGDF;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
%INCLUDE ADMUPINW;
END OPWIN2;
```

Figure 132 (Part 3 of 3). The “OPWIN2” program

The above program illustrates a “transaction processing” type of input design. This kind of design could form the basis of a window management program that did not use a coordination exit.

The program is essentially similar to OPWIN1. The DSOPEN call at **A** opens the real device for windowing. This time two operator windows are created, by the calls to WSCRT at **B** and **C**. When you first create a number of overlapping operator windows in an application, the viewing order depends on the order that you create the operator windows in. The operator window that you create first is at the bottom of the viewing order, and the operator window that you create last is at the top. On the display screen, each operator window appears in front of the operator windows that are below it in the viewing order. The topmost window (operator window 1 in the example) is the active operator window. Your program can change the viewing order, as described in “Prioritizing operator windows” on page 491.

As mentioned in the previous section, the current operator window is the one whose attributes can be modified by a WSMOD call. The candidate operator window is the window with which the next virtual device to be opened is associated. In a single application like the example, not running under a task manager, the current operator window is always the candidate operator window. So, when is the current operator window **not** the candidate operator window? Remember, when you have several applications running concurrently under a task manager, only one of those applications is actually **executing**, while the others are waiting because they have unsatisfied reads outstanding. Each of the applications can have a current operator window. But no matter how many devices or applications there may be, only the operator window made current by **the most recently executed** WSCRT, WSSEL, or WSIO call is the candidate operator window with which the next virtual device to be opened is associated.

After the WSCRT call at **C**, operator window 1 is the current and candidate operator window. At **D** the program calls SHOWGDF(START) to open and use a virtual device for operator window 1 and to initialize the window with procedural alphanumerics. Following the call to SHOWGDF(START), the program issues a call to DSUSE to reuse the real device, because SHOWGDF(START) contains a DSUSE to a virtual device.

The WSSEL call at **E** selects operator window 2 to be the candidate operator window. WSSEL also makes the operator window current. Making an operator window current has no effect on the viewing order. At **F** the program calls SHOWGDF(START) to open a virtual device for the operator window 2 and to initialize the window with procedural alphanumerics.

To keep the program as simple as possible, it calls the routine SHOWGDF for both operator windows. You could easily alter the program to call a different routine for each window. The DSOPEN call at **J** opens a virtual device and gives it the same identifier as the operator window with which it is associated. This has been done because SHOWGDF is called for two windows, and therefore two separate virtual devices are opened. It also makes it clear which virtual device is associated with which operator window. In practice you could give the virtual device any valid identifier, if it differs from any other device identifier within the same instance of GDDM. GDDM automatically associates each virtual device with its respective operator window.

The “do loop” that follows performs the I/O for the real device and, for the active operator window, calls SHOWGDF(READ) to restore and display a picture. Where there are several operator windows in an application, as in the example, the operator window that is the highest in the viewing order immediately before the input/output call (WSIO in the example) is the active operator window. The first time through the do loop operator window 1 is the topmost operator window when I/O takes place for the real device, at **H**. It is therefore initially the active operator window.

The user can change the viewing order by selecting a different window to be active. You can also change the viewing order in your program, as described in “Prioritizing operator windows” on page 491.

The call to DSUSE at **G** is necessary to reuse the real device as the primary device, because in SHOWGDF, which is called at **D**, **F**, and **I**, there is a DSUSE to a virtual device.

The WSIO call at **H** displays the two windows and their contents on the screen of the real device. In WSIO’s one parameter, GDDM returns the identifier of the active operator window. WSIO also makes the active operator window the current operator window. If the terminal user should alter the viewing priority of the two windows and make a different window active, using control mode for example, GDDM returns the identifier of the new active operator window in WSIO's parameter.

When WSIO is called for a device that has windows that do not specify coordination exits, as in the above example, GDDM behaves as follows: When the terminal user interacts with the active window, WSIO completes, and creates a pending attention interrupt for the virtual device associated with the window. (If an attention interrupt is already pending for the virtual device, it is replaced by the new one.) The interrupt is left pending until the next I/O function is called for the virtual device. If the next I/O function is an ASREAD, as in the example, then, as it normally waits for an attention interrupt, it is satisfied by the one that is pending. Therefore, no I/O is performed by the ASREAD, but the pending information is returned. For a description of how WSIO behaves **with** coordination exits, see “Task management” on page 494.

In SHOWGDF, called at **I**, the DSUSE call at **L** uses the operator window identifier returned by WSIO to ensure that the virtual device used is the one associated with that operator window. The ASREAD that follows, at **M** is therefore always issued against the virtual device associated with the active operator window.

The first two windowing programs have introduced the basic principles of operator windows, and some of the calls. The following sections describe some of the other windowing calls.

Modifying the attributes of an operator window, using call WSMOD

Normally you set the attributes of an operator window when you create it, using WSCRT. If you want to subsequently redefine the attributes of a window, you can use the WSMOD call. WSMOD modifies the attributes of the **current** operator window. Where there are several operator windows in a program, and the window whose attributes you want to modify is not at present the current one, you can

make it current using the WSSEL call, already described. Here is an example of WSMOD:

```
CALL WSMOD(1,6,MOD_ARRAY,14,'COMMAND WINDOW');
```

The parameters of WSMOD are as follows:

- The number of the first element of the array in the third parameter. It must have a value in the range 0 through 6.
- The number of elements of the array in the third parameter. It must have a value in the range 0 through 6.
- An array containing the attributes for the current operator window. If any attribute is not specified, or is specified as –1, the existing value is unchanged. The attributes that you can modify correspond exactly to the last six elements of the array in the WSCRT call. If any attribute is specified as 0, the default value is used. See the WSCRT call above for the default values. The six attributes corresponding to the elements are as follows:
  1. The row position of the top-left-hand corner of the operator window on the real screen.
  2. The column position of the top-left-hand corner of the operator window on the real screen.
  3. The number of rows in the operator window. This does not include any rows occupied by the window frame.
  4. The number of columns in the operator window. This does not include any columns occupied by the window frame.
  5. The row position of the top-left-hand corner of the operator window on the virtual screen.
  6. The column position of the top-left-hand corner of the operator window on the virtual screen.
- The length in bytes of the character string in the final parameter.
- The title to be incorporated into the frame of the operator window.

Prioritizing operator windows

You can change the priority of some or all of the operator windows in the viewing order, using the call WSSWP. This call lets you specify an array of identifiers of operator windows whose priorities are to be adjusted by placing them as neighbors to one of the other operator windows in the viewing order. The topmost window is always the active operator window.

For example, assume that the following seven operator windows are in descending order:

TOP 7, 6, 5, 4, 3, 2, 1 BOTTOM

If you wanted to take operator windows 7, 2, and 1, and change their order of viewing so that they are after window 5 and before window 4, like this:

TOP 6, 5, **1, 7, 2,** 4, 3 BOTTOM

you would issue the following call:

```
DCL PRI_ARRAY (3) FIXED BIN(31) INIT(2,7,1);

CALL WSSWP(1,4,3,PRI_ARRAY);     /* Change window viewing order */
```

The parameters are as follows:

- The first parameter specifies whether the operator windows in the array in the final parameter are to be placed in descending or ascending order from the **reference operator window**. The reference operator window is the reference point in the viewing order about which the reordering of the windows is to take place. It is specified in the second parameter. In the example, it is operator window 4.

  The first parameter can have these values:

  -1 Descending order. The operator windows in the array are placed behind the reference operator window.

  1 Ascending order (as in the example). The operator windows in the array are placed in front of the reference operator window.

- The second parameter contains the identifier of the reference operator window relative to which the reordering is to take place. It can have a value of –1, the effect of which depends on whether you set the first parameter to ascending or descending order:

  Descending The first operator window in the array becomes the top operator window in the viewing order, and the rest of the operator windows in the array are placed behind it.

  Ascending The first operator window in the array becomes the bottom operator window in the viewing order, and the rest of the operator windows in the array are placed in front of it.

- The third parameter contains the number of elements in the array in the final parameter

- The final parameter is an array of identifiers of operator windows whose priorities are to be adjusted relative to the reference operator window. Any element of the array can contain a value of –1, which causes all further elements to be ignored.

The reordering process takes the first operator window in the array, and places it above or below the reference operator window in the viewing order, depending on the order specified in the first parameter. It then takes the second operator window in the array, and places it above or below the first operator window, and so on, until all the elements of the array parameter have been processed, or until a value of –1 is found in the array.

Querying the priority of overlapping operator windows

There are two calls that you can use to query the priority of operator windows.

In the last section, the WSSWP call was used to change the viewing priority of a specified array of operator window identifiers relative to a specified reference operator window identifier. The corresponding query call WSQWP returns an array of operator window identifiers relative to a specified reference operator window identifier. For example, here is a typical call that returns the identifiers of the three operator windows that are above operator window 5 in the viewing order:

```
DCL PRI_ARRAY (3) FIXED BIN(31);

CALL WSQWP(1,5,3,PRI_ARRAY);      /* Query window viewing order */
```

The parameters are as follows:

- The first parameter specifies whether the array in the final parameter is to return the identifiers of operator windows that are in descending or ascending order from the reference operator window. The possible values are:

  -1 Descending order

  1 Ascending order

- The second parameter specifies the identifier of the reference operator window that the query relates to. It can have a value of –1, the effect of which depends on whether you set the first parameter to ascending or descending order:

  Descending The first operator window in the array is the top operator window in the viewing order, and the rest of the operator windows in the array are those that are behind it.

  Ascending The first operator window in the array is the bottom operator window in the viewing order, and the rest of the operator windows in the array are those that are in front of it.

- The third parameter contains the number of elements to be returned in the array in the final parameter
- The final parameter is an array that holds the returned identifiers of operator windows that descend or ascend from the reference operator window.

There is another query call, WSQWI, that returns the identifiers of all operator windows. Here is a typical call:

```
DCL PRI_ARRAY (7) FIXED BIN(31);

CALL WSQWI(1,7,PRI_ARRAY);      /* Query all window identifiers */
```

The parameters are as follows:

- The first parameter specifies the type of operator window that you want information returned about:

  1 All operator windows.

- The second parameter specifies the number of elements in the array in the final parameter.
- The final parameter is the name of an array in which GDDM returns the requested information.

There is also a call WSQWN that you can use to query the total number of operator windows. See the *GDDM Base Application Programming Reference* book for more details.

Querying operator window attributes, using WSQRY

You can query the attributes of the current operator window. Here is a typical call:

```
DCL WSARR(11) FIXED BIN(31);
DCL ACTUAL_LENGTH FIXED BIN(31);
DCL STRING CHAR(20);

CALL WSQRY(1,11,WSARR,ACTUAL_LENGTH,20,STRING);
```

The parameters are as follows:

- The number of the first element in the array.
- The number of elements in the array.
- An array in which the attributes of the current operator window. are returned. In the first element, GDDM returns the window identifier. The remaining ten elements correspond to the ten elements that you can set using WSCRT.
- The length of the window title is returned by GDDM.
- In this parameter you specify how much of the title you want returned.
- The window title is returned by GDDM.

Task management

The “Example: Program using two operator windows” on page 485 showed how a single GDDM application could use windowing in its dialog with the terminal user, to present separate functions of the application, each in an operator window. You may recall that the application used the DSOPEN call to open the real device, and two WSCRT calls to open two operator windows. A subroutine was then called for each window. The subroutine contained a DSOPEN, that opened a virtual device for each operator window.

You can use the same windowing principles to write your own task manager program. The GDDM-supplied example task manager (ADMUTMT for TSO, ADMUTMV for VM/CMS) is an example of such a program. The task manager uses DSOPEN to open the real device, and WSCRT and the other windowing calls to create and control an operator window for each application program. Subsequent DSOPEN calls **in each application program** open one or more virtual devices, which are associated with the operator windows created by the task manager. This is illustrated in Figure 133 on page 495.

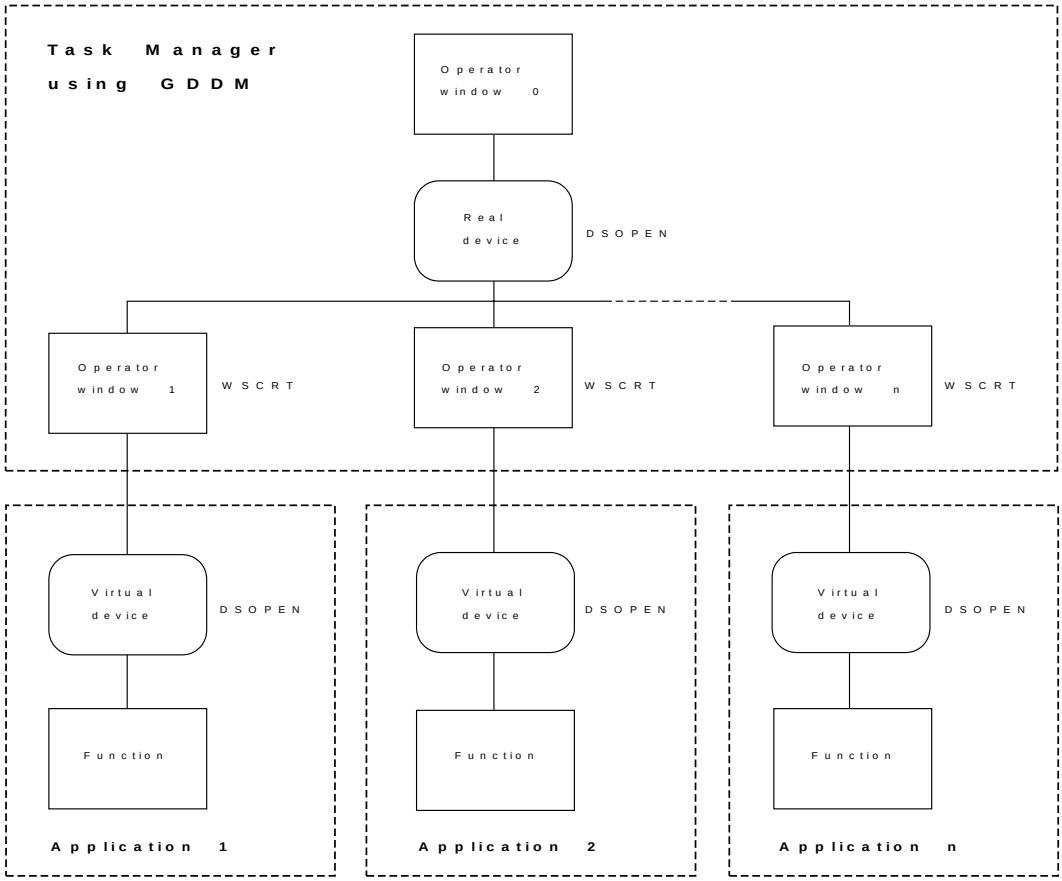


Figure 133. Task manager with several applications

The task manager manages the display device screen and other resources. In addition, the task manager must either use the task-management facilities of the operating system, or use its own pseudotasking facilities (TSO has full task-management facilities but CMS does not). The system tasking or pseudotasking executes each program in a separate sub-task.

GDDM enables several application programs to share the screen by allowing the task manager to intervene in the execution of the program's input/output calls. When each operator window is created, the task manager specifies (in the first array element of the last parameter of the WSCRT call) the address of a coordination exit routine. This runs in the application program subtask, and is invoked by GDDM whenever the application calls a function that requires input/output for the terminal–an ASREAD call, typically, as shown in Figure 134 on page 496.

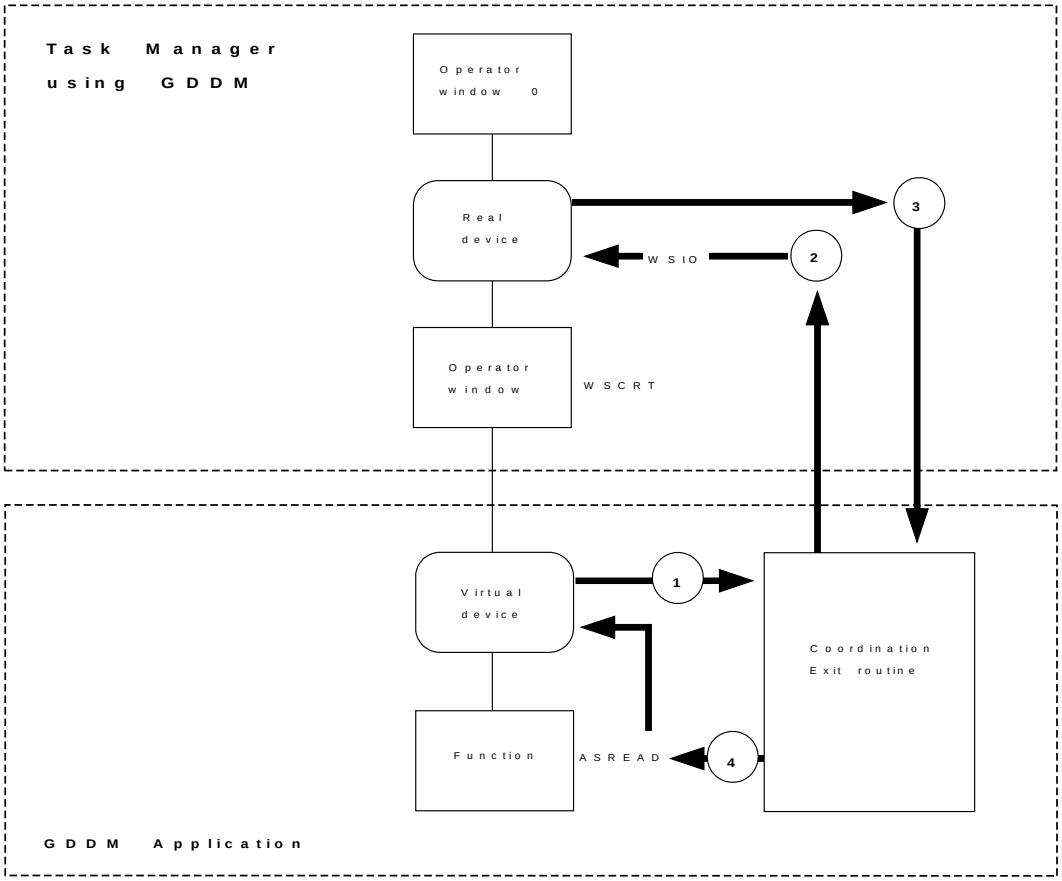


Figure 134. The coordination exit routine

The numbers in the figure represent the following events:

1. An input/output call is issued by the application, causing GDDM to invoke the coordination exit routine.
2. The exit routine, when invoked, must post the task-manager task and wait. The task manager must then call WSIO, the coordinated output/input call. The WSIO call updates all the windows on the screen. WSIO also returns the identifier of the topmost window on the screen. The task manager uses this to find out which subtask to post. It then posts that subtask and waits.
3. When the task manager posts the subtask, control passes back to the coordination exit routine, which in turn returns control to GDDM.
4. Control then returns to the ASREAD (or other application input/output call). GDDM completes the processing of this call, and pass control back to the application program. Any input data entered by the terminal user is then available to the application.

When the application terminates, normally or abnormally, control is passed to the task manager, which typically calls WSIO to find out from the terminal user what to do next.

The purpose of the coordination exit routine is to switch control from the subtask to the main task, or the other way round. There is a direction parameter to tell it which way to switch.

Running existing GDDM applications under a task manager

You can usually run existing GDDM applications under a task manager without having to change, recompile, or re-link-edit the application.

However, under CMS, if an application is in the form of a text file, the application must have been written using the reentrant interface. This is because text files are automatically link-edited at run time, and applications written in the nonreentrant interface attempt, when link-edited, to pick up the same entry points as the task manager. If you want to run a nonreentrant application under a task manager, explicitly link-editing the load module ensures that the application picks up its own entry points.

Under TSO, the only restriction is that you cannot, under a task manager, run more than one application where each application uses the **same** ddname, but accesses **different** data sets.

GDDM-IMD can be run under a task manager, but it may not be run in an operator window that is smaller than the screen.

How FSSAVE and FSSHOW perform with operator windows

An FSSAVE call in an application running in an operator window saves the contents of the virtual screen (without borders), subject to the outer limits of the real screen. For example:

- If the virtual screen is smaller than the real screen, the virtual screen is saved.
- If the virtual screen is larger than the real screen, a real screen-sized virtual screen is saved.

A picture restored by an FSSHOW or FSSHOR call in an application takes over the whole real screen when the call is issued. No other operator window or window borders are seen.

The next input by the terminal operator is passed to the device that issued the FSSHOW or FSSHOR, and all previously displayed windows are redisplayed.

Allocation of resources to operator windows

When operator windows are used to run several independent programs at the same time on the same device, more than one program may try to use the same PS store. In this case, of the operator windows requiring the PS store, the one that has the highest viewing priority uses it, and the others use the default PS store. This sharing of PS stores is transparent to the program.

On devices like the 3279, GDDM uses programmed symbols for graphics, and to draw the borders around operator windows. The PS stores are allocated in the following order:

- Symbol sets reserved by the application for the active operator window
- Graphics in the active operator window
- The borders of all operator windows
- Symbol sets and graphics for non-active operator windows.

If you have a number of operator windows containing graphics, that are to be displayed on the screen at the same time, PS overflow can occur. In this case, GDDM guarantees picture fidelity for the active operator window only, and may have to degrade the appearance of borders and picture fidelity for non-active operator windows.

How to free resources when a task terminates

MVS provides full task-management facilities, one feature being that when a task terminates, all the resources obtained by that task, and by any subtasks that it might have, are freed. This applies to virtual storage, files, and enqueue requests.

If you are using MVS, and not taking advantage of its tasking facilities, or if you are using VM, which does not have this feature, you can use a GDDM call to group one or more applications into an **application group**. Using the ESACRT call in your task manager program creates an application group and makes it current. All instances of GDDM that are initialized are associated with the current application group. Using the ESADEL call causes GDDM to issue an internal FSTERM for each instance of GDDM associated with the specified application group. Storage, files, and enqueue requests owned by all the instances are therefore freed.

When control is passed from an instance of GDDM in one application group to an instance of GDDM in a different application group, neither instance having terminated, you can use the ESAQRY call to save the current application group, and use the ESASEL call to make the target application group current.

If you **are** using MVS real-tasking facilities, you should not use the ESAxxx group of calls. If you do, GDDM may try to release resources that have already been released by MVS.

See the *GDDM Base Application Programming Reference* book for a description of each call.

Part 3 Examples of GDDM programs

© Copyright IBM Corp. 1982, 1996

Chapter 23. Programming examples

This section provide you with examples of GDDM programs in the following languages.

- System/370 Assembler
- APL2
- BASIC
- C/370
- REXX
- PL/I CICS pseudoconversation

Note: Unlike **sample** programs, these programs are not provided with the GDDM Base licensed program.

A System/370 Assembler programming example

This program uses the nonreentrant interface to GDDM. It draws straight lines in response to cursor movement and user-generated attentions. PF3 or a GDDM error stops the program. You might want to check for the successful completion of all GDDM calls; most of these checks have been omitted here for clarity.

```
ASMNR    CSECT ,
         STM   R14,R12,12(R13)     SAVE REGISTERS
         BALR  R12,0               BASE REGISTER FOR CODE
         USING *,R12               ... AND TELL THE ASSEMBLER
         ST    R13,SAVEAREA+4      SAVE CALLER'S SAVE AREA ADDRESS
         LA    R11,SAVEAREA        GET SAVE AREA ADDRESS
         ST    R11,8(,R13)         ... AND STORE IT
         LR    R13,R11             SAVE AREA FOR CALLED ROUTINES
         CALL  FSINIT,(0),VL       INITIALIZE GDDM
         LTR   R9,R15              ... AND CHECK FOR NORMAL RETURN
         BNZ   RETURN
         CALL  GSSEG,(SEGNO),VL    OPEN SEGMENT NUMBER 1
         LTR   R9,R15              ... AND CHECK FOR NORMAL RETURN
         BNZ   RETURN
         CALL  GSMOVE,(X,Y),VL     INITIALIZE CURRENT POSITION
LOOP     DS    0H                  TOP OF LOOP
         CALL  ASREAD,(TYPE,VALUE,MODS),VL WAIT FOR OPERATOR ACTION
         CALL  GSQCUR,(PTYP,X,Y),VL FIND WHERE CURSOR IS
         CALL  GSLINE,(X,Y),VL     DRAW LINE THERE FROM PREVIOUS POINT
         CLC   TYPE(4),F1          CHECK FOR A PROGRAM FUNCTION KEY
         BNE   LOOP                ... CONTINUE IF NOT
         CLC   VALUE(4),F3         CHECK FOR PF3
         BNE   LOOP                ... CONTINUE IF NOT
         CALL  GSSCLS,(0),VL       CLOSE SEGMENT
         CALL  FSTERM,(0),VL       TERMINATE GDDM IF PF3 WAS USED
RETURN   L     R13,4(,R13)         RECOVER SAVE AREA ADDRESS
         LR    R15,R9              SET RETURN CODE
         L     R14,12(,R13)        RESTORE REGISTERS
         LM    R0,R12,20(R13)      ... FOR CALLER
         BR    R14                 RETURN TO CALLING ROUTINE
*                        PROGRAM CONSTANTS
F1       DC    F'1'                INDICATES A PF KEY WAS USED
F3       DC    F'3'                PF KEY NUMBER 3
```

© Copyright IBM Corp. 1982, 1996

```
SEGNO    DC    F'1'                SEGMENT NUMBER 1
SAVEAREA DS    18F                 REGISTER SAVE AREA
TYPE     DS    F                   TYPE OF ATTENTION (FULLWORD INTEGER)
VALUE    DS    F                   ATTENTION VALUE (FULLWORD INTEGER)
MODS     DS    F                   MODIFIED FIELDS (FULLWORD INTEGER)
PTYP     DS    F                   WINDOW INDICATION (FULLWORD INTEGER)
X        DC    E'50'               X-COORDINATE (SHORT FLOATING POINT)
*                                  ... INITIALIZED TO 50
Y        DC    E'50'               Y-COORDINATE (SHORT FLOATING POINT)
*                                  ... INITIALIZED TO 50
```

```
*                        EQUATES FOR REGISTERS
R0       EQU   0
R1       EQU   1
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   ASMNR
```

An APL2 programming example

With APL2 version 1 release 2 and later a function is supplied that enables you to use GDDM. The function is in a workspace called GDMX. When in APL2, you can type

```
)LOAD 2 GDMX
DESCRIBE
```

for more details.

GDMX lets you pass GDDM call names and parameter values to it. The calls can be coded in a similar way to that used for other programming languages, with the exception that you write the call name to the left of the function GDMX, and the arguments to the right of GDMX.

The following program draws straight lines in response to cursor movement and user-generated interrupts. PF3 stops the program.

```
      ∇ DEMO
[1]  ⍝ APL2 example using GDDM function supplied with APL2 Release 2
[2]   'GSSEG' GDMX  1                    ⍝ Open segment 1
[3]   'GSMOVE' GDMX  1                   ⍝ Initialise current position
[4]   LOOP:'ASREAD'  GDMX ''             ⍝ Wait for operator action
[5]    → ⍳1   3 ∨ ∊ = 2 ↑ 5 ↓ 1 ⊃RET_G   ⍝ Check for PF3
[6]   'GSQCUR' GDMX ' '                  ⍝ Find where cursor is
[7]   'GSLINE' GDMX ~2 ↑ 1 ⊃RET_G        ⍝ Line from previous point
[8]    →LOOP                             ⍝ Loop back
[9]   'END:'GSSDEL' GDMX 1               ⍝ Delete segment 1
      ∇
```

Figure 135. APL2 programming example

A BASIC programming example

```
0010 REM  How to use GDDM from an IBM BASIC program
0020 REM  === == === ==== ==== == === ===== =======
0030 REM
0040 REM  This sample program gives
0050 REM  some suggestions on how to use this function.
0060 REM
0070 REM  Two general things to remember - you need to be linked to
0080 REM  GDDM and have the appropriate GLOBAL command in effect to run.
0090 REM  An example is:
0100 REM     GLOBAL TXTLIB ADMRLIB ADMHLIB ADMPLIB ADMGLIB
0110 REM  If your default CMS LDRTBLS is less than 5,
0120 REM  you probably need to SET LDRTBLS 5 or more.
0130 REM
0140 REM  BASIC GDDM coding employs numbers for calls. To make it more
0150 REM  obvious which call is doing what, these numbers have been
0160 REM  equated with the familiar call names.
0170 REM
0180 OPTION BASE 1
0190 INTEGER
COLORS,HEAD_ATT,LABEL_ATT,KEY_ATT,ATMOD,PAT_ATT,AX_ATT,COPYP
0200 DIM
YARRAY(8),COLORS(4),HEAD_ATT(4),LABEL_ATT(4),KEY_ATT(4),PAT_ATT(3)
0210 DIM AX_ATT(3),COPYP(3)
0220 DATA 24,41,18,17,31,29,13,27
0230 MAT READ YARRAY
0240 DATA 1,2,4,6
```

```
0250 MAT READ COLORS
0260 DATA 7,3,0,175
0270 MAT READ KEY_ATT
0280 DATA 6,3,0,200
0290 MAT READ HEAD_ATT
0300 DATA 2,3,0,300
0310 MAT READ LABEL_ATT
0320 DATA 16,16,16
0330 MAT READ PAT_ATT
0340 DATA 7,0,0
0350 MAT READ AX_ATT
0360 DATA 0,1,1
0370 MAT READ COPYP
0380 CHRNIT = 268501248    :  CHXLAB = 268567811  :  CHHEAD = 268567042
0390 CHKEY  = 268567041    :  CHPIE  = 269289990  :  CHTERM = 268435712
0400 ASREAD = 202375168    :  CHSET  = 268566785  :  CHCOL  = 268567299
0410 GSCLR  = 202113795    :  CHHATT = 268568833  :  CHLATT = 268568835
0420 CHKATT = 268568837    :  CHKEYP = 268568577  :  CHPAT  = 268567300
0430 ASCPUT = 201852419    :  ASDFLD = 201852672  :  CHAATT = 268568321
0440 FSOPEN = 202899456    :  GSCOPY = 202899458  :  FSCLS  = 202899460
0450 CALL GDDM (CHRNIT)
0460 CALL GDDM (GSCLR)
0470 CALL GDDM (CHCOL,4,COLORS())
0480 CALL GDDM (CHPAT,3,PAT_ATT())
0490 CALL GDDM (CHXLAB,2,4,'19721984')
0500 CALL GDDM (CHSET,'CBOX')
0510 CALL GDDM (CHSET,'ABPI')
0520 CALL GDDM (CHSET,'KBOX')
0530 CALL GDDM (CHKEYP,"H","T","C")
0540 CALL GDDM (CHKATT,4,KEY_ATT())
0550 CALL GDDM (CHLATT,4,LABEL_ATT())
0560 CALL GDDM (CHAATT,3,AX_ATT())
0570 CALL GDDM (CHKEY,4,12,"PROGRAMMERS    PROFESSORS MAIL CARRIER DP OPERATOR")
0580 CALL GDDM (CHPIE,2,4,YARRAY())
0597 CALL GDDM (ASREAD,ATTYPE,ATMOD,COUNT)
0600 CALL GDDM (FSOPEN,'PIE     ',3,COPYP())
0613 CALL GDDM (GSCOPY,60,120)
0626 CALL GDDM (FSCLS,1)
0630 CALL GDDM (GSCLR)
0640 CALL GDDM (CHTERM)
0650 END

0660 REM
0670 REM  Here is a little guidance as to how data is passed in an array.
0680 REM  The basic calls are positional, and they expect a specific
0690 REM  number of parameters.  If you have an array
0700 REM  defined with data in it to pass to GDDM you can't just name
0710 REM  the array. In other words, if you have:
0720 REM    100 DIM NUMBERS(8)
0730 REM  to define an array with 10 elements
0740 REM  when you pass this array to GDDM within a call it goes inside
0750 REM  the parenthesis as:
0760 REM    NUMBERS()
0770 REM  not as:
0780 REM    NUMBERS or NUMBERS(8) or NUMBERS(X)
0790 REM
```

A C/370 programming example

```
/*********************************************************************/
/* THIS IS A C370 PROGRAM THAT MAKES USE OF THE GDDM API TO LOAD A    */
/* SAVED PICTURE (ADMGDF), ROTATE IT THROUGH AN ANGLE SPECIFIED BY    */
/* THE END USER AND SEND IT TO A GL PLOT FILE.                        */
/*********************************************************************/

#include <math.h>           /* Include C header files.                */
#include <stdio.h>
#include <string.h>

#include <admucina.h>       /* Include header files for entry points  */
#include <admucind.h>       /* into the GDDM nonreentrant API         */
#include <admucinf.h>
#include <admucing.h>
#include <admtstrc.h>

#define SEGBASE   99        /* Define constant.                       */

void rotate(float deg,float *rx,float *ry);  /* Declaration of rotate */
                                             /* function.             */

/* Main function: Displays a page for the user to enter the name of   */
/* an ADMGDF file and also a rotation. The ADMGDF file is then loaded */
/* and the contents of the file rotated by the angle specified. The   */
/* resulting picture is then sent to a directly attached plotter.     */
main ()
    {
     int attype,attval,count;
     int qry[2];
     int opt_arr[2]={99,2}; /* Initialize options for gsload.          */
     int segcnt;
     char desc[253];
     char name[8],degst[3];
     float angle;
     int s;
     float segx,segy;
     int plist[3]={46,1,0}; /* Initialize processing options for      */
                            /* dsopen. Specify output is to be produced */
                            /* as a GL file rather than go to a directly*/
                            /* attached plotter.                      */
     char nlist[2][8];      /* Declaration of name list parameter for   */
                            /* dsopen. Note the parameter is declared as*/
                            /* a 2D array as the call is defined as     */
                            /* being 'an array of 8 byte character      */
                            /* tokens'.                                 */
     int opt[1]={0};        /* Initialize option for dscopy.            */
                            /* Parameters to 'gssaga' function.         */
     float sx=1;            /* Scale factor.                            */
     float sy=1;
     float hx=0;            /* Shear.                                   */
     float hy=1;
     float rx=1;            /* Rotation.                                */
     float ry=0;
     float dx=0;            /* Translation.                             */
     float dy=0;
```

```
union {              /* Union used for fsqerr call. The fsqerr   */
       Admers error; /* call has a character string as the 2nd   */
       char str[160];/* parameter. However the returned          */
      }u;            /* information contains both character and  */
                     /* numeric values, the union operator is    */
                     /* used to map the structure onto the       */
                     /* character string. The Admers structure   */
                     /* is a predefined version of the structure */
                     /* returned by fsqerr. It is supplied in    */
                     /* the 'admtstrc' header file.              */

fsinit();            /* Initialize GDDM.                         */
fsqury(0,3,2,qry);   /* Query the number of rows and cols on the */
                     /* current device.                          */
fspcrt(1,qry[0],qry[1],0); /* Create a page using the number of  */
                     /* rows and cols queried from the device.   */
                     /* Note the first element in the array 'qry'*/
                     /* is qry[0] not qry[1]                    */
asdfld(1,10,20,1,20,2); /* Define some alphanumeric fields to    */
asdfld(2,10,41,1,8,0);  /* display and receive information.      */
asdfld(3,12,20,1,25,2);
asdfld(4,12,46,1,8,0);
ascput(1,20,"Name of ADMGDF file:");
strncpy(name,"DDJXBT2 ",8);
ascput(2,8,name);
ascput(3,25,"Rotation angle (degrees):");
strncpy(degst,"0  ",3);
ascput(4,3,degst);
asfcur(2,1,1);       /* Position cursor in the second field.     */
asread(&attype,&attval,&count); /* Display the page and wait for */
                     /* user interrupt. Note the parameters to   */
                     /* asread are returned by GDDM, so pass the */
                     /* address of the variables.                */
ascget(2,8,name);    /* Read the contents of the input fields.   */
ascget(4,3,degst);
sscanf(degst,"%f",&angle); /* Scan the float number from the     */
                     /* string 'degst' and store it in angle.    */
fspdel(1);           /* Delete the page.                         */

gsuwin(0,100,0,100);
gsload(name,2,opt_arr,&segcnt,253,desc); /* Load the requested   */
                     /* ADMGDF file. The values stored in        */
                     /* 'opt_arr' will load it to fill the screen*/
                     /* and begin storing the segments from the  */
                     /* file in numbers starting at 99.          */
fsqerr(160,u.str);   /* Pass the character string from the union */
                     /* to fsqerr.                               */
if (u.error.severity < 4) /* The severity element in the error   */
                     /* structure can be tested on return from   */
                     /* fsqerr. Continue if severity is less than*/
                     /* a warning.                               */
  {
   gsseg(1);         /* Begin a new segment.                     */
   for (s=SEGBASE;s<(SEGBASE+segcnt);s++)
      {
                     /* As gsscpy copies the segment so that the */
                     /* segment origin is at the current position*/
```

```
            gsqorg(s,&segx,&segy); /* first query the segment origin, */
            gsmove(segx,segy); /* then move to that point.           */
            gsscpy(s);    /* Copy all the segments loaded from the    */
                          /* ADMGDF file in the stream of primitives  */
                          /* making up segment 1.                     */
            gssdel(s);    /* Also delete the original loaded segment  */
                          /* once it has been included.               */
           }
        gssorg(1,50,50);  /* Set the segment origin, used for         */
                          /* rotation, to the middle of the screen.   */
        gsscls();         /* Close this segment.                      */
        rotate(angle,&rx,&ry); /* Call the function rotate to convert */
                          /* the angle specified by the user to a     */
                          /* point (rx,ry) used to define the segment */
                          /* transform.                               */
        gssaga(1,sx,sy,hx,hy,rx,ry,dx,dy,0); /* Rotate the segment by */
                          /* setting the segment geometric attributes.*/
        asread(&attype,&attval,&count); /* Display the transformed    */
                          /* segment and wait for user interrupt.     */
        strncpy(nlist[0],"PLOT    ",8); /* Initialize the name list   */
        strncpy(nlist[0],"FILE    ",8); /* for the dsopen. Use strncpy*/
                          /* to avoid copying the NULL from the end of*/
                          /* the string. The string is padded with    */
                          /* spaces up to 8 characters, because "PLOT"*/
                          /* would copy 'P','L','O','T',NULL to the   */
                          /* target string.                           */
        dsopen(99,1,"L3179G80",3,plist,2,nlist); /* Open a plotter    */
                          /* with procopts to direct the output to a  */
                          /* GL file. The name list will become the   */
                          /* name of the GL file. The device token    */
                          /* specifies the type of plotter the GL file*/
                          /* is being created for.                    */
        dsuse(2,99);      /* Use the plotter as the alternate device. */
        dscopy(100,100,0,0,1,opt); /* Copy the graphics from the      */
                          /* current page to the alternate device.    */
       }
     fsterm();            /* Terminate GDDM.                          */
    }


/* Rotate function: Returns a point (rx,ry) which can be used to    */
/* specify the rotation of a segment. The function accepts an angle */
/* specified in degrees. The returned point is the result of        */
/* rotating a point (1,0) by 'angle' degrees about the origin.      */
void rotate(float deg,float *rx,float *ry)
    {
     double angle;
     angle=((deg*3.1415)/180);
     *rx=cos(angle);
     *ry=sin(angle);

    }
```

A REXX programming example

This programming example allows the end user to draw lines and areas on the display screen by moving the cursor to the chosen end position of each line and pressing the ENTER key.

```
/* REXX */
/* Simple GDDM-REXX interactive graphics program                    */

Parse upper source opsys .

Signal on Error                        /* Set up error handling     */
Signal on Syntax                       /*                           */
Signal on Halt                         /*                           */

/* load and address GDDM-REXX */

If opsys='TSO' then Address link    'GDDMREXX INIT'   /* for TSO    */
If opsys='CMS' then Address command 'GDDMREXX INIT'   /* for CMS    */
Address gddm

/*******************************************************************/
/*     Query device to get screen size and mouse availability      */
/*******************************************************************/
"fsqury 0 3 2 .data0."
"fsqury 2 11 1 .data2."
oldx = 50
oldy = 50

/*******************************************************************/
/*           Put instructions on the screen                        */
/*******************************************************************/
row = data0.1
"asdfld 1 1 1 1 50 2"
"ascput 1 . 'Move cursor and hit Enter or Mouse key'"
"asfcol 1 1"
"asdfld 2 .row 1 1 36 2"
"ascput 2 . 'PF1=Area fill  PF3=End'"
"asfcol 2 1"

/*******************************************************************/
/*           Enable graphics input devices                         */
/*******************************************************************/
"gsenab 2 1 1"                  /* mouse or cursor                 */
"gsenab 1 0 1"                  /* Enter key                       */
"gsenab 1 1 1"                  /* PF keys                         */
"gsenab 1 4 1"                  /* PA keys                         */
"gsenab 1 5 1"                  /* Clear key                       */
if data2.1 > 5 then            /* If mouse buttons are available  */
  "gsenab 1 10 1"               /* enable them as choice devices   */
"fsenab 1 1"                    /* enable alphanumeric input       */
"fsenab 2 1"                    /* enable graphics input           */
```

```
/*******************************************************************/
/*           Main process loop                                     */
/*******************************************************************/
j=0
do i = 1 to 999
  "asread .attype .atval ."      /* update the screen              */
  if (attype = 1) & (atval = 3)  /* if user hit PF3 then exit      */
  then signal endit
  "gsread 0 .type .id"           /* get graphics input coordinates  */
  do while type ¬= 0
    if type = 2 then
      "gsqloc .win .inx .iny"
    "gsread 0 .type .id"
  end
  "gsseg .i"
  "gscol 6"
  "gsmove .oldx .oldy"
  "gsline .inx .iny"             /* Draw line to new location      */
  oldx = inx                     /* Save locations                 */
  oldy = iny
  savex.j = inx
  savey.j = iny
/*     Paint area (PF1 hit)      */
  if (attype = 1) & (atval = 1)
  then do
    if j <> 0 then do
      "gsarea 1"
      "gsmove .savex.0 .savey.0"
      "gsplne .j .savex. .savey."
      "gsenda"
      j=0
    end
  end
  "gsscls"
  j = j + 1
end

/*******************************************************************/
/*           Handle abnormal terminations                          */
/*******************************************************************/
Error:
Syntax:
 Say 'Exec ended abnormally. Return code' rc 'from line' sigl'.'
 Say  sourceline(sigl)
Halt:
Endit:
/* Terminate GDDM-REXX & GDDM */
If opsys='TSO' then Address link    'GDDMREXX TERM'   /* for TSO    */
If opsys='CMS' then Address command 'GDDMREXX TERM'   /* for CMS    */
Exit
```

A CICS pseudoconversational programming example

The following program shows a reentrant GDDM mapping application written as a CICS pseudoconversation. There are several points to note about the program:

- The program MENUP1 has been defined to CICS and associated with transaction ID DFP1.
- MENUP1 determines, from the absence or presence of the COMMAREA, whether this is the first time through the program.
- The first time through, DSOPEN is called with the PSCNVCTL,START processing option.
- Subsequent invocations call DSOPEN with the PSCNVCTL,CONTINUE processing option (this tells GDDM to retrieve the saved device information from temporary storage).
- All DSCLS calls except the last specify option 1. This tells GDDM not to erase the screen, but to unlock the keyboard (thus allowing input). It also tells GDDM to save, in temporary storage, all information about the device. This is required for GDDM to successfully re-initialize on the next invocation.
- Required ADS information is saved in the COMMAREA.

  GDDM saves all information concerning the nature of the device between transactions, but it is the responsibility of the application to save data required by the application.

```
MENUP1:           PROC(COMAP) OPTIONS(MAIN);
/*******************************************************************/
/* Test Program to display a set of panels using Mapping.          */
/* MENU00 is displayed first, and PF Keys 3 or 4 entered from this  */
/* panel causes the end of the application with either an erased    */
/* screen or not respectively.                                      */
/* Entering options '1', '2' or '3' from MENU00 causes the display  */
/* of MENUs 01, 02, and 03 respectively, each with their own        */
/* legends displayed in a color generated by the program.           */
/* MENU00 is then re-displayed after input.                         */
/*                                                                  */
/* This program is pseudoconversational.                            */
/*                                                                  */
/* The logic is as follows:                                         */
/*   On first invocation (COMMAREA length = 0)                      */
/*     Display MENU00                                               */
/*     Save Application data in the COMMAREA                        */
/*     Return to CICS requesting transaction DFP1 next time         */
/*   On subsequent invocations (COMMAREA length ¬=0)                */
/*     Restore Application data from COMMAREA                       */
/*     Re-define appropriate Map                                    */
/*     Receive Input                                                */
/*     If Finish not requested                                      */
/*       Display MENU00                                             */
/*       Return to CICS requesting transaction DFP1 next time       */
/*     Else                                                         */
/*       Return to CICS                                             */
/*******************************************************************/
DCL
  COMAP                 PTR;           /* COMMAREA PTR             */
%INCLUDE ADMUPIRA;
```

```
%INCLUDE ADMUPIRD;
%INCLUDE ADMUPIRF;
%INCLUDE ADMUPIRM;
DCL
  1 MENU00,                                 /* ADS */
    10 MSG_SEL                            CHAR(1),
    10 MSG_COL_SEL                        CHAR(1),
    10 MSG_COL                            CHAR(1),
    10 MSG_PS_SEL                         CHAR(1),
    10 MSG_PS                             CHAR(1),
    10 MSG                                CHAR(78),
    10 OPT                                CHAR(2),
  MENU00_ASLENGTH                       FIXED BIN(31,0) STATIC
                                        INIT(85);
DCL
  1 MENU01,                                 /* ADS */
    10 MSG_SEL                            CHAR(1),
    10 MSG_COL_SEL                        CHAR(1),
    10 MSG_COL                            CHAR(1),
    10 MSG_PS_SEL                         CHAR(1),
    10 MSG_PS                             CHAR(1),
    10 MSG                                CHAR(42),
  MENU01_ASLENGTH                       FIXED BIN(31,0) STATIC
                                        INIT(47);
DCL
  1 MENU02,                                 /* ADS */
    10 MSG_SEL                            CHAR(1),
    10 MSG_COL_SEL                        CHAR(1),
    10 MSG_COL                            CHAR(1),
    10 MSG_PS_SEL                         CHAR(1),
    10 MSG_PS                             CHAR(1),
    10 MSG                                CHAR(39),
  MENU02_ASLENGTH                       FIXED BIN(31,0) STATIC
                                        INIT(44);
DCL
  1 MENU03,                                 /* ADS */
    10 MSG_SEL                            CHAR(1),
    10 MSG_COL_SEL                        CHAR(1),
    10 MSG_COL                            CHAR(1),
    10 MSG_PS_SEL                         CHAR(1),
    10 MSG_PS                             CHAR(1),
    10 MSG                                CHAR(60),
  MENU03_ASLENGTH                       FIXED BIN(31,0) STATIC
                                        INIT(65);
DCL
  DEVID                 FIXED BIN(31) INIT(0),
  FAMID                 FIXED BIN(31) INIT(1),
  PCCNT                 FIXED BIN(31) INIT(2),
  NMCNT                 FIXED BIN(31) INIT(0),
  PCLSTS(2)             FIXED BIN(31) INIT(25,1),     /* START      */
  PCLSTC(2)             FIXED BIN(31) INIT(25,2),     /* CONTINUE   */
  DEVTK                 CHAR(8) INIT('*'),
  NMLST(1)              CHAR(8) INIT(' ');
DCL
  COPTES                FIXED BIN(31) INIT(0),
  COPTLS                FIXED BIN(31) INIT(1),
  COPTEU                FIXED BIN(31) INIT(2),
  COPTLU                FIXED BIN(31) INIT(3);
```

```
DCL TRANID              CHAR(8) INIT('DFP1');
DCL (ATYPE,AVAL,AMOD)   FIXED BIN(31);      /* I/P CONTROL FLDS     */
DCL FINISH              BIT(1) INIT('0'B);
DCL PICOPT              PIC'99';            /* NUMERIC OPTION       */
DCL AAB                 CHAR(8);            /* ANCHOR BLOCK         */
DCL MAPG                CHAR(8)             /* MAP GROUP NAME       */
                        INIT('DFMGC1D5');
DCL SSID                CHAR(1);            /* SYMBOL SET ID        */
DCL SSID_BIT         BIT(8) DEF(SSID);   /* SYMBOL SET ID        */
DCL X41                 BIT(8) INIT('01000001'B);
DCL MAP(0:3)            CHAR(8)             /* MAP NAMES            */
                        INIT('MENU00','MENU01','MENU02','MENU03');
DCL
  1 COMMAREA            BASED(COMAP),  /* COMMAREA                  */
   2 MAPNO              FIXED BIN(15), /* MAP NAME ARRAY NO         */
   2 COL                PIC'9',        /* CURRENT COLOR             */
   2 COUNT              FIXED BIN(31),
   2 PSSID              CHAR(1),
   2 CLSOPT             FIXED BIN(31);
/* CODE STARTS HERE */

CALL FSINIT(AAB);                           /* INIT GDDM            */
IF EIBCALEN = 0 THEN
  DO;
    /*************************************************************/
    /* SINCE NO COMMAREA EXISTS ALREADY, THIS MUST BE THE 1ST TIME   */
    /* THROUGH.                                                  */
    /*************************************************************/
    ALLOCATE COMMAREA;
    CALL DSOPEN(AAB,DEVID,FAMID,DEVTK,PCCNT,PCLSTS,NMCNT,NMLST);
                                      /* OPEN THE DEVICE SPECIFYING*/
                                      /* START PSEUDO-CONV         */
    SSID_BIT = X41;                        /* ITALICS ID           */
    PSSID = SSID;                     /* SAVE IT                   */
    CLSOPT = 1;                       /* DO NOT ERASE SCREEN       */
    MENU00 = '';                           /* CLEAR PRIMARY MENU   */
    MENU00.MSG_SEL = '1';             /* GET DATA FROM ADS         */
    CALL MSPCRT(AAB,1,-1,-1,MAPG);    /* PAGE CREATE               */
    CALL MSDFLD(AAB,1,-1,-1,'MENU00'); /* MAP MENU00               */
    CALL MSPUT(AAB,1,0,MENU00_ASLENGTH,MENU00);
                                      /* PUT DATA INTO MAP         */
    CALL FSFRCE(AAB);                 /* WRITE DATA TO SCREEN      */
    CALL DSCLS(AAB,DEVID,CLSOPT);     /* CLOSE THE DEVICE          */
    CALL FSTERM(AAB);                 /* END GDDM                  */
    MAPNO = 0;                        /* SAVE LAST MAP NO.         */
    COUNT = 0;                        /* INITIALIZE COUNT          */
    EXEC CICS RETURN TRANSID(TRANID) COMMAREA(COMMAREA);
  END;
ELSE                                  /* A COMMAREA EXISTS         */
  DO;
    /*************************************************************/
    /* GET I/P                                                   */
    /*************************************************************/
    CALL DSOPEN(AAB,DEVID,FAMID,DEVTK,PCCNT,PCLSTC,NMCNT,NMLST);
                                      /* OPEN THE DEVICE SPECIFYING*/
                                      /* CONTINUE PSEUDO-CONV      */
    IF MAPNO = 0 THEN
      DO;
```

```
    /*********************************************************/
    /* RESTORE MENU00                                        */
    /*********************************************************/
    MENU00 = '';
    MENU00.MSG_SEL = '1';
    CALL MSPCRT(AAB,1,-1,-1,MAPG);
    CALL MSDFLD(AAB,1,-1,-1,'MENU00');
    CALL MSPUT(AAB,1,0,MENU00_ASLENGTH,MENU00);
  END;
ELSE
  IF MAPNO = 1 THEN
    DO;
      /*********************************************************/
      /* RESTORE MENU01                                        */
      /*********************************************************/
      MENU01 = '';
      MENU01.MSG_SEL     = '2';
      MENU01.MSG_COL_SEL = '1';
      MENU01.MSG_COL     = COL;
      CALL MSPCRT(AAB,1,-1,-1,MAPG);
      CALL MSDFLD(AAB,1,-1,-1,'MENU01');
      CALL MSPUT(AAB,1,1,MENU01_ASLENGTH,MENU01);
    END;
  ELSE
    IF MAPNO = 2 THEN
      DO;
      /*********************************************************/
      /* RESTORE MENU02                                        */
      /*********************************************************/
        MENU02 = '';
        MENU02.MSG_SEL     = '2';
        MENU02.MSG_COL_SEL = '1';
        MENU02.MSG_COL     = COL;
        CALL MSPCRT(AAB,1,-1,-1,MAPG);
        CALL MSDFLD(AAB,1,-1,-1,'MENU02');
        CALL MSPUT(AAB,1,1,MENU02_ASLENGTH,MENU02);
      END;
    ELSE
      DO;
      /*********************************************************/
      /* RESTORE MENU03                                        */
      /*********************************************************/
        MENU03 = '';
        MENU03.MSG_SEL     = '2';
        MENU03.MSG_COL_SEL = '1';
        MENU03.MSG_COL     = COL;
        CALL MSPCRT(AAB,1,-1,-1,MAPG);
        CALL MSDFLD(AAB,1,-1,-1,'MENU03');
        CALL MSPUT(AAB,1,1,MENU03_ASLENGTH,MENU03);
      END;
CALL ASREAD(AAB,ATYPE,AVAL,AMOD);
                                /* GET I/P DATA              */
COL = MOD(COUNT,7) + 1;
COUNT = COUNT + 1;
IF MAPNO = 0 THEN
  DO;
    CALL MSGET(AAB,1,0,MENU00_ASLENGTH,MENU00);
```

```
IF ATYPE = 1 THEN
  DO;
    IF AVAL = 3 THEN
      DO;
        CLSOPT = COPTEU;
        FINISH = '1'B;
      END;
    IF AVAL = 4 THEN
      DO;
        CLSOPT = COPTLU;
        FINISH = '1'B;
      END;
  END;

IF ¬ FINISH THEN
  DO;
    IF OPT ¬= '01'
     & OPT ¬= '02'
     & OPT ¬= '03' THEN
      DO;
        MENU00.MSG = 'INVALID OPTION SELECTED';
        MENU00.MSG_COL_SEL = '2';
        MENU00.MSG_COL = COL;
        CALL MSPUT(AAB,1,0,MENU00_ASLENGTH,MENU00);
      END;
    ELSE
      DO;
        IF OPT = '01' THEN
          DO;
            MENU01 = '';
            MENU01.MSG_SEL     = '2';
            MENU01.MSG_COL_SEL = '1';
            MENU01.MSG_COL     = COL;
            CALL MSDFLD(AAB,1,-1,-1,'MENU01');
            CALL MSPUT(AAB,1,1,MENU01_ASLENGTH,MENU01);
            MAPNO = 1;
          END;
        ELSE
          IF OPT = '02' THEN
            DO;
              MENU02 = '';
              MENU02.MSG_SEL     = '2';
              MENU02.MSG_COL_SEL = '1';
              MENU02.MSG_COL     = COL;
              CALL MSDFLD(AAB,1,-1,-1,'MENU02');
              CALL MSPUT(AAB,1,1,MENU02_ASLENGTH,MENU02);
              MAPNO = 2;
            END;
          ELSE
            DO;
              MENU03 = '';
              MENU03.MSG_SEL     = '2';
              MENU03.MSG_COL_SEL = '1';
              MENU03.MSG_COL     = COL;
              MENU03.MSG_PS_SEL  = '1';
              MENU03.MSG_PS      = PSSID;
              CALL MSDFLD(AAB,1,-1,-1,'MENU03');
              CALL MSPUT(AAB,1,1,MENU03_ASLENGTH,MENU03);
```

```
                    MAPNO = 3;
                  END;
              END;
          END;
      END;
    ELSE
      DO;
        MENU00 = '';
        MENU00.MSG_SEL = '1';
        CALL MSDFLD(AAB,1,-1,-1,'MENU00');
        CALL MSPUT(AAB,1,0,MENU00_ASLENGTH,MENU00);
        MAPNO = 0;
      END;
    IF ¬FINISH THEN                    /* CONTINUE TRANSACTION      */
      CALL FSFRCE(AAB);                /* WRITE DATA TO SCREEN      */
    CALL DSCLS(AAB,DEVID,CLSOPT);      /* CLOSE THE DEVICE          */
    CALL FSTERM(AAB);                  /* END GDDM                  */
    IF ¬FINISH THEN
      EXEC CICS RETURN TRANSID(TRANID) COMMAREA(COMMAREA);

  END;
END;                                        /* - OF MENUP1          */
```

Appendixes

© Copyright IBM Corp. 1982, 1996

Appendix A. GDDM sample programs

This appendix describes the GDDM sample application programs that are supplied with this release of GDDM. These programs may be listed by licensees of GDDM as stated in the "Notices" on page xxiii section of this book.

These sample programs are supplied with the GDDM base product:

Sample 1 A program that draws a simple line graph. A version of this program is provided in each of these languages:

Language	Program name
C/370	ADMUSB1
COBOL	ADMUSC1
FORTRAN	ADMUSF1
PL/I	ADMUSP1

(A special PL/I version is provided for the IMS subsystem.)

Sample 2 A program to display an alphanumeric panel. A version of this program is provided in each of these languages:

Language	Program name
C/370	ADMUSB2
COBOL	ADMUSC2
FORTRAN	ADMUSF2
PL/I	ADMUSP2

Sample 3 A program that shows line types, colors, and patterns. Versions of this program are provided in PL/I, (ADMUSP3) and in C/370 (ADMUSB3).

Sample 4 ADMUSP4, which is a graphics editor program, written in PL/I, that allows pictures to be created. This sample program is designed to run on a workstation; the pictures created by this program can be drawn on a plotter attached to the workstation.

Sample 8 ADMUTMT (for TSO), and ADMUTMV (for VM/CMS), both provide a sample task manager that demonstrates the use of GDDM's windowing functions. Both versions of this program are written in PL/I.

You can find general information on how to compile, link-edit, and run the PL/I versions of samples 1, 2, 3, and 4 in "Compiling the programs" on page 524. Sample 8 is a special case and is discussed in "Sample program 8" on page 521.

REXX Six REXX sample programs are also supplied on the GDDM Base tape. You can invoke these programs by typing their names and pressing ENTER. The names of the REXX sample programs are as follows:

ERXMODEL
ERXPROTO
ERXTRY
ERXMENU
ERXOPWIN
ERXORDER

© Copyright IBM Corp. 1982, 1996

The functions of these programs are discussed in “REXX sample programs” on page 528.

Sample program 1

This program constructs a simple graph on a multicolored grid. The picture is displayed with an alphanumeric input field that requests the name of a printer (print file under VM/CMS). If a printer name is specified, the program generates a print data set comprising two copies of the graph, preceded by a header page. The program also saves the graphics output in a GDF file.

IMS version of sample program 1

The IMS version of this sample program has a slightly different interface. The printer LTERM name can be supplied on the transaction invocation. The program displays the picture and, if requested, copies it to a printer. The displayed picture contains an input alphanumeric field into which the next transaction code can be entered.

The source for the IMS version is named ADMUSP1I on the GDDM distribution library. The main procedure name is still ADMUSP1.

Sample program 2

This program displays an alphanumeric panel requesting the name of a GDF file to be loaded from auxiliary storage. The file generated by the program ADMUSx1, where “x” is C, F, or P, (called “sample1”) can be used; the original picture is then displayed again. After an interrupt, the original panel is redisplayed, awaiting new input. Pressing key PF3 or PF15 terminates the program.

IMS version of sample program 2

The IMS version of this sample program has a slightly different interface. The name of the saved picture is supplied as a parameter to the transaction. A second, optional, parameter names the LTERM to which the picture is to be sent. If this name is omitted, the picture is sent to the terminal that entered the transaction. As well as the picture, the program generates a simple alphanumeric menu, which is sent to the originating terminal. This contains a field into which the next transaction can be entered.

The source for the IMS version is named ADMUSP2I on the GDDM distribution library. The main procedure name is still ADMUSP2.

Sample program 3

This program uses GDDM to show:

- The 8 or 16 standard colors provided (depending on the display device)
- The 64 user colors in the supplied symbol set ADMCOLSD
- The 16 standard geometric shading patterns provided
- The 64 user geometric shading patterns in the supplied symbol set ADMPATTC
- The 8 standard line types provided
- The 2 standard line widths provided
- The 10 standard marker symbols provided

- A color-mixing table in mix mode

Each of these is shown on a separate GDDM display page. Displays can be viewed sequentially in the order given above, or individually by selection from a menu panel listing the various options. At any stage, a printed copy can be obtained by following the instructions generated at the bottom of each display.

Source for this sample program is provided in PL/I and in C/370. This sample program cannot be run under IMS.

Sample program 4

The ADMUSP4 sample program provides an example of a graphics editor program. Its purpose is to let the user create pictures that are made up of graphics lines, shaded boxes, and strings of text.

Although it is written in PL/I, it can be used as the basis for programs of a similar type in other programming languages.

What sample program 4 does

The program divides the screen into three parts:

- An information area, that contains prompts or messages from the program, and into which you can enter information.
- A drawing area, where you can draw primitives, and into which you can load already saved graphics (in ADMGDF format).
- A menu area, from which you can select various functions. Selections from the menu are made by moving the cursor and either pressing a button on the mouse or by pressing the ENTER key, depending on the device. Selections in this area result in further pull-down menus when more choices are required.

Invoking ADMUSP4

There are no special considerations for compiling, link-editing, and running this sample program, see “Compiling the programs” on page 524 or, if necessary, the appendix relating to the use of GDDM on your subsystem.

Sample program 8

Two versions of this sample program are provided, both written in PL/I. ADMUSTMT can be run on the TSO subsystem and ADMUTMV can be run on the VM/CMS subsystem. This program prompts the operator to select a program to run in a window. The sample program uses GDDM windowing calls and some sample Assembler routines, which are also supplied with GDDM, to perform the tasking functions. It supplies a running task manager under which you may run ADMUSP3 and ADMUSP4.

To compile, link-edit, and run sample program 8, you need to follow the procedure described below that pertains to your subsystem.

Compiling and link-editing sample program 8 under TSO

1. Assemble the assembler programs ADMUTMIT, ADMUTMTT, ADMUTMPT, ADMUTMAT, ADMUTMDT, ADMUTMST, and ADMUTMCT into an OBJ library.
2. Compile the PL/I program ADMUTMT into the same OBJ library.
3. Link-edit ADMUTMT into a LOAD library by using the following linkage editor control statements (which must start in column 2):

   ```
   INCLUDE SYSLIB(ADMUTMT)
   INCLUDE SYSLIB(ADMUTMIT)
   INCLUDE SYSLIB(ADMUTMTT)
   INCLUDE SYSLIB(ADMUTMPT)
   INCLUDE SYSLIB(ADMUTMAT)
   INCLUDE SYSLIB(ADMUTMDT)
   INCLUDE SYSLIB(ADMUTMST)
   INCLUDE SYSLIB(ADMUTMCT)
   NAME ADMUTMT(R)
   ```

4. Compile and link-edit the sample PL/I programs, ADMUSP3 and ADMUSP4 into the same LOAD library as ADMUTMT.

Running sample program 8 under TSO

If the terminal does not have a PA3 key, create a GDDM defaults file, containing the CTLKEY procopt to assign a PF or PA key to invoke User Control. The following profile statement (which starts in column 2) assigns PF2 for this purpose:

```
ADMMNICK FAM=1,PROCOPT=((CTLKEY,1,2))
```

To run the sample task manager use the commands:

```
ALLOC F(ADMDEFS)  DA(GDDM-defaults-file-name) REUS SHR
ALLOC F(ADMSYMBL) DA(GDDM-symol-sets-file-name) REUS SHR
CALL load-library-name(ADMUTMT)
```

Compiling sample program 8 under VM/CMS

Because the task manager program is written in PL/I, any other PL/I programs to be run under it must be link edited with PL/I on CMS before being loaded by the task manager, or else the load fails with duplicate PL/I main sections. Because of this, the program requires these special procedures for compiling, link-editing, and running:

1. Build ADMUTMV TXTLIB by assembling the assembler programs ADMUTMIV, ADMUTMTV, ADMUTMPV, ADMUTMAV, ADMUTMDV, ADMUTMSV, and ADMUTMCV.

   Use the following command to generate the TXTLIB:

   ```
   TXTLIB GEN ADMUTMIV ADMUTMTV ADMUTMPV ADMUTMAV ADMUTMDV ADMUTMSV ADMUTMCV
   ```

2. Compile the PL/I program ADMUTMV, but do not put it into the TXTLIB.
3. You can run other programs, such as ADMUSP3, ADMUSP4, or one of your own, from within the sample task manager. However, because the task manager uses the GDDM reentrant interface, and ADMUSP3 and ADMUSP4 use the GDDM nonreentrant interface, these programs must be run from a LOADLIB. First compile these PL/I programs and then build the LOADLIB as follows:

```
FILEDEF SYSLIB DISK ADMNLIB TXTLIB  *
LKED ADMUSP3 ( LIBE ADMUTMV
FILEDEF SYSLIB DISK PLILIB  TXTLIB  *
FILEDEF INCLIB DISK ADMUTMV LOADLIB A ( DSORG PO RECFM U
LKED INCUSP3 ( LIBE ADMUTMV
```

Where `INCUSP3 TEXT` contains these linkage editor control statements (which must start in column 2):

```
INCLUDE SYSLIB(DMSIBM)
INCLUDE INCLIB(ADMUSP3)
ENTRY DMSIBM
NAME ADMUSP3(R)
```

This procedure link-edits ADMUSP3 and places the load module in ADMUTMV LOADLIB. The same procedure must be repeated for ADMUSP4.

Running sample program 8 under VM/CMS

If the terminal does not have a PA3 key, create a GDDM defaults file, PROFILE ADMDEFS, containing the CTLKEY processing option to assign a PF or PA key to invoke User Control. The following profile statement (which starts in column 2) assigns PF2 for this purpose:

```
ADMMNICK FAM=1,PROCOPT=((CTLKEY,1,2))
```

To run the sample task manager use the commands:

```
GLOBAL LOADLIB ADMUTMV
GLOBAL TXTLIB  ADMUTMV ADMRLIB ADMHLIB ADMGLIB ADMPLIB PLILIB CMSLIB
LOAD ADMUTMV ( START
```

Using the sample task manager

The sample task manager displays a panel asking you to select which program to run from a menu of programs. Select one.

Whenever the selected program is waiting for input from you, you can call up User Control by pressing PA3 – or the alternative key as defined by the CTLKEY processing option. Make the task manager window active by using the NEXT function and END the User Control session. You can then start to run another program from the menu. You can even run the same program again so that it appears in more than one window.

At any time a program is waiting for input from you, you can call up User Control to move or size the operator windows, or make a different operator window active.

To end the sample task manager, first end each program, then end the task manager.

You can change the sample task manager menu so that it runs your own programs. For more information, see the prolog of the ADMUTMT or ADMUTMV programs.

Compiling, link-editing, and running the sample programs

With the exception of ADMUTMT and AMDUTMV, the programs should be compiled, link-edited, and run as follows.

Note: See also these appendixes for more detail:

- Appendix B, “Programming with GDDM under VM/CMS” on page 529
- Appendix C, “Programming with GDDM under TSO” on page 539
- Appendix D, “Programming with GDDM under IMS” on page 553
- Appendix E, “Programming GDDM applications for use with CICS” on page 567

Compiling the programs

The source programs do not need to be modified except for:

- Optional changes to the FSINIT call, as noted under “Link-editing the programs” on page 525.
- Replacing the STOP RUN statements in the COBOL programs if they are to run under CICS. The statements should be replaced with GO BACK or EXEC CICS RETURN.
- Modifying ADMUSP3 if it is to be run on a device with less than 32 rows.

The programs must be compiled by a compiler appropriate to the source language and target subsystem (for example VS COBOL II, VS FORTRAN, PL/I Optimizing Compiler, or C/370). Note that CICS does not support programs written in FORTRAN.

The PARM options RESIDENT and DYNAMIC must be explicitly set to NORESIDENT and NODYNAMIC.

ADMUSP1, ADMUSP3, and ADMUSP4 use the supplied files of GDDM PL/I entry declarations. The members (containing PL/I declarations for nonreentrant base functions) must be available to the compiler in a source statement library under DOS/VSE, by means of SYSLIB specification under MVS, or by means of a GLOBAL MACLIB command under VM/CMS. The compilation of ADMUSP1, ADMUSP3, and ADMUSP4 must be performed with the MACRO option. No errors should result from the compilation steps.

ADMUSP3 is written to run on a device with at least 32 rows. However, because it is only the initial menu panel that requires more than the 24 rows available on an IBM 3278/3279 Model 2, the program can be run on this and other devices if the following change is made:

- Amend the initial value in the second column of “FIELD_DEF” to:

  `(2,3,4,6,8,10,12,14,16,18,20,24,1,22,5)`

- Change the initial value in the second column of “PRINT_DEF” to:

  `(2,5,7,7,24)`

For information on the ADMUTMT/V sample programs, see “Compiling and link-editing sample program 8 under TSO” on page 522, and “Compiling sample program 8 under VM/CMS” on page 522.

Link-editing the programs

Except under CMS, the object code from the compilation must be link-edited with a GDDM interface routine appropriate to the subsystem and to the interface used (reentrant or nonreentrant).

Under MVS, the link-edit SYSLIB concatenation must include the GDDM SADMMOD data set. The correct interface module is selected by an INCLUDE control statement specifying the appropriate member, as shown in Table 10.

Or, the automatic-library-call facility can be used. For this, the source programs must be changed to replace the references to FSINIT with the appropriate alternative, as shown in Table 11 on page 526. (However, this is not necessary for ADMUTMT/V as they can only run under CMS and TSO and they are coded with FSINR already.)

Note that for PL/I, the standard declarations do not include the alternative forms of FSINIT. They must, therefore, always be explicitly declared thus:

```
DCL FSINNC ENTRY EXTERNAL OPTIONS (ASM INTER);
```

Under VSE, GDDM must be included from the relocatable libraries during link-editing.

The correct interface modules should be selected as shown in Table 12 on page 526 and should be included as described in Appendix B, “Programming with GDDM under VM/CMS” on page 529. The automatic inclusion of the interface modules by source-program modification is not available under VSE.

Table 10. GDDM load library for link-edit SYSLIBs

Interface	Sample programs	Required member for subsystem		
		CICS/ESA	IMS	TSO
Nonreentrant	ADMUSB1 ADMUSC1 ADMUSF1 ADMUSF2 ADMUSP1 ADMUSP3 ADMUSP4	ADMASNC	ADMASNJ	ADMASNT
Reentrant	ADMUSB2 ADMUSC2 ADMUSP2	ADMASRC	ADMASRJ	ADMASRT

Table 11. GDDM automatic library calls

Interface	Sample programs	Replace FSINIT references by the following for each subsystem		
		CICS/ESA	IMS	TSO
Nonreentrant	ADMUSB1 ADMUSC1 ADMUSF1 ADMUSF2 ADMUSP1 ADMUSP3 ADMUSP4	FSINNC	FSINNPI	FSINN
Reentrant	ADMUSB2 ADMUSC1 ADMUSP2	FSINRC	FSINRPI	FSINR

Table 12. GDDM interface modules

Interface	Sample programs	CICS/VSE modules
Nonreentrant	ADMUSB1 ADMUSC1 ADMUSF1 ADMUSF2 ADMUSP1 ADMUSP3 ADMUSP4	ADMASNB and ADMASLC
Reentrant	ADMUSB2 ADMUSC1 ADMUSP2	ADMASRB and ADMASLC

Table 13. GDDM global TXTLIBs

Interface	Sample programs	Required VM/CMS library
Nonreentrant	ADMUSB1 ADMUSC1 ADMUSF1 ADMUSF2 ADMUSP1 ADMUSP3 ADMUSP4	ADMNLIB
Reentrant	ADMUSB2 ADMUSC1 ADMUSP2	ADMRLIB

Note: Plus,

- If DCSS is available.....no extra TXTLIBs
- If no DCSS is available.....ADMHLIB and ADMGLIB

Under CMS, there is no link-editing. However, the CMS GLOBAL TXTLIB command must be executed as described in Appendix B, “Programming with GDDM under VM/CMS” on page 529 to identify TXTLIBs from which GDDM routines can be loaded. The TXTLIBs required depend on the sample program

attributes and the presence of GDDM in a Discontiguous Shared Segment (DCSS). See Table 13.

For information on the ADMUTMT/V sample programs, see “Compiling and link-editing sample program 8 under TSO” on page 522, and “Compiling sample program 8 under VM/CMS” on page 522.

Running the sample programs

Note that the COBOL programs must not be run under CICS unless the STOP RUN statements have been replaced by a GO BACK statement or an EXEC CICS RETURN.

When the programs are run, the GDDM load (or core-image) library must be available. The same library is used for nonreentrant and reentrant programs. The first two sample programs make use of a file containing saved data streams. Except under VM/CMS, this file must be created before running the programs. The first program (ADMUSC1, ADMUSF1, ADMUSP1) also optionally generates a print file.

Under CICS, the programs must be added to the program control table (PCT) and processing program table (PPT). The GDDM load library (or core-image library) must be specified when CICS is started. For the saved data stream, the GDDM VSAM file (by default, ADMF) must have been created and entered in the CICS file control table (FCT); this is part of the installation procedure.

Under IMS, the programs must be added to the IMS program library, and the transaction codes and ACB set up during IMS system definition. Also, a database must be assigned and initialized to contain the saved data stream. These actions are part of the installation procedure.

Under TSO, the GDDM load library should be available (for example, in a STEPLIB). It is also necessary to have created a partitioned data set to contain the saved GDF data. As specified in Appendix C, “Programming with GDDM under TSO” on page 539, this has a record length of 400. A suitable space allocation for the program is one directory block and 100 400-byte data blocks. The DDname ADMGDF should be allocated to the data set before execution. If a print is requested, the print queue data set (ADMPRINT.REQUEST.QUEUE) must have been created and initialized.

Under VM/CMS, the GDDM TXTLIBs must be included in the GLOBAL libraries during execution, as described above, together with the language libraries. Program loading may be prolonged if a module is not generated. Files containing saved GDF and print data are generated dynamically by GDDM.

To allow enough storage for GDDM, PL/I execution must specify ISASIZE; a value of 10K bytes is usually sufficient.

For information on the ADMUTMT/V sample programs, see “Running sample program 8 under TSO” on page 522, and “Running sample program 8 under VM/CMS” on page 523.

REXX sample programs

The following sample execs are supplied for use by programmers at installations where GDDM's support for REXX has been enabled.

ERXMODEL This program displays a saved GDDM picture and adds some alphanumeric and graphics text. The program provides a model for the structure of a GDDM program written in REXX. You can copy this exec and use this structure for your own programs.

ERXPROTO If you pass the name of a GDDM API call to this exec, as a run-time parameter, it shows you the syntax of the call in REXX and defines the call's parameters. You can use this exec from within a file you are editing and then substitute values or variable names into the parameters.

ERXTRY This is an interactive exec that enables you to try out GDDM calls and see their effects as you issue them. It is very useful for learning to program with GDDM.

ERXMENU This is a REXX version of the program in Figure 76 on page 257. It uses alphanumeric and graphics text to display a menu from which the end user selects dishes. It then adds up the cost of the meal and recommends a bottle of wine.

ERXOPWIN This exec is a REXX version of the program in Figure 132 on page 486. It enables the end user to display two saved pictures on the screen, each in a different operator window.

ERXORDER This exec demonstrates how you can use mapped alphanumerics in REXX applications. It enables end users to order components from a list and calculates the cost of their orders.

Appendix B. Programming with GDDM under VM/CMS

This appendix describes the use of GDDM under the VM/CMS operating system. It contains the following topics:

- Compiling, loading, and running a GDDM PL/I application program
- Running a GDDM utility program
- Data sets and file processing
- Display terminal conventions
- Using APL terminals
- Batch processing
- Running programs under VM/XA

Invoking the GDDM print utility is described in the *GDDM Base Application Programming Reference* book.

Note: GDDM cannot be run in the VM CMS/DOS environment. Therefore, it cannot be successfully invoked under VM/CMS by application programs compiled using DOS compilers such as the PL/I DOS Optimizing Compiler.

When writing an application program, you must access MACLIBs to compile your programs, if you are to include the GDDM standard declarations. You must also access TXTLIBs to load your program, and possibly to run your program, as described in “How to compile, load, and run a PL/I GDDM application program.”

How to compile, load, and run a PL/I GDDM application program

1. Link and access the disks that hold GDDM and the PL/I compiler at your installation.
2. If you use the GDDM-supplied declarations in your program, you must link the macro library containing the GDDM entry-point declarations, using this command:

   ```
   GLOBAL MACLIB ADMLIB
   ```

3. Invoke a PL/I compiler to compile the program. This example invokes the PL/I Optimizing Compiler, passing it the name of the file containing the program.

   ```
   PLIOPT filename (INCLUDE FLAG(I)
   ```

   The INCLUDE option is required to pick up the GDDM entry points for calls used in the program.

   The FLAG(I) option is not essential, but it ensures that useful messages about dummy variables are not suppressed. These are issued when parameters do not match GDDM’s requirements exactly.

4. Specify the run-time libraries to be used by the program. Before loading a VM/CMS application, the CMS GLOBAL command must be executed to identify the appropriate GDDM TXTLIB to be searched for GDDM function references.

   The GDDM TXTLIB to be specified in the CMS GLOBAL command depends on the type of GDDM interface being used, as follows:

© Copyright IBM Corp. 1982, 1996

Interface	GDDM TXTLIB
Nonreentrant	ADMNLIB
Reentrant	ADMRLIB
System programmer	ADMRLIB

The command takes the form:

```
GLOBAL TXTLIB ADMxLIB
```

where ADMxLIB is one of the TXTLIBs above.

If all the required run-time GDDM facilities have been made available in a VM/CMS Discontiguous Shared Segment (DCSS) as described in the *GDDM/VM Program Directory*, ADMxLIB is all you need to specify. Otherwise, before running a GDDM application program or utility, the CMS GLOBAL command must also identify appropriate GDDM TXTLIBs to be searched for routines required dynamically during execution.

If only GDDM/VM ("GDDM Base") has been installed, the installation procedure will have placed the required routines in ADMGLIB TXTLIB.

If GDDM-PGF has also been installed, the installation procedure will have placed additional GDDM-PGF routines in ADMPLIB TXTLIB.

The ADMHLIB TXTLIB contains language-dependent routines for each supported national language so ADMHLIB must be specified as a run-time library for every program.

Depending on which GDDM products have been installed, the CMS GLOBAL command to be executed is:

GDDM Base only:

```
GLOBAL TXTLIB ADMxLIB ADMHLIB ADMGLIB
```

GDDM Base and GDDM-PGF:

```
GLOBAL TXTLIB ADMxLIB ADMHLIB ADMPLIB ADMGLIB
```

5. Load the program into storage. On the LOAD command, specify the name of the TEXT file generated by the complier followed by any options you select:

   ```
   LOAD txt_file_name (option1 option2
   ```

6. Start the program running. You do this by specifying the entry point for the application on the START command followed by any options you select:

   ```
   START appl_entry_point (parameter
   ```

   You can specify `*` on the START command, which instructs CMS to find the entry point itself.

   Note: If you do not need to pass any parameters to the application on the START command, you can combine the last two steps of this process by specifying START as an option of the LOAD command.

   ```
   LOAD txt_file_name (START
   ```

Running a GDDM utility program

GDDM utility programs are supplied in source form. The steps for compiling and running a utility are the same as those described under "How to compile, load, and run a PL/I GDDM application program" on page 529. Systems-support personnel can compile the programs and create a module which end users can invoke simply by typing the name and pressing the ENTER key.

Considerations for running multiple instances of GDDM

An application using the reentrant or system programmer interface to GDDM may invoke more than one instance of GDDM concurrently. Such an application should ensure that the first instance of GDDM to be initialized (using FSINIT or SPINIT) is also the last to be terminated (using FSTERM). This prevents any GDDM Shared Segment (DCSS) being unloaded prematurely.

Native CMS files

Table 14 (Page 1 of 2). GDDM data-set characteristics for VM/CMS

Type of data	GDDM default filetype	Record format (RECFM)	Record length (LRECL)
Symbol sets	ADMSYMBL	F	400
Pictures	ADMSAVE	F	400
Generated mapgroups	ADMGGMAP	F	400
GDF files	ADMGDF	F	400
Text files	ADMDECK	F	80
System printer output	ADMLIST (but directed to virtual printer by default)	V	according to device characteristics
Family-4 output	ADMCOLn or ADMIMAGE	V	≤ 2000 (for 4250) ≤ 8202 (for 38xx)
PostScript family-4 output	ADMIMAGE	V	≤ 1024
4250 printer fonts (see Note)	FONT4250	V	≤ 2048
4250 printer code pages (see Note)	FONT4250	V	≤ 2048
Queued printer files	ADMPRINT	F	80
GL plot files	(none)	V	1020
Trace records	ADMTRACE (default filename is ADM00001)	V	≤ 121
External default files	ADMDEFS (default filename is PROFILE)	F or V	≤ 256
Image files	ADMIMG	F	400
Image Projection Files	ADMPROJ	F	400
CDPDS	LISTCDP	V	≤ 8200
AFPDS	ADMIMAGE	V	≤ 8202

Table 14 (Page 2 of 2). GDDM data-set characteristics for VM/CMS

Type of data	GDDM default filetype	Record format (RECFM)	Record length (LRECL)
CGM files (output)	(none)	F	400
CGM files (input)	(none)	F or V	≤8000
CGM profiles	ADMCGM	F or V	≤256
GIF files (output only)	GIFBIN	F	80

Note: 4250 printer fonts and code pages are referenced by GDDM and are supplied as part of the 4250 typographical fonts licensed programs (program numbers 5771-AAA through 5771-AAW, and 5771-ACx, where x varies).

GDDM stores and retrieves data using CMS file identifiers where, by default:

filename Determined according to the type of data, as follows:

- For symbol-sets, pictures, generated mapgroups, ADMGDF files, GDDM print files, CGM files, CGM profiles, CDPU input files, AFPDS print files, and 4250 printer fonts and code pages, the file names used are those specified in the corresponding GDDM calls as symbol-set names, picture names, group names, ADMGDF file names, print-destination names, device names, and code-page names, subject to modification of these names by character-substitution rules.
- For text files generated from symbol sets, the file names used are those specified through the symbol editor. Each text file generated contains a correspondingly-named control section (CSECT), and is in a form suitable for link-editing with an application program for subsequent reference, typically by the GSDSS or PSDSS call.
- For trace output, the file name used is as defined in Table 14 on page 531 or as modified by the user in the CMSTRCE option in the current GDDM external defaults; see the *GDDM Base Application Programming Reference* book.
- For External Defaults File input, the file name used is as defined in Table 14 on page 531 or as modified by the user in the CMSDFTS option in the current GDDM external defaults; see the *&bapr.* manual.

filetype Determined by the GDDM default name (see Table 14 on page 531) or as modified by the user in the current GDDM external defaults, see the list of external defaults for CMS in the *GDDM Base Application Programming Reference* book. In the case of CGM files and CDPU input files, the file type is provided on the corresponding GDDM call.

filemode

"A1" for output, causing data to be stored on the A-disk (which should be accessed as read/write for such operations).

"★" for input, causing accessed data to be searched in the standard order. In the case of CGM files and CDPU input files, the file mode is provided on the corresponding GDDM call.

The DSOPEN call allows the file names, file types, and file modes of queued printer, system printer, and high-resolution image (family-4) disk file devices to be explicitly specified by means of the name-list parameter.

The Interactive Chart Utility (part of GDDM-PGF) includes a directory function that supports list, delete, and copy operations on GDDM objects such as symbol sets, pictures, generated mapgroups, and ADMGDF files.

Native CMS spool files

GDDM writes 3270 device (family-1) output either directly to a 3270-type terminal or to the virtual punch, according to the name specified in the DSOPEN call. 3270 device output written to a virtual punch is in the form of 80-byte records in the following format:

Record 1 Virtual CCW (8 bytes) including SIO count. The CCW opcode is one of the following:

- **X'01'** Write
- **X'05'** Erase/Write
- **X'0D'** Erase/Write Alternate
- **X'11'** Write Structured Field.

Record 2 Data stream – as many 80-byte records as are necessary to contain “SIO count” bytes of data.

Record n Virtual CCW (8 bytes) including SIO count.

Record n+1 Data stream – as many 80-byte records as are necessary to contain “SIO count” bytes of data.

CP SPOOL and CP TAG commands should be used to direct the virtual punch output to a destination that is capable of processing data in the above format (such as RSCS Networking Version 2). The CPSPOOL and CPTAG processing options in DSOPEN can be used to issue such commands automatically.

GDDM writes System Printer output either to a disk file or to the virtual printer, according to the name specified by the DSOPEN call. Data written to a System Printer device contains ASA control characters and, for 3800 devices, Translation Reference Characters (TRCs). The CP SPOOL and CP TAG commands should be used to specify additional special parameters such as CHARS, FLASH, or FCB that may be required for 3800 devices.

GDDM writes trace output either to a disk file or to the virtual printer, according to the file name defined in the current GDDM external defaults (or modified in the CMSTRCE option). If the file name is defined as all blanks, GDDM directs the trace output to the virtual printer.

Display terminal conventions

The following comments apply only when the display terminal being used is the CMS user virtual console.

Under VM/CMS, by default, the PA1 and PA2 keys are processed separately from other terminal input. The effect of using these keys is as follows:

PA1 Pressing this key causes CP mode to be entered and a CP READ status to be displayed. In this environment, any CP commands may be issued. To return from the CP environment, issue the CP command BEGIN.

PA2 Pressing this key causes the CMS SUBSET environment to be entered and a RUNNING status to be displayed. In the CMS SUBSET environment, any CMS commands that run in the transient area may be issued. For example:

```
ACCESS    LISTFILE    RENAME
CP        PRINT       RETURN
DISK      PUNCH       SET
ERASE     QUERY       STATE
EXEC      READCARD    TYPE
```

To return from the CMS SUBSET environment, issue the CMS SUBSET command RETURN.

On return from the CP or CMS SUBSET environment, GDDM retransmits the screen buffer contents, and then waits for more input.

As a result of the above special processing, PA1 and PA2 cannot, by default, be returned as terminal input by the ASREAD, GSREAD FSSHOR, or MSREAD calls. However, the CMS PA1/PA2 protocol option of the DSOPEN function can be used to suppress this special processing selectively. The use of this option to the DSOPEN function is described in the *GDDM Base Application Programming Reference* book.

PA3 The default action when pressing this key is to activate user control. If user control is not available, or if a key other than PA3 has been designated for activating user control, then PA3 causes the screen to be refreshed. PA3 is never passed to the application.

Interception of PA1 Programs that request (with the GDDM CMSINTRP processing option) that PA1 key interrupts be passed to them causes the CP TERMINAL BRKKEY value to be set to NONE, regardless of its original setting. This action is consistent with that of CMS when its full-screen mode is entered.

Asynchronous interrupts on VM/CMS

The following comments apply only when the display terminal being used is the CMS user virtual console.

Your application program can enable or disable asynchronous interrupts from the CMS console using the FSENAB call (see the *GDDM Base Application Programming Reference* book).

Application programs that need to check whether the level of GDDM on their system supports disabling of asynchronous console interrupts can use this sequence of calls:

```
CALL FSEXIT(0,12);            /* only display severe error messages  */
CALL FSENAB(4,0);             /* disable CMS asynchronous interrupts */
⋮
                              /* check return code from FSENAB, and  */
                              /* if FSENAB(4,0) is not supported, do */
                              /* appropriate processing              */
⋮
CALL FSEXIT(0,4);             /* restore normal error handling       */
```

Using the ENTER key

Unless the application program has established any special attention-processing functions, the ENTER key (and no other attention key) may be used while GDDM is operating to cause an asynchronous CMS attention interrupt. This suspends the operation of both the application program and GDDM, and causes control to be passed to the terminal user, with the terminal in line-by-line VM READ mode.

In this mode, normal CMS protocols usually allow the terminal user to take one or more of the following actions:

- Resume at the point of interruption, by pressing the ENTER key.
- Enter an “immediate” CMS command (such as, HI, HO, HT, HX, RO, RT, or SO).
- Enter other commands – such commands are stacked for execution at the next entry into normal CMS or CMS SUBSET mode.

After any of the above actions (except HX), GDDM ensures that the screen buffer contents are restored.

Using other attention keys

Application programs can request extended processing of asynchronous interrupts by specifying the CMS attention handling option (processing option group 1001) of the DSOPEN call.

Requesting “extended attention handling” indicates that an application program attention feedback block may have been located by means of the DSOPEN CMS attention option.

If this is done, an attention key may be used while GDDM is operating to cause an asynchronous CMS attention interrupt (unless a line-by-line message has already placed the terminal into line-by-line mode, in which case, only the ENTER key causes an attention interrupt). An exception is the PA1 key, which causes CP mode to be entered, unless the PA1 special processing was suppressed as described above.

Also, if the attention feedback block is of nonzero length, GDDM stores up to two words of information in this block (according to the length specified), indicating the nature of the interrupt. The information stored is as follows:

Attype attention type (fullword integer)
Attval attention type value (fullword integer).

where these parameters are as defined for the ASREAD call (see the *GDDM Base Application Programming Reference* book).

An application program may intercept such attention interrupts by establishing a special attention-processing exit using the VM/CMS simulation of the TSO STAX macro. A STAX exit of this form should be established before the device representing the virtual console is initialized (that is, before SPINIT/DSOPEN), and should not be cleared until after the device has been terminated (that is, after FSTERM/DSCLS). A STAX exit may examine the contents of the attention feedback block to determine the cause of the interrupt. GDDM must not be invoked from a STAX exit if GDDM was already running at the time of the interrupt.

GDDM disables all STAX exits and attention-processing functions before initiating the CMS SUBSET environment, and restores them on return.

VM-initiated asynchronous interrupts

VM/CMS may generate “virtual” asynchronous interrupts before the display of a priority message.

If such an interrupt occurs while the terminal user is entering data in response to an ASREAD, GSREAD FSSHOR, or MSREAD call, GDDM allows the priority message to be displayed immediately, but saves and restores any data entered by the terminal user. An interrupt occurring at this time may also cause any application program attention-processing exit to be entered, with an attention feedback block indicating an interrupt of type 6 (“Undefined”).

VM-initiated asynchronous interrupts are not otherwise apparent to the GDDM terminal user or application program.

Interactions with non-GDDM device interrupt handling

An application program that uses GDDM to communicate with the CMS virtual console and uses the CMS HNDINT macro as part of its own interrupt handling for devices not controlled by GDDM must be written in such a way as to avoid recursion of the CMS HNDINT macro.

If the virtual console operator causes an asynchronous attention interrupt, GDDM’s STAX exit gains control. This exit attempts to read from the terminal to determine the nature of the interrupt. During this processing, GDDM issues a CMS HNDINT WAIT macro.

If the application program already has a CMS HNDINT WAIT macro active at the time, interference between the macros occurs, and the application program’s HNDINT WAIT macro is likely to complete immediately, with random results.

To prevent this type of interaction, the application program should suppress GDDM’s STAX exit (and the attention-processing functions that go with it) over the duration of its own HNDINT WAIT macro. The application program can do this by clearing (and saving) the value in the TAXEADDR field in the CMS Nucleus Constant Area (NUCON) before invoking HNDINT WAIT and by restoring the value in TAXEADDR after the HNDINT WAIT macro has completed.

Dialed devices

If GDDM is used to drive a dialed display device, then when that device is closed it is also dropped from the virtual machine. This is due to a feature of the CMS Console Services support that causes a dialed device to be dropped when the last console path to it is closed.

Using APL terminals

This section describes how GDDM interacts with nonqueriable displays and printers that have the APL feature.

Using nonqueriable displays with the APL feature

Under VM/CMS, device information provided by the subsystem does not indicate whether a nonqueriable 3278 or 3279 display has the appropriate APL feature. (A "queriable" terminal is one that supports the Read Partition (Query) structured field.)

If the CP TERM APL ON command was issued, GDDM assumes by default that such a device has the APL feature, and selects an appropriate set of translation tables. (For more information, see the *GDDM System Customization and Administration* book and the description of ASTYPE in the *GDDM Base Application Programming Reference* book.) If the device does not have the APL feature, the use of character code points corresponding to APL characters may result in wrong output at the device.

If the CP TERM APL OFF command was issued, GDDM assumes that such a device does not have the APL feature.

The GDDM default can be overridden in either of the following ways. The application program can:

- Specify an explicit device token (for example, ADMK7720) in a DSOPEN call to initialize the device or by means of nickname facilities (see "Coding a partial device definition for end users to change with nicknames" on page 374).
- Use the ASTYPE call to specify the appropriate set of translation tables, as follows:

Device type	Translation type number
3278, 3279	3279
3278-APL, 3279-APL	32791

For a description of alphanumeric translation tables, see the *GDDM System Customization and Administration* book.

Using nonqueriable printers with the APL feature

Under VM/CMS, device information provided by the subsystem does not distinguish between IBM 3270 printers, unless they are "queriable" (that is, unless they support the Read Partition (Query) Structured Field).

By default, GDDM assumes that any APL feature on a nonqueriable printer is the APL/Text Feature, rather than the Data Analysis – APL Feature. If a printer (such as an IBM 3284 or 3286) has the Data Analysis – APL Feature, and if the APL character set is to be referenced, the GDDM default assumption must be overridden to ensure correct operation of the device.

The CMSAPLF option in GDDM's external defaults can be modified (by specifying the value DATAANAL) to cause GDDM to assume by default that an APL feature installed on a nonqueriable IBM 3270 printer terminal is the Data Analysis – APL Feature. This option can be specified:

- In an External Defaults Module, or
- In a User Defaults (ADMDEFS) File.

See the *GDDM System Customization and Administration* book.

Batch processing

A disconnected Virtual Machine, such as a machine using the CMS batch facility, can simulate batch processing. In such an application, you **cannot** communicate with the default primary device because there is no such device. The application must use DSOPEN to indicate the device that is to be used; for example:

- A dummy device
- A queued printer
- A high-resolution image file
- A dialed-in display station
- An attached printer.

In batch processing, an application might:

- Create queued printer output for subsequent printing by the GDDM print utility. The queued printer output would, perhaps, be created by using the chart utility noninteractively.
- Create a high-resolution image file for a family-4 device.
- Create FSSAVE files for subsequent interactive use with FSSHOW. The files would be created by using a dummy device.

GDDM application programs under VM/XA

The GDDM Base product, GDDM/VMXA, enables GDDM and application programs to be bigger than 16MB. Generally, programming for GDDM/VMXA is no different from programming for GDDM/VM, although there are a few special considerations.

Migration: To run under VM/SP, modules must be generated with GDDM/VM. To run under VM/XA SP, modules may be generated with either GDDM/VM or GDDM/VMXA.

Modules generated by either GDDM/VM or GDDM/VMXA can be run under VM/XA SP, except that programs generated under GDDM Release 2 Version 1 or earlier must be regenerated if they are transferred to a VM/XA system with GDDM/VM or GDDM/VMXA at Release 2 Version 2 or later.

User exits: Programmers should take care when specifying the addresses of user exits to GDDM. GDDM uses the convention that the top bit of such addresses identifies its addressing mode (AMODE). Also, if GDDM is initialized with the SPINIT call, and this call was issued in 24-bit mode, GDDM clears bits 1 through 7 of each address word that it processes.

Appendix C. Programming with GDDM under TSO

This appendix describes the use of GDDM under the TSO operating system. It covers these topics:

- Link-editing a GDDM application program
- Data sets and file processing
- Display terminal processing
- Using APL terminals
- Using GDDM under TSO or MVS batch
- An example of JCL for link editing GDDM applications.

Application programs using GDDM have no particular restrictions or requirements. However, if a PL/I program uses the GDDM-supplied declarations it must have access to the library on which they are held. It must also be link-edited with one of the interface modules as described below.

Terminal users should be aware of the GDDM usage of PA1, PA2, and the CLEAR keys. Also, there is a possibility of unexpected terminal responses after a GDDM application program has ended abnormally. These matters are described under "Display terminal processing" on page 544.

Link-editing a GDDM application program

An example of the JCL that can be used to link-edit GDDM application programs is listed in "Example: JCL for link-editing GDDM applications under TSO" on page 552.

Unless the application program uses dynamic-load facilities to access GDDM by means of the system programmer interface (see below), an application program using GDDM under TSO must be link-edited with an appropriate GDDM interface module. This interface module can be specifically included in the link-edit process. Alternatively, if the application program uses one of the other FSINIT entry points described in the *GDDM Base Application Programming Reference* book, the required GDDM interface module can be included by linkage-editor automatic library-call facilities.

This is a list of the GDDM interface modules for TSO:

Interface	**Interface module**	**FSINIT alternative entry**
Nonreentrant	ADMASNT	FSINN
Reentrant	ADMASRT	FSINR
System programmer	ADMASPT	–

Using the system programmer interface by means of dynamic load

If an application program uses only the System Programmer Interface, all invocations of GDDM are through the entry point ADMASP. This entry point can be resolved by link-editing the application with the GDDM interface module ADMASPT, as described above.

© Copyright IBM Corp. 1982, 1996

Alternatively, the application can avoid these linkage-edit considerations by using system facilities (the OS LOAD function) to load dynamically a GDDM interface module ADMASPLT. The main entry point for this module is defined with both names: ADMASP and ADMASPLT.

Note: If an installation uses the OS LOAD function to load GDDM dynamically, applications that use GDDM's system programmer interface cannot use the ADMUFO to bypass parameter checking.

Data sets

When running under TSO, GDDM-Base and GDDM-PGF use three types of data sets:

- Partitioned data sets
- Sequential data sets and SYSOUT classes
- Direct access data sets, such as the Master Print Queue data set used to control queued printer devices.

GDDM-IMD uses additional types of file processing. For more information, see the *GDDM Interactive Map Definition* book.

Partitioned data sets

Partitioned data sets are used by GDDM for:

- Image Symbol Sets (ISS), Vector Symbol Sets (VSS), by calls to GSLSS, PSLSS, PSLSSC, SSREAD, and SSWRT, and also by using the Image Symbol Editor.
- Device-dependent pictures by calls to FSSAVE, and FSSHOW.
- CGM conversion profiles.
- GDDM-IMD-generated mapgroups, as required by calls to MSPCRT, MSQADS, MSQGRP, MSQMAP, and MSREAD.
- Graphics data format (ADMGDF) files, as required by calls to GSLOAD and GSSAVE.
- 4250 printer typographical font and code page data, as required by calls to GSCPG and GSLSS.
- Computer Graphics Metafile (CGM) as required by calls to CGLOAD.

GDDM maintains symbol sets, pictures, generated mapgroups, and ADMGDF files as members of partitioned data sets. The member-names that GDDM uses are those specified in the corresponding GDDM calls as "symbol-set names", "picture-names", "group-names", and "names" subject to modifications of these names by any character-substitution rules that apply.

The use of partitioned data sets containing symbol sets, pictures, generated mapgroups, and ADMGDF files can be controlled by the ESLIB routine whose syntax is described in the *GDDM Base Application Programming Reference* book. This routine establishes the set of partitioned data sets that are to be used to store or retrieve a given type of object. The partitioned data sets used are identified to this routine by a list of file names.

The partitioned data sets allocated to the specified file names are searched in the order given to try to find an object. An object is stored only using the first file name of the list, even though it may have been retrieved from another one. If no file name list is provided, only the default file name is used for retrieving and storing GDDM objects.

GDDM ensures the integrity of partitioned data sets as they are written to.

The Interactive Chart Utility (part of GDDM-PGF) includes a directory function that supports list, delete, and copy operations on GDDM objects such as symbol sets, pictures, generated mapgroups, and ADMGDF files.

Sequential data sets

Sequential data sets are used by GDDM for:

- External Defaults File as part of initialization processing.
- Object modules as the result of requests from the Image Symbol Editor.

  Within a single invocation of the Image Symbol Editor, object modules are written consecutively to the selected sequential output destination. Each object module generated in this manner contains a control section (CSECT) with the name as specified by the editor, and is in a form suitable for link-editing with an application program for subsequent reference (typically, by the GSDSS or PSDSS calls). The TSO LINK command can be used to call the OS Linkage Editor for this purpose.
- Intermediate sequential data sets used in the processing of calls to DSOPEN, DSCLS, FSOPEN, and FSCLS for queued printer output. The temporary data sets created are read by the TSO Print Utility, and after output to the printer is completed, the data sets are purged.
- Output destined for a System Printer device as the result of calls to DSOPEN and DSCLS.
- High-resolution image files created as the result of calls to DSOPEN and DSCLS for family-4 devices.
- Trace records resulting from the FSTRCE function in GDDM. For a description of the use of the GDDM trace function, see the *GDDM Diagnosis* book.
- Computer Graphics Metafile (CGM) as required by calls to CGSAVE and CGLOAD.

Direct access data sets

Direct access data sets are used by GDDM for the Master Print Queue data set, used by GDDM to control requests for queued printer output made by calls to DSOPEN, FSOPEN, DSCLS, and FSCLS. GDDM ensures the integrity of the Master Print Queue, because it is written to by multiple TSO users and by the GDDM TSO Print Utility. GDDM ensures that at any one time, no more than one instance of GDDM has the Master Print Queue available for input/output processing.

File-name usage

GDDM uses **file names** to refer to all the partitioned data sets and sequential destinations, with the exception of:

- The Master Print Queue and intermediate sequential data sets that are used in the processing of queued printer output.
- (Optionally, in the absence of appropriate file names): High-resolution image files used in the processing of family-4 devices.

The file names used are as defined in Table 15 on page 543. They can be changed, if required, after installation, by specifying new values in GDDM's external defaults, as described in the *GDDM System Customization and Administration* book.

The user should ensure that the required file names are allocated to suitable data sets or destinations before GDDM is called. The data sets or destinations should have Data Control Block (DCB) characteristics as shown in Table 15 on page 543. The DCB characteristics for the data sets that contain GDDM-IMD's generated application data structures (file name ADMGNADS) and export files (file name ADMIFMT) are given in the *GDDM Interactive Map Definition* book.

If necessary, GDDM supplies default DCB characteristics when output data sets are first opened.

Required file names can be allocated to the selected data sets or destinations using the TSO ALLOCATE command. Or, the file names can be allocated by DD statements in the user's TSO logon procedure, or by dynamic allocation routines in the application program.

GDDM uses dynamic allocation to refer to the Master Print Queue and associated intermediate sequential data sets. The data-set names used include a qualifier that is defined in the current GDDM external defaults. This can be changed, if required, after installation, as described in the *GDDM System Customization and Administration* book. Or, the file name ADMPRNTQ can be used to identify a Master Print Queue data set other than that defined by the current GDDM external defaults.

The intermediate sequential data sets are allocated with a space allocation that is defined in the TSOS99S option in the current GDDM external defaults. The default allocation is equivalent to SPACE=(13030,(57,57)). If required, this can be changed after installation, as described in the table of external defaults in the *GDDM Base Application Programming Reference* book.

Dynamic allocation is also used if a print request has been specified to go directly to JES – by means of the PRINTDST processing option; see the description of GDDM print utilities in the *GDDM Base Application Programming Reference* book.

GDDM also uses dynamic allocation to refer to high-resolution image files (for family-4 output) and CGM files, unless suitable file names were previously allocated.

Table 15 (Page 1 of 2). GDDM data-set characteristics for TSO

Type of Data	GDDM default file name	Data set type	DCB characteristics		
			Record format (RECFM)	Record length (LRECL)	Block size (BLKSIZE)
Symbol sets	ADMSYMBL	Partitioned	FB	400	400*n
Pictures	ADMSAVE	Partitioned	FB	400	400*n
Generated mapgroups	ADMGGMAP	Partitioned	FB	400	400*n
GDF files	ADMGDF	Partitioned	FB	400	400*n
GIF output	-	Sequential	FB	80	80*n
4250 fonts (Note 3)	FONT4250	Partitioned	VB	2052 (includes RDW)	≥ LRECL+4
4250 code pages (Note 3)	FONT4250	Partitioned	VB	2052 (includes RDW)	≥ LRECL+4
Object modules	ADMDECK	Sequential data sets or SYSOUT classes	FB	80	80*n
System Printer Output	ADMLIST	Sequential data sets or SYSOUT classes	VBA	≥142 (Notes 1 and 2)	≥ LRECL+4
AFPDS and CDPF (4250) data stream	ADMCOLn or ADMIMAGE (optional)	Sequential data sets	VBM	2004 (for CDPF) 8202 (for AFPDS) (excludes RDW)	≥ LRECL+4
PostScript output	PS	Sequential data sets	VB	1028	≥ LRECL+4
Master print queue	ADMPRNTQ (optional)	Direct access data set	(Data set attributes provided when data set is allocated dynamically by GDDM)		
Queued printer files	(Assigned by GDDM)	Sequential data sets	FBM	80	3200
GL plotter files	(none)	Sequential or partitioned	FB	≤8000	LRECL*n
			VB		≥ LRECL+4
Trace records	ADMTRACE	Sequential data sets or SYSOUT classes	VBA	≥ 125 (includes RDW)	≥ LRECL+4
External default files	ADMDEFS	Sequential data sets	FB	≤ 256	LRECL*n
			VB		≥ LRECL+4
Image files	ADMIMG	Partitioned	FB	400	400*n
Image projection files	ADMPROJ	Partitioned	FB	400	400*n
CDPDS	LISTCDP	Sequential data sets	VB	≤ 8200	≥ LRECL+4

Table 15 (Page 2 of 2). GDDM data-set characteristics for TSO

Type of Data	GDDM default file name	Data set type	DCB characteristics		
			Record format (RECFM)	Record length (LRECL)	Block size (BLKSIZE)
CGM files (input)	(none)	Sequential or Partitioned	FB	≤8000	LRECL★n
			VB		≥ LRECL+4
CGM files (output)	(none)	Sequential	FB	400	400★n
CGM Profiles	ADMCGM	Partitioned	FB	≤256	LRECL★n
			VB		≥ LRECL+4

Notes:

1. The logical record length specified for files allocated for System Printer Output should be sufficient to contain the 4-byte Record Descriptor Word (RDW), the ASA control character, any Translation Reference Character (TRC) for 3800 devices, and the maximum number of columns for the type of System Printer selected by the application. The value 142 is adequate for any of the System Printer device characteristic tokens distributed with GDDM.
2. The output for all 3800 devices should contain table reference characters (TRCs). Consequently, the parameter DCB=OPTCD=J must be included in the output JCL. Additional parameters such as CHARS, FLASH, or FORMS may be required. For more information, see the *OS/VS2 MVS JCL* manual.
3. 4250 printer fonts and code pages are referenced by GDDM and are supplied as part of the 4250 typographical fonts licensed programs (program numbers 5771-AAA through 5771-AAW, and 5771-ACx, where x varies).

In TSO foreground operation, GDDM allows the unit specification for dynamically allocated data sets to be defaulted from the TSO user attribute data set (UADS).

In TSO Batch or MVS Batch, GDDM uses a unit specification taken from the TSOS99U option in the current GDDM external defaults. The default specification is “SYSDA”. If required, this can be changed after installation, as described in the table of external defaults in the *GDDM Base Application Programming Reference* book.

Display terminal processing

By default, the CLEAR, PA1, and PA2 keys are processed separately from other terminal input. The effects of these keys are:

CLEAR clears the screen (no other action)
PA1 raises a TSO attention interrupt
PA2 raise a GDDM “reshow” condition.

The TSO CLEAR/PA1 protocol option of the DSOPEN function can be used to suppress this separate processing of the PA1 and CLEAR keys. The TSO Reshow protocol option of the DSOPEN function can be used to specify that a key other than PA2 should act as a “reshow” key. The use of these DSOPEN options is described in the *GDDM Base Application Programming Reference* book.

The processing of these key functions is described in more detail below. Note that, because of this special processing, these key functions cannot be returned as terminal input by the ASREAD, FSSHOR, or MSREAD call, unless the key processing was modified by use of the DSOPEN protocol options.

Using the CLEAR key in full-screen mode

By default, terminal input using the CLEAR key is prevented by full-screen-mode protocols from being returned to GDDM and the application program. If the terminal user presses the CLEAR key, the screen is cleared, but no other operations occur. Specifically, GDDM may still wait to read input from the terminal, as a result of a call to ASREAD, FSSHOR, or MSREAD. Subsequently, terminal input by the user may conflict in format with that expected by GDDM; in this case, on return to the application program, an ASREAD or MSREAD operation issues this error message:

```
ADM0270 E SCREEN FORMAT ERROR
```

If this error message is issued, GDDM ensures that the screen buffer contents are subsequently restored.

The TSO CLEAR/PA1 protocol option of the DSOPEN function can be used to suppress this special processing of the CLEAR key. See the description of processing option 2000 in the *GDDM Base Application Programming Reference* book.

Entering attention interrupts in full-screen mode

By default, PA1 may be used, while GDDM is operating the terminal in full-screen mode, to cause a TSO attention interrupt. Unless the application program has established a special attention-processing function by means of the TSO STAX macro, using PA1 suspends the operation of both the application program and GDDM, and causes control to be passed to the terminal user, with the terminal in READY mode.

At this point, normal TSO protocols allow the terminal user to take the following alternative actions concerning the application program and GDDM:

- Abandon, by entering a new command to be executed
- Resume at the point of interruption, by using the ENTER key.

In the latter case, if GDDM had been interrupted while waiting for terminal input (as the result of a call to ASREAD, FSSHOR, or MSREAD), the ASREAD, FSSHOR, or MSREAD operation is completed without reading any input. On return to the application program, this error message is displayed:

```
ADM0405 E ATTENTION INTERRUPT
```

GDDM ensures that the screen buffer contents are subsequently restored.

If the application program has established a special attention-processing function by means of the TSO STAX macro, using PA1 clears the screen and displays an attention indicator, but does not force a paging condition or otherwise indicate to GDDM that the screen buffer contents were cleared. In these circumstances, the application program should subsequently issue an FSREST(1) call to cause the display buffer contents to be restored.

The TSO CLEAR/PA1 protocol option of the DSOPEN function can be used to suppress this special processing of the PA1 key.

If the terminal keyboard has a PA3 key, the default action when pressing it is to activate user control. If user control is not available, or if a key other than PA3 has

been designated for activating user control, then PA3 causes the screen to be refreshed. PA3 is never passed to the application.

Reshow key processing in full-screen mode

Under TSO, GDDM operates an IBM 3270 series display in what is known as "full-screen mode". In this mode, if the terminal is to receive a non-full-screen message, such as an error message, or a message from another TSO user, the display screen is cleared, the alarm is sounded (if applicable), and the message is displayed.

If several such messages occur consecutively, the screen is cleared once, the alarm is sounded, and the messages are displayed in sequence. When the next GDDM full-screen transmission is received, a paging condition (indicated by three asterisks, ***, at the current line) is forced.

Pressing the ENTER key at this point queues a request to GDDM to completely retransmit the display buffer contents to the terminal (this is equivalent to the call FSREST(1)). Note that GDDM receives this reshow request only if it is (or when it is next) testing for input as a result of a call to ASREAD, FSSHOR, FSSHOW, GSREAD, MSREAD, or FSFRCE. TSO protocols are such that more partial GDDM transmissions may occur before GDDM starts retransmission of the contents of the buffers.

Using the reshow key (by default, PA2) during normal full-screen processing simulates the above conditions and causes GDDM to retransmit the contents of the buffers.

The TSO Reshow protocol option of the DSOPEN function can be used to define a key other than PA2 to act as the reshow key.

Device errors in full-screen mode

Under TSO in full-screen mode, non-full-screen output to the terminal can cause some full-screen transmissions to be "discarded" or wrongly interpreted. In some circumstances, this can cause device errors (displayed in the Operator Information Area of the terminal as "X PROGnnn").

After non-full-screen output has been received at the terminal, it is possible for more partial GDDM transmissions to occur before GDDM is able to begin retransmission of the screen contents; see "Reshow key processing in full-screen mode."

In some circumstances, such partial GDDM transmissions may no longer be valid, and may cause device errors; for example:

- A partial transmission may contain a reference to a PS set. The PS set may not have been initialized because:
  - The particular PS set has not been used since the device was powered on, and
  - The GDDM transmission initializing the PS set was discarded by TSO in favor of a non-full-screen message.
- A partial transmission may assume the existence of a specific partition state on a 3290. The partition state may not exist because the GDDM transmission

creating the partition state was followed by non-full-screen output that cleared the screen and thus destroyed the partition state.

If such device errors occur ("X PROGnnn" displayed in the terminal Operator Information Area), the terminal user should press the ENTER key to acknowledge the transmission. More partial transmissions (and more device errors) may occur until GDDM receives the reshow request, at which time GDDM automatically reconstructs the entire screen contents.

Line-by-line input in full-screen mode

In full-screen mode, TSO does not update line counts for any non-full-screen input entered at the terminal. This may result in such input being obliterated by subsequent non-full-screen output to the terminal.

Usually, this does not concern an application program using GDDM, because the program expects to use GDDM to read input from the terminal in full-screen mode. Also, GDDM sets full-screen mode off when invoked for termination by means of the FSTERM call.

However, if an application program ends without a call to FSTERM (as the result of an ABEND or other error), it is possible for the terminal user subsequently to be prompted to enter line-by-line input with full-screen mode still enabled for that terminal. In this situation, the terminal user may be able to prevent obliteration of the line-by-line input by using PA1. This raises a TSO attention interrupt, and turns off full-screen mode.

NOEDIT mode under TSO

Under TSO, GDDM uses NOEDIT mode to operate a "queriable" IBM 3270 series terminal (that is, a terminal that supports the Read Partition (Query) Structured Field).

Usually, this would not concern an application program using GDDM, because GDDM maintains this mode only when reading from a terminal. However, if GDDM or the application program is abnormally terminated, it is possible for the terminal user subsequently to be prompted to enter line-by-line input with the NOEDIT mode still enabled for that terminal.

In this situation, the user may find that line-by-line input cannot be correctly interpreted, and may receive one of these messages:

```
IKJ56601I COMMAND SYSTEM RESTARTING DUE TO CRITICAL ERROR
IKJ56600I UNRECOVERABLE COMMAND SYSTEM ERROR
```

To recover from this situation, and to prevent the TSO logon session from being terminated, the terminal user must press PA1; this causes a TSO attention interrupt and turns off the NOEDIT mode.

Mixing GDDM I/O with nonGDDM I/O

If your application mixes GDDM terminal I/O with nonGDDM terminal I/O, you must issue an FSREST call before the first call to ASREAD (or other GDDM I/O call) that follows any nonGDDM I/O. This ensures that GDDM restores the screen contents correctly.

Using APL terminals

Under TSO, device information provided by the subsystem does not distinguish between an IBM 3277 Model 2 display terminal and an IBM 3278 or 3279 Model 2 display terminal, unless the latter is defined to be “queriable”; that is, is defined to support the Read Partition (Query) Structured Field by the 3274 Controller Configuration Support C and the Extended Character Set Adapter (feature number 3610).

By default, GDDM resolves this ambiguity by assuming that the device is an IBM 3278. If the device is actually a nonqueriable IBM 3278 or 3279 Model 2 with an APL Feature, and if the APL character set is to be referred to by an application, the GDDM default assumption must be overridden to ensure correct operation of the device. The GDDM default can be overridden in any of these ways:

- The application can specify an explicit device token (for example, ADMK782A) on a DSOPEN call to initialize the device.
- The TSOAPLF option in GDDM’s current external defaults can be modified to cause GDDM to assume by default that a nonqueriable Model 2 display terminal is an IBM 3278 or 3279. This option can be specified in one of the following places:
  - In an External Defaults Module
  - In an External Defaults File that was allocated to ddname ADMDEFS or
  - On a SPINIT, ESSUDS, or ESEUDS call in an application program.

  For details, see the *GDDM Base Application Programming Reference* book.

Also, under TSO, device information provided by the subsystem does not indicate whether a 3277 Model 2 display or a nonqueriable 3278 or 3279 display actually has the appropriate APL feature.

By default, GDDM assumes that such a device has the APL feature, and it selects an appropriate set of translation tables. (For more information, see “Country-extended code pages” on page 248 and the description of ASTYPE in the *GDDM Base Application Programming Reference* book.) If the device does not have the APL feature, the use of character code points that correspond to APL characters may result in incorrect output at the device.

The GDDM default can be overridden in either of the following ways. The application program can:

- Specify an explicit device token (for example, ADMK7720) in a DSOPEN call to initialize the device (see the *GDDM Base Application Programming Reference* book) or by means of nickname facilities (see “Coding a partial device definition for end users to change with nicknames” on page 374).
- Use the ASTYPE call to specify the appropriate set of translation tables, as follows:

Device type	Translation type number
3277	3277
3277-APL	32771
3278, 3279	3279
3278-APL, 3279-APL	32791

For a description of the operation of alphanumeric translation tables, see the *GDDM System Customization and Administration* book.

Using GDDM under TSO batch

TSO Extensions (TSO/E) is a licensed program (program number 5665-285) that provides a TSO Batch environment in which TSO commands and command procedures can be run in the background. GDDM can be used in this environment, in normal MVS Batch, subject to the following considerations.

- TSO Batch applications must be link-edited using the information under "Link-editing a GDDM application program" on page 539.
- GDDM processes any External Defaults File allocated by means of a DD statement; the default ddname is ADMDEFS.
- The GDDM default error exit reports errors. These messages usually appear on the JOB LOG output.
- GDDM dynamically allocates queued printer files or high-resolution image files for family-4 devices using a unit specification that is defined in the TSOS99U option in the current GDDM external defaults. The default unit specification is SYSDA. If required, this can be changed, as described in the *GDDM Base Application Programming Reference* book.
- The GDDM-supplied interactive utilities necessarily use the default primary device (the "TSO terminal"), unless called for noninteractive processing. Therefore, these utilities cannot be run interactively in TSO batch.
- The default primary device (the simulated TSO terminal) is not suitable for GDDM full-screen operations. GDDM diagnoses any attempt to use this device.

  Therefore, an application must include an explicit DSOPEN to identify a nondefault primary device (for example, a dummy device or non-family-1 device).
- The GDDM default error exit reports errors. User PROFILE options can be used to cause the messages to appear as part of the session output file (SYSTSPRT). The TSO command to request that messages appear on the session output file is:

  ```
  PROFILE WTPMSG
  ```

  and this should be included in the session input file (SYSTSIN) before GDDM is used.
- Unless the application is running as part of a RACF job with USERID, no default data-set-name prefix or user ID is defined. A default data-set-name prefix may be required by GDDM for dynamic allocation of queued printer files or high-resolution image files (for family-4 devices). The TSO command to establish a default data-set-name prefix is:

  ```
  PROFILE PREFIX(dsname-prefix)
  ```

  and this should be included in the session input file (SYSTSIN) before GDDM is used.
- GDDM uses the user ID only for annotation purposes (in print files and trace files). In the absence of a user ID, GDDM uses the JOB name.

Using GDDM under MVS batch

These items are specific to processing under MVS Batch:

- MVS Batch applications must be link-edited using the information under "Link-editing a GDDM application program" on page 539.
- GDDM processes any External Defaults File allocated by means of a DD statement; the default ddname is ADMDEFS.
- The GDDM default error exit reports errors. These messages usually appear on the JOB LOG output.
- GDDM dynamically allocates queued printer files or high-resolution image files for family-4 devices using a unit specification that is defined in the TSOS99U option in the current GDDM external defaults. The default unit specification is SYSDA. If required, this can be changed, as described in the *GDDM Base Application Programming Reference* book.
- The GDDM-supplied interactive utilities necessarily use the default primary device (the "TSO terminal"), unless called for noninteractive processing. Therefore, these utilities cannot be run interactively MVS Batch.
- The default primary device (the simulated TSO terminal) is not available for GDDM full-screen operations. GDDM diagnoses any attempt to use this device.

  Therefore, an application should include an explicit DSOPEN to identify a nondefault primary device (for example, a dummy device or non-family-1 device).
- The default data-set-name prefixes or user IDs that are given under TSO are not applied. GDDM does not apply such a prefix for dynamic allocation of queued printer files or high-resolution image files for family-4 devices. Queued printer files are allocated with names of the form:

  ```
  ADMPRINT.REQUEST.#nnnnn
  ```

  where the string ADMPRINT is as provided in GDDM's defaults. The name ADMPRINT can be changed by specifying a new value in the TSOPRNT option in GDDM's external defaults. For more information, see the table of external defaults in the *GDDM Base Application Programming Reference* book.
- GDDM uses the JOB name for annotation purposes in print and trace files.

Programming under TSO on extensions of MVS

This section describes the special programming considerations for 31-bit mode GDDM applications, and provides general information on GDDM code and application programs that can run under TSO on the MVS/ESA operating system.

GDDM code above 16 megabytes

Under suitable subsystems and operating systems, the main body of GDDM code can reside above 16MB.

Under TSO, some GDDM routines are located below 16MB.

Application code above 16 megabytes

Under suitable releases of TSO, GDDM applications can reside above 16MB.

AMODE(31) applications and application parameters above 16 megabytes

Under TSO, applications can run in 31-bit mode and can pass to GDDM parameters that are located above 16MB.

If GDDM is called in 31-bit mode, it assumes that any parameter addresses that are passed represent 31-bit addresses.

Application programming considerations

Under MVS/ESA, a GDDM application program may have any valid AMODE attribute, and may call GDDM in any mode (24-bit or 31-bit) consistent with its location. In fact, it is possible (though not recommended) for an application program to call GDDM in both 24-bit and 31-bit modes in the same session.

User exits

A number of other user exits can be defined for programs using the SPI. These exits and the special consideration for their use on MVS/ESA are described in the *GDDM Base Application Programming Reference* book.

Example: JCL for link-editing GDDM applications under TSO

```
//************************ TSO ****************************************
//*
//* Example of JCL to link-edit a GDDM/TSO
//* sample program or user-written application.
//*
//* xxxxxxxx is the name under which the program load module is
//* generated.
//*
//*********************************************************************
//*
//jobname    JOB  accounting info,..........
//*
//* Link-edit step
//*
//* Include INCLIB to reference library containing GDDM interface
//* modules, as shown.
//*
//* In the specified INCLUDE statement,
//* leave   ADMASNT unchanged  if using the nonreentrant interface
//* replace ADMASNT by ADMASRT if using the reentrant interface
//*            or   by ADMASPT if using the system programmer interface
//*
//LKED       EXEC PGM=IEWL,PARM='XREF,LIST',REGION=768K
//SYSPRINT   DD   SYSOUT=A
//SYSLIB     DD   DSN=as-required-by-application,DISP=SHR
//INCLIB     DD   DSN=GDDM.SADMMOD,DISP=SHR
//SYSLMOD    DD   DSN=user-load-module-dataset,DISP=SHR
//SYSUT1     DD   UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLIN     DD   *
      . . . .
      . . . .
 Program object deck here.
      . . . .
      . . . .
 INCLUDE INCLIB(ADMASNT)
 NAME    xxxxxxxx(R)
/*
```

Appendix D. Programming with GDDM under IMS

This appendix describes the use of GDDM under the IMS operating system. It covers the following topics:

- Restrictions on the use of GDDM under IMS
- Application program structure
- Link-editing a GDDM application
- Using the system programmer interface with dynamic load
- PSBs for GDDM applications
- Data sets and file processing
- The IMS default error exit
- GDDM and MFS
- GDDM DL/I interface
- IMS considerations for GDDM utilities
- GDDM object import/export utility
- Examples of JCL.

The use of the IMS version of the GDDM print utility is described in the *GDDM System Customization and Administration* book.

Application programs for IMS should carefully follow the instruction given under "The structure of GDDM application programs for use on IMS" on page 554. Careful note of the restrictions should also be taken. The IMS samples in Appendix A, "GDDM sample programs" on page 519 can be used as a model for application programs.

The description in this appendix assumes a working knowledge of IMS.

Restrictions on the use of GDDM under IMS

The main restrictions on the use of GDDM in an IMS environment are:

- Output on IPDS printers is not supported.
- Picture interchange format (PIF) files are not supported.
- GDDM-IMD is not supported.
- GDDM-PCLK is not supported.
- GDDM-OS/2 Link is not supported.
- The 5080 and 6090 Graphics Systems are not supported.
- Interchange with Computer Graphics Metafile (CGM) is not supported.
- The creation of plotter files containing IBM GL data stream is not supported.
- GDDM supports only system network architecture (SNA) connection for 3179-G, 3192-G, and 3472-G display stations, 3270-PC/G, /GX, and /AT workstations, and 5550-family multistations.
- For 327x displays the amount of data that can be created by GDDM and successfully transmitted by IMS depends on the line protocol and access method used to send this data to the terminal.

  For terminals defined as SLUTYPE2, or remote 3270 devices specified with data transparency, OPTIONS=XPAR, there are no restrictions.

© Copyright IBM Corp. 1982, 1996

For all other 3270 displays the amount of data that may be created and sent by GDDM in one message is controlled by the OUTBUF parameter specified during system definition.

For very complex pictures the length of the data streams generated by GDDM may exceed this maximum value. In such cases, the output message is rejected by IMS and an IMS error message is displayed at the terminal. If this occurs and the device token being used specifies COMPRES=NO, one way of reducing the length of the data stream is to use a different device token (one that has COMPRES=YES) that allows data-stream compression (assuming that the 3274 control unit is configured for PS compression). For more information, see the list of device tokens in the *GDDM Base Application Programming Reference* book.

- For 3270-family terminals and printers output may only be sent to logical terminals that are defined in the GDDM System Definition database. This contains information that describes the physical characteristics of the device.

  The information in the database is located using the LTERM name of a message queue as a key rather than the physical terminal name, because only that piece of information is available to the application and thus GDDM. To prevent transmission errors the device to which the LTERM is assigned must have the characteristics identified in the database. Reassignment of LTERMS must be reflected by changes to the database.

- GDDM cannot be used to process input from the terminals. The use of message queues and the scheduling algorithms of an IMS system are unsuited to the direct interaction allowed on other subsystems.

  Information on the interaction of GDDM and the message format service (MFS) and a description of how input from a display formatted by GDDM should be processed, is given in “GDDM and the Message Format Service” on page 560.

- FSSAVE files generated under IMS cannot be used under another subsystem, such as TSO, nor may such files created under other subsystems be sent to a device attached to IMS using the FSSHOW functions.

- For the interactive utilities only, the use of PF key 12 allocated by IMS to the COPY function should be avoided. If the keyboard has only 12 PF keys, the IMS system definition for the terminal should specify NOCOPY.

- Plotters attached to 3179-G, 3192-G, or 3472-G display stations, or to 3270-PC/G, /GX, or /AT workstations are not supported under IMS.

- The WINDOW processing option and operator window functions are not supported under IMS.

- ICU flat-file data import is not supported under IMS.

The structure of GDDM application programs for use on IMS

The following list contains the steps that an IMS transaction program might make when using GDDM.

1. Issue a GU call to the I/O program communication block (PCB) to acquire the first segment of the input message.
2. Issue FSINIT, or any of its aliases, to enable GDDM processing.

3. Optionally issue an FSEXIT call to nominate a user-provided error exit to replace the default exit provided with GDDM, or to raise the threshold of errors below which errors are not reported.
4. Issue one or more ESPCB calls to identify to GDDM the PCBs that it may use.
5. Issue one or more ESLIB calls to show which databases are to be searched when retrieving and storing GDDM data.
6. If the I/O PCB has not been identified by an ESPCB call above, or if output is to go to a destination other than that of the I/O PCB, issue DSOPEN calls to define to GDDM the possible output destinations.

   If the PCB to be used by GDDM is modifiable, the destination of the PCB must be set using the CHNG call before the DSOPEN call is issued.

   This step is not needed if output is to go to the source of the input message and the I/O PCB has been identified to GDDM because this is the default destination and PCB used by GDDM.
7. Process the input message using GN calls to acquire subsequent message input. Generate output messages using the GDDM subroutines to describe any field-formatted or graphics output. Use the DSUSE statement to select the output destination if devices have been explicitly defined by DSOPEN.
8. Issue DSCLS statements for each device opened using DSOPEN.
9. Issue the FSTERM call to end GDDM processing.
10. Repeat from step 1 to process any more input messages.

This arrangement of an application program ensures that GDDM is inactive across a GU call that may reset certain information used by GDDM. Its drawback is the repeated initialization and termination of GDDM. An alternative structure that avoids this overhead is shown below. Care should be taken to ensure that all devices are closed across the GU call.

1. Issue FSINIT, or any of its aliases, to enable GDDM processing.
2. Optionally issue an FSEXIT call to nominate a user-provided error exit to replace the default exit provided with GDDM, or to raise the threshold of errors below which errors are not reported.
3. Issue one or more ESPCB calls to identify to GDDM the PCBs that it may use.
4. Issue one or more ESLIB calls to show which databases are to be searched when retrieving and storing GDDM data.
5. Issue a GU call to the I/O PCB to acquire the first segment of the input message.
6. If the I/O PCB has not been identified by an ESPCB call above, or if output is to go to a destination other than that of the I/O PCB, issue DSOPEN calls to define to GDDM the possible output destinations.

   If the PCB to be used by GDDM is modifiable, the destination of the PCB must be set using the CHNG call before the DSOPEN call is issued.

   This step is not needed if output is to go to the source of the input message and the I/O PCB has been identified to GDDM because this is the default destination and PCB used by GDDM.
7. Process the input message using GN calls to acquire subsequent message input. Generate output messages using the GDDM subroutines to describe any

field-formatted or graphics output. Use the DSUSE statement to select the output destination if devices have been explicitly defined by DSOPEN.

8. Issue DSCLS statements for each device opened using DSOPEN.

   If the default destination was used, GDDM automatically opens a device with an identifier of 0. This should be closed using a statement of the form

   ```
   CALL DSCLS(0,1)
   ```

9. Repeat from step 5 on page 555 to process any more input messages.
10. Issue the FSTERM call to end GDDM processing when all input messages have been processed.

Programming under IMS on extensions of MVS

This section describes the special programming considerations for 31-bit mode GDDM applications, and provides general information on GDDM code and application programs that can run under IMS on the MVS/ESA operating system.

GDDM code above 16 megabytes

Under suitable subsystems and operating systems, the main body of GDDM code can reside above 16MB.

Under IMS, some GDDM routines are located below 16MB.

Application code above 16 megabytes

Under IMS, GDDM applications cannot reside above 16MB.

AMODE(31) applications and application parameters above 16 megabytes

Under IMS, applications can run in 31-bit mode and pass to GDDM parameters that are located above 16 megabytes.

If GDDM is called in 31-bit mode, it assumes that any parameter addresses that are passed represent 31-bit addresses.

Application programming considerations

Under MVS/ESA, a GDDM application program may have any valid AMODE attribute, and may call GDDM in any mode (24-bit or 31-bit) consistent with its location. In fact, it is possible (though not recommended) for an application program to call GDDM in both 24-bit and 31-bit modes in the same session.

User exits

A number of other user exits can be defined for programs using the SPI. These exits and the special consideration for their use on MVS/ESA are described in the *GDDM Base Application Programming Reference* book.

Link-editing a GDDM application program

Examples of the JCL that can be used to compile and link-edit application programs written in PL/I or COBOL are listed in “Example: JCL to compile and link PL/I GDDM applications under IMS” on page 565 and “Example: JCL to compile and link COBOL GDDM applications under IMS” on page 566.

Unless an application program uses dynamic load facilities to access GDDM through the system programmer interface (see below), a GDDM application program must be link-edited with the appropriate GDDM interface module as well as the DL/I interface module. The interface module used depends on the type of GDDM interface used and the language of the application program, or, to be precise, of the program specification block (PSB) for the transaction.

The module to be used may be explicitly controlled by linkage editor control statements, or one of the alternative versions of the initialization entry point can be used. The latter causes the correct GDDM interface modules to be loaded by the automatic library call capability of the linkage editor.

There are four alternative initialization calls for GDDM in an IMS environment. They allow for a choice of nonreentrant and reentrant interface and nonPL/I and PL/I PSBs. The names of the initialization calls are as follows:

Interface	Non-PL/I PSB	PL/I PSB
Nonreentrant	FSINNI	FSINNPI
Reentrant	FSINRI	FSINRPI

If direct control of the link-edit process is chosen, the initialization call should be coded using the FSINIT (or SPINIT) entry point, and the following modules explicitly included by the link-edit process:

Interface	Non-PL/I PSB	PL/I PSB
Nonreentrant	ADMASNI	ADMASNJ
Reentrant	ADMASRI	ADMASRJ
System Programmer	ADMASPI	ADMASPJ

Using the system programmer interface with dynamic load

If an application program uses only the system programmer interface (SPI), all invocations of GDDM are through the entry point ADMASP. This entry point can be resolved by link-editing the application program with one of the GDDM interface modules, ADMASPI or ADMASPJ, as described above.

However, the application program can avoid these linkage-edit considerations by using system facilities (the OS LOAD function) to dynamically load a GDDM interface module (ADMASPLI for non-PL/I PSBs or ADMASPLJ for PL/I PSBs). The main entry points for these modules are defined both with their load module names and with the name ADMASP.

Note: If an installation uses the OS LOAD function is used to dynamically load GDDM, applications that use GDDM's system programmer interface cannot use the ADMUFO to bypass parameter checking.

Program specification blocks for GDDM applications

The PSB for a GDDM application must include the PCBs required by GDDM. These are:

- One TP PCB for each concurrently active device (for example, for which a DSOPEN call was issued).

  For family-1 and family-3 (3270-family and system printer) devices, the LTERM quoted in the PCB statement must be that of the terminal to which the output is to be sent. For family-2 devices, the NAME parameter should specify the transaction code assigned to the GDDM print utility.

  If the NAME or LTERM parameter is not supplied on the PCB statement, the PCB should be defined as modifiable and the application program should issue a CHNG call to set the destination before defining the PCB to GDDM.

- A DB PCB for the system definition database if GDDM output is to be generated. A PROCOPT of G should be specified because no normal GDDM operation can alter information in this database. For multiple IMS sessions, however, a procopt of GO is recommended so that segment locking is avoided.

  A sample PCB statement for such a database is:

  ```
  PCB    TYPE=DB,NAME=ADMSYSDF,PROCOPT=G,KEYLEN=8
  SENSEG NAME=ADMSDSGM,PARENT=0
  ```

  Ensure that the names used in the above sample are not altered during the initialization process. If they are changed, corresponding changes must be made in the IMSSDBD and IMSSEGS options in GDDM's external defaults, described in the *GDDM Base Application Programming Reference* book.

- A DB PCB for each object database required.

  A sample PCB statement for such a database is:

  ```
  PCB    TYPE=DB,NAME=ADMOBJ1,PROCOPT=G,KEYLEN=20
  SENSEG NAME=ADMOBROO,PARENT=0
  SENSEG NAME=ADMOBDEP,PARENT=ADMOBROO
  ```

  A PROCOPT of A should be specified if the program is to alter information in the database using GDDM calls. Note the restriction that information is written only to the first of the databases quoted in the ESLIB parameter list for any given type of object.

  It is possible to vary the DBD and segment names from those quoted above during IMS system generation. If they are changed, corresponding changes must be made in the OBJFILE and IMSSEGS options in GDDM's external defaults, as described in the *GDDM Base Application Programming Reference* book.

  However, if only the data-base name is to be altered, the ESLIB statement can be used to notify GDDM of the data-base name rather than altering the external defaults. The name in the external defaults is only used to find the database to search for objects if no ESLIB statement is coded.

Table 16. GDDM data-set characteristics for IMS

Type of Data	GDDM default filename	Data set type	DCB characteristics		
			Record format (RECFM)	Record length (LRECL)	Block size (BLKSIZE)
Symbol sets	ADMTRACE	Sequential data sets or SYSOUT classes	VBA	≥125	≥LRECL + 4

An ESPCB call should be coded in the application for each PCB to be used by GDDM.

Data sets and file processing

When running under IMS, GDDM uses two types of file processing:

- QSAM (Queued Sequential Access Method) is used to write data to sequential output destinations when certain trace functions are requested using the FSTRCE call. For information on the use of FSTRCE, see the *GDDM Diagnosis* book.
- DL/I is used to read and write information into the two types of DL/I database used by GDDM.

In the first type, GDDM refers to the file using a ddname. The default value of this name is taken from the IMSTRCE option in GDDM's external defaults. (For more information, see the table of external defaults for IMS in the *GDDM Base Application Programming Reference* book.) If output is to be created from this file, the dependent region JCL must be modified to include a DD statement for it. The data set type and DCB characteristics should be as shown in Table 16.

The Interactive Chart Utility (part of GDDM-PGF) includes a directory function that supports list, delete, and copy operations on GDDM DL/I objects such as symbol sets and pictures.

Specifying the default error exit under IMS

GDDM provides a default error exit, which is given control when GDDM detects an error in its processing. The user can control the severity level of an error that causes the exit to be taken and may also identify a user-written error exit, as described in “Example of an error exit routine, using FSEXIT” on page 139 and in the entry for FSEXIT in the *GDDM Base Application Programming Reference* book.

The default error exit provided in the IMS environment reports the error using a /BROADCAST command directed to the LTERM named in the I/O PCB. The transaction must, therefore, be authorized to issue this command. If the I/O PCB was not identified to GDDM by the ESPCB call, or the CMD call fails, the error message is issued using a “write to operator” (WTO) function. The route code and message descriptor for this WTO function are contained in GDDM's external defaults. The IMSWTOR and IMSWTOD external defaults can be changed to suit the installation. For information on how to do this, see the *GDDM Base Application Programming Reference* book.

GDDM and the Message Format Service

GDDM uses the Message Format Service (MFS) BYPASS function to send output to 3270 displays and to non-SCS printers. Output to SCS printers is sent using Basic Edit.

For displays, each message created by GDDM contains the information needed to format the screen. By default, it is sent using a Message Output Descriptor (MOD) with the name DFS.EDT (for a user application) or DFS.EDTN (for a GDDM or GDDM-PGF interactive utility). When a message using one of these MODs is detected by MFS, it does not format the information in the message but instead assumes that it contains a data stream that may be sent to the device without more processing.

Any input subsequently received from the device for a user application is not processed against a Message Input Descriptor (MID) but is instead passed to the Basic Edit process. This removes the device-dependent control information from the data stream and replaces it with blanks.

Using GDDM it is possible to create a message containing a picture and one or more input fields. When this has been displayed, the end user can enter the next transaction request from the terminal by typing into the input field and pressing the ENTER key.

The segment returned from the GU DL/I function call in the application program contains the contents of the fields modified by the end user in a single segment. There is no indication of the key (PF, ENTER, or PA) that caused the data to be sent to IMS. The fields are of variable length, separated from each other by one or more blanks.

For more information on the detailed formatting of the input data stream, see the description of the Message Format Service in the IMS reference manuals.

An installation can provide its own MOD to be used by GDDM for transmitting nonconversational messages from a user application to 3270-family devices. In this way, an installation can make special provision for processing subsequent input messages. To cause GDDM to use a MOD name other than DFS.EDT, the alternative MOD name must be specified in the IMSMODN option in GDDM's external defaults, as described in the *GDDM Base Application Programming Reference* book.

GDDM DL/I interface

The GDDM routines use the same DL/I interface as a standard application program. To do so, GDDM needs to know which of the PCBs, passed to the application when it is scheduled, are to be used by GDDM. This information is passed to GDDM by the ESPCB call. The syntax of this function is described in detail in the *GDDM Base Application Programming Reference* book.

Using this function, the application program can identify the I/O PCB, other TP PCBs, and DB PCBs. The use GDDM makes of each of these types of PCB is described in the next sections.

The following general rules apply to the sharing of PCBs between an application and GDDM:

1. GDDM uses the TP PCBs to insert the data streams that it generates to the message queues. Such a PCB is considered to be in use between the times that the GDDM device services calls DSOPEN and DSCLS are issued. These calls are described in more detail in the *GDDM Base Application Programming Reference* book. While a PCB is in use, the application program must not also insert data on the queue through the same PCB nor must it cause the data on the PCB to be enqueued by issuing a GU to the I/O PCB or any other action that causes a checkpoint.
2. If an application program tries to send output when no primary device was explicitly defined, GDDM tries to open a device to use the I/O PCB.
3. If the application needs to insert another message to the message queue, using a PCB that was used by GDDM, the first segment of the message must be inserted using the DL/I PURG function to enqueue any message created by GDDM. GDDM itself inserts the first message segment, using this function to enqueue any application output already placed on the message queue before a device is opened.

Use of message queues

GDDM uses the I/O and TP PCBs to insert output to message queues for the primary and alternate devices. These devices can be 3270-family devices, queued printer devices, or system printer devices.

The PCB used by any device depends on the way in which the device was identified using the DSOPEN function and on the type of device. The method used by GDDM to select the PCB to be used is given below.

Each message is created by inserting one or more segments. The number of segments is dependent on the complexity of the output. For system printer devices, each output segment is a print record. For the other types of device, the message is segmented at arbitrary points in the generated output. In this latter case, the maximum size of the output segment is 84 bytes for a queued printer device, and is taken from the value of the IOBFSZ option in the current GDDM external defaults for a 3270-family device.

3270-family devices

The NAME parameter on DSOPEN supplies the name of the LTERM to which output is to be sent. GDDM selects the PCB to be used by checking first the I/O PCB and then each of the TP PCBs, in the order in which they were identified by ESPCB calls, for a destination of the given LTERM. It uses the first one of these PCBs that is not already in use for another device.

If the NAME parameter is omitted, or coded as "*", GDDM tries to use only the I/O PCB.

If no PCB with a matching name is found, or if all PCBs checked are already in use, the DSOPEN function fails.

The number of messages generated by GDDM for this family of device is dependent on the type of the target terminal. If it is a display, the output created from each FSFRCE or ASREAD call is sent as an individual message. If the

terminal is a printer, all output created by the application program using the GDDM device is sent in a single message.

If the application is conversational and the I/O, or another PCB, is selected by GDDM for use with a display device, the application may only issue the FSFRCE or ASREAD call once because, in this situation, GDDM cannot issue the DL/I PURG request required to cause the message created by the first call to be enqueued.

Queued printer devices

These devices generate output that is sent to the GDDM-provided Print Utility for subsequent transmission to a real 3270-family terminal. The NAME parameter specified on DSOPEN identifies the LTERM name of the latter terminal and cannot be omitted. The output generated by GDDM directly from the application program is inserted to the first PCB in which the LTERM name is the transaction code of the GDDM print utility. The default value for this transaction name is ADMPRINT, but the installation may change this by altering the IMSPRNT option in the current GDDM external defaults, as described in the *GDDM Base Application Programming Reference* book. If no such PCB can be found, or if all such PCBs are already being used by other GDDM devices, the DSOPEN function fails.

All the output created by GDDM between DSOPEN and DSCLS for a device of this type is sent as a single IMS message.

System printer devices

The NAME parameter specified on DSOPEN should identify an LTERM to which print records, including carriage control characters, can be sent. If omitted, a default destination is assumed by GDDM. This is ADMLIST, but the installation may change the value by altering the IMSSYSP option in the current GDDM external defaults, as described in the *GDDM Base Application Programming Reference* book.

The PCB to be used is again chosen by checking first the I/O PCB, and then all TP PCBs, in the order identified by the application, for an LTERM name matching that given or assumed on the DSOPEN call. If no match is found, or if all matching PCBs are already in use, the DSOPEN function fails.

All the output created by GDDM for any one device of this type forms a single IMS message.

Use of databases

GDDM uses two types of database: one to contain the terminal characteristics information, and another to contain the “objects”, such as symbol sets, saved pictures, generated mapgroups, and ADMGDF files. The DB PCBs that are to be used must be identified to GDDM by the ESPCB call before executing any routine that might require access to the data bases.

The use of the databases containing objects is further controlled by the ESLIB call, the syntax of which is described in the *GDDM Base Application Programming Reference* book. This routine establishes the set of databases that are to be used to store or retrieve a given type of object. The data bases to be used are identified to this routine as a list of DBD names. Before issuing this call the user must have issued ESPCB calls that referred to DB PCBs for all the databases mentioned on the ESLIB call.

The databases are searched in the order given in an attempt to find an object. An object is stored only in the first database of the list, even though it may have been retrieved from another one.

The DBD name of the system definition database is taken from the value in the IMSSDBD option in the current GDDM external defaults. The external defaults also contain default DBD names for the databases to be used for each of the object types.

IMS considerations for GDDM utilities

Under IMS, the GDDM and GDDM-PGF interactive utilities are run under the control of a single transaction that emulates the environment that they expect. The transaction is a “wait for input” conversational transaction. In these notes, the transaction code for the utility is assumed to be “ADM,” but this may have been changed by the installation.

- The transaction can support only a predefined number of concurrent transactions. Any attempt to start a new session with a utility that would cause the limit to be exceeded is rejected with message ADM0772.

  The number of concurrent transactions allowed may be altered by modifying the value in the IMSUMAX option in the current GDDM external defaults. For more information, see the table of GDDM external defaults for IMS in the *GDDM Base Application Programming Reference* book.

- The transaction cannot continue conversations if, for any reason, it is rescheduled during the lifetime of a conversation. Such conversations are terminated with message ADM0774.

- A particular scheduling of the transaction usually ends when it has no record of any existing conversations. Because it is possible for a conversation to be terminated without the transaction being aware of the fact (for example, because of particular error conditions), the transaction may not be completed even though the end user has terminated the conversation. In such a case, the end user should enter the request:

  ```
  ADM EXIT
  ```

  which causes the utility to note that all conversations against the LTERM, from which the request originates, were terminated.

- To force a return to the region controller by the transaction irrespective of the current state of any active conversations, the request:

  ```
  ADM SHUTDOWN
  ```

  can be entered from an authorized terminal. By default this authorized terminal has an LTERM name of MASTER.

The keywords EXIT and SHUTDOWN, and the LTERM name of the terminal authorized to issue the latter request, are as defined in the IMSEXIT, IMSSHUT, and IMSMAST options in the current GDDM external defaults. For more information, see the table of GDDM external defaults for IMS in the *GDDM Base Application Programming Reference* book.

- If, during a session with a utility, the current screen format is destroyed (for example, by a high priority or error message), it can be restored by entering two blank characters as the next input message.

- On some terminals, IMS reserves Program Function key 12 for use as a print request key and does not pass this as a valid interrupt to the utility transaction. If the terminal has 24 rather than 12 PF keys, the use of PF key 12 can be avoided because PF 24 usually has the same function.

  If only 12 PF keys are available, the IMS system definition for a terminal should specify NOCOPY if the GDDM utilities are to be accessed from that terminal.

GDDM object import/export utility

The GDDM object import/export utility is used to transfer GDDM objects (generated mapgroups from GDDM-IMD, ADMGDF objects, symbol sets, chart formats or data, or FSSAVE objects) between partitioned data set(s), and the database in which they are kept for IMS use, or to delete them from the database.

Its purpose is to enable objects to be transferred between GDDM applications running on one IMS system, and those running on either another IMS system, or in a totally different environment (for example a TSO development system).

The operation and use of the utility are described in the *GDDM System Customization and Administration* book.

Example: JCL to compile and link PL/I GDDM applications under IMS

```
//******************** IMS PL/I ****************************************
//*
//* Example of JCL to compile, and link-edit a GDDM/IMS
//* sample program or user-written application.
//*
//* This JCL assumes the use of the IMS-supplied
//* cataloged procedure “IMSPLI”.
//*
//* The IMS/GDDM sample program or user-written application is
//* placed in IMSVS.PGMLIB.
//*
//* xxxxxxxx is the name under which the program load module is
//* generated.
//*
//*********************************************************************
//*
//jobname    JOB   accounting info,..........
//           EXEC  PROC=IMSPLI,MBR=xxxxxxxx,REGION.C=512K,
//             PARM.C='XREF,A,OBJ,NODECK,INC,OPT(TIME)'
//*
//* Compilation step
//*
//* Insert SYSLIB to reference library containing GDDM sample
//* PL/I declarations, as shown.
//*
//C.SYSLIB   DD   DSN=GDDM.SADMSAM,DISP=SHR
//C.SYSIN    DD   *
      . . . .
      . . . .
 Source deck here.
      . . . .
      . . . .
/*
//*
//* Link-edit step
//*
//* Insert INCLIB to reference library containing GDDM interface
//* modules, as shown.
//*
//* In the specified INCLUDE statement,
//* leave   ADMASNJ unchanged  if using the nonreentrant interface
//* replace ADMASNJ by ADMASRJ if using the reentrant interface
//*            or   by ADMASPJ if using the system programmer interface
//*
//L.INCLIB   DD   DSN=GDDM.SADMMOD,DISP=SHR
//L.SYSIN    DD   *
 INCLUDE INCLIB(ADMASNJ)
/*
```

Example: JCL to compile and link COBOL GDDM applications under IMS

```
//******************** IMS COBOL ************************************
//*
//* Example of JCL to compile, and link-edit a GDDM/IMS
//* sample program or user-written application.
//*
//* This JCL assumes the use of the IMS-supplied
//* cataloged procedure “IMSCOBOL”.
//*
//* The IMS/GDDM sample program or user-written application
//* is placed in IMSVS.PGMLIB.
//*
//* xxxxxxxx is the name under which the program load module is
//* generated.
//*
//*********************************************************************
//*
//jobname    JOB   accounting info,..........
//*
//           EXEC  PROC=IMSCOBOL,MBR=xxxxxxxx
//*
//* Compilation step
//*
//C.SYSIN    DD    *
      . . . .
      . . . .
 Source deck here.
      . . . .
      . . . .
/*
//*
//* Link-edit step
//*
//* Insert INCLIB to reference library containing GDDM interface
//* modules, as shown.
//*
//* In the specified INCLUDE statement,
//* leave   ADMASNI unchanged  if using the nonreentrant interface
//* replace ADMASNI by ADMASRI if using the reentrant interface
//*            or   by ADMASPI if using the system programmer interface
//*
//L.INCLIB   DD   DSN=GDDM.SADMMOD,DISP=SHR
//L.SYSIN    DD   *
 INCLUDE INCLIB(ADMASNI)
/*
```

Appendix E. Programming GDDM applications for use with CICS

This appendix describes the use of GDDM under the CICS subsystem. It contains these sections:

- Programming languages and restrictions
- CICS conversational applications
- CICS pseudoconversational applications
- Requesting transaction-independent services
- Using the GDDM nonreentrant interface
- Using the GDDM system programmer interface with dynamic load
- Using GDDM with Basic Mapping Support
- CICS GDDM default error exit
- Display terminal conventions
- CICS GDDM data sets and file processing
- Compiling and link-editing GDDM application programs

The GDDM print utility is described in the *GDDM Base Application Programming Reference* book.

A working knowledge of CICS is assumed throughout this appendix.

Programming languages and restrictions

GDDM can be used by CICS command-level (EXEC) application programs written in the PL/I, C/370, COBOL, or Assembler languages.

COBOL restriction: COBOL programs that run under CICS must not use the STOP RUN statement.

IBM GL restriction: The creation of plot files containing IBM GL datastream, is not possible under CICS.

CICS conversational applications

Applications that are initiated from a terminal and consist of a dialogue with the terminal user can be described as **conversational** applications. The logical flow of such programs can be summarized as follows:

1. Start the application
2. Perform initialization
3. Do until finish requested
   a. **Converse** with the terminal user (typically, display a panel and wait for input)
   b. Process the terminal input
4. Perform termination
5. End the application

© Copyright IBM Corp. 1982, 1996

You can see from the above summary that the program conducts a conversation with the terminal user.

GDDM provides several calls that you can use to perform the conversation:

ASREAD Output the current page and await alphanumeric input
FSSHOW Display a saved picture (device dependent)
FSSHOR Display a saved picture (device dependent) and return information on key used to terminate display
GSREAD Output the current page and await graphics input
MSREAD Output the current map and await mapped input
WSIO Control input/output on a windowed device

A conversational program of the type summarized above can be run on the CICS subsystem, where it is known as a **conversational transaction**.

However, one disadvantage of a conversational program under CICS is that the application holds onto system resources while it waits for terminal input.

CICS pseudoconversational applications

If a conversational application is widely used, under CICS this could adversely affect overall system performance.

For this reason, CICS provides **pseudoconversational** support, in which a series of nonconversational transactions gives the appearance to the terminal user of a single conversational transaction.

The pseudoconversational version of the application outlined in “CICS conversational applications” on page 567 is as follows:

1. Start the application
2. If first invocation

   Then perform initialization

   Else

   - **Receive** the terminal input
   - Process the input
3. If finish not requested

   Then **send** data to the terminal (typically, display panel)

   Return to CICS requesting reinvocation of transaction

   Else Return to CICS

As you can see, the conversation is implemented as discrete **send** and **receive** calls, and while terminal input is being awaited, no transaction exists. CICS takes care of reading the input when the user enters it, and then starts a transaction to process it.

There are a number of considerations affecting the choice of conversational or pseudoconversational programming for a particular application–the amount of usage, and file integrity across transactions being examples.

Information about these and other considerations affecting application design under CICS can be found in the *CICS Application Programming Guide*.

A CICS pseudoconversational application appears to the terminal user as a normal conversational transaction, but is, in fact, a series of separate transactions where the CONVERSE is implemented as SEND and RECEIVE. One transaction ends with a SEND, and the next starts with a RECEIVE.

In this way, system resources can be released for the duration of “operator think time” thus making more efficient use of CICS.

GDDM provides pseudoconversational support for all types of alphanumeric data, for output-only graphics and images, and for partitions by means of a strictly defined protocol for GDDM application call sequences.

You can find an example of a CICS pseudoconversational transaction in “A CICS pseudoconversational programming example” on page 510.

GDDM provides two modes of pseudoconversation:

- Transaction-dependent mode
- Transaction-independent mode.

Transaction-dependent pseudoconversations

Essentially, while operating in this mode, GDDM storage and resources (except for device query data) are released at the termination of a particular transaction, and are reinitialized when the next transaction is reinvoked by CICS to process the next device input.

As no information is retained by GDDM across transactions (other than device query data), it is the responsibility of the transaction to maintain the continuity between the initial instance of GDDM and subsequent instances within the transaction. For example, you need to ensure that transactions maintain the output to screen and save, at the end of each instance, information that is require for subsequent ones. Some call sequences that can help you are described in “Typical call sequences for transaction-dependent pseudoconversations” on page 570.

GDDM provides the ability for a GDDM application to use CICS pseudoconversational programming by changing the function of the following calls, when transaction-dependent mode is being used:

DSOPEN
The PSCNVCTL processing option indicates to GDDM whether this mode is in use, and whether this is the start of it, or a continuation.

- The processing option group code is 25
- The length is 2 full-words
- The values are 0, 1, and 2 corresponding to NO, START, and CONTINUE respectively
- The default is NO.

The nickname syntax for this processing option is:

```
(PSCNVCTL,{NO|START|CONTINUE})
```

ASREAD
When the application is in “Continue pseudoconversational” mode (`PSCNVCTL,CONTINUE`), the first ASREAD call issued by the application causes the output transmission to be suppressed, and only the input part of the ASREAD call functions.

Subsequent ASREAD calls work in the usual way, that is, they result in output plus a “wait” for input. In this way, transactions can drop into conversational mode if they need to; see the description of the CLEAR key handling and line-output errors below.

Only the first ASREAD in CONTINUE pseudoconversational mode performs as a RECEIVE; subsequent ASREADs work as normal, that is they output, wait, and receive input.

Note: There is no pseudoconversational support for the GSREAD and MSREAD calls.

DSCLS
If transaction-dependent mode is in use, a DSCLS call always causes the device keyboard to be unlocked. Also, two options are provided that can be used by pseudoconversational applications to end the pseudoconversational mode, and are available to conversational applications to cause explicit keyboard Unlock.

The complete DSCLS options and their meanings are:

0 Erase the screen; if in transaction-dependent pseudoconversational mode, unlock the keyboard, and save any changed device data.

1 Do not erase the screen; if in transaction-dependent pseudoconversational mode, unlock the keyboard, and save any changed device data.

2 Erase the screen and unlock the keyboard; if in transaction-dependent pseudoconversational mode, release the saved device data.

3 Do not erase the screen but unlock the keyboard; if in transaction-dependent pseudoconversational mode, release the saved device data.

Typical call sequences for transaction-dependent pseudoconversations

The following application scenario illustrates the call protocol for transaction-dependent pseudoconversations:

- On the initial invocation of the transaction:
  - FSINIT
  - DSOPEN (Start pseudoconversational mode)
  - Create alphanumeric data for the first screen
  - Create any graphics output
  - FSFRCE
  - DSCLS (Option 1 – do not erase the screen)
  - FSTERM
  - EXEC CICS RETURN TRANSID(Tname) COMMAREA(Carea) LENGTH (Clen)

    The array “Carea” should contain all information required to continue the transaction processing, such as, Application Data Structures used for output of mapped data.
- On subsequent invocations of the transaction:
  - FSINIT.

  - DSOPEN (Continue pseudoconversational mode).
  - Create alphanumeric data for the “previous” screen using the identical set of calls used the last time, and also, if mapping is used, with the same Application Data Structures (as saved in “Carea”).
  - **Do not** issue any graphics calls.
  - ASREAD.
  - Process input in the usual way.
  - Create alphanumeric data for the next screen.
  - Create any graphics output.
  - FSFRCE.
  - DSCLS (Option 1 – do not erase the screen).
  - FSTERM.
  - EXEC CICS RETURN TRANSID(Tname) COMMAREA(Carea) LENGTH(Clen).

    The array “Carea” should contain any information required to continue the transaction processing; in particular, it should contain the ADSs used for the output of any mapped data.
- To terminate the pseudoconversational mode use DSCLS with options 2 or 3, and EXEC CICS RETURN without the TRANSID, COMMAREA, or LENGTH.

As stated above, the first ASREAD call in a transaction specifying “Continue pseudoconversational” mode, only performs the input function; all output is suppressed.

There are, however, two exceptions to this rule.

The first exception occurs, if the application uses mapped alphanumerics and the map group requests automatic handling of the CLEAR key. The ASREAD call performs as usual; that is, it bypasses output and processes the input data (only a cursor address and the CLEAR aid), whereupon mapping signals a screen refresh.

This affects the transaction in the same way as issuing a second ASREAD call; that is, the screen is output again and the transaction waits for input.

Thus the ASREAD call effectively works in the usual way, and the transaction becomes a conversation for this invocation.

The other exception occurs when GDDM issues a line-output error message before the ASREAD call.

In this case, the screen contents have been destroyed, and for GDDM to continue to process correctly, the screen has to be created again.

Thus once more, the ASREAD call works in the usual way; that is, output plus a “wait for input” and the transaction becomes “conversational” for this invocation.

Always-unlock-keyboard mode
Use of the always-unlock-keyboard processing option improves the performance of CICS pseudoconversational applications by unlocking the keyboard at FSFRCE instead of DSCLS.

Transaction-independent pseudoconversations

When CICTIF=EXT is specified on the SPINIT call, GDDM pseudoconversations run in transaction-independent mode. In this mode:

- GDDM does not have to be initialized and terminated by each pseudoconversational transaction. Instead, the GDDM instance can be retained between transactions, so the transactions only have to pass the GDDM application anchor block (AAB) to each other.
- The ASREAD call only receives input from the device. It never sends output to the device.
- The FSFRCE call should be used to send output to the device.
- The external default CICAUD should be set to YES to facilitate storage release in abnormal situations.
- The external default AUNLOCK should be set to YES to unlock the keyboard during the FSFRCE call.
- No special action is required to handle messages from CICS or GDDM.

The following restrictions apply:

- The EXT option requires CICS support for both SHARED and FLENGTH options in the EXEC CICS GETMAIN command.
- The I/O calls GSREAD, MSREAD, FSSHOW, FSSHOR, and WSIO are not supported in pseudoconversational mode, and must not be used.
- The processing options PSCNVCTL, CTLMODE, and WINDOW must have the value NO. If you specify any other values, they may give unpredictable results.

Typical call sequences for transaction-independent pseudoconversations

The following application scenario illustrates the call protocol for transaction-independent pseudoconversations:

- On the initial transaction of the application:
  - SPINIT (Specifying CICTIF=EXT, CICAUD, and AUNLOCK)
  - Calls to create alpha, graphic, and image data for the first screen
  - FSFRCE
  - EXEC CICS RETURN TRANSID(tname) COMMAREA(carea) LENGTH(clen)
- On subsequent transactions of the application:
  - ASREAD
  - Calls to process alpha input from the screen
  - Calls to create alpha, graphic, and image data for the next screen
  - FSFRCE
  - EXEC CICS RETURN TRANSID(tname) COMMAREA(carea) LENGTH(clen)
- On the final transaction of the application:
  - ASREAD
  - Calls to process alpha input from the screen
  - FSTERM
  - EXEC CICS RETURN

Note: High performance alphanumerics can be used in MOVE and LOCATE modes. In LOCATE mode, the field list, bundle list, and data buffer must all be located in shared storage.

Requesting transaction-independent services

When running under CICS, GDDM usually uses transaction-dependent services to acquire storage and load programs. That is, GDDM uses CICS services that ensure that storage and program resources are released should the task terminate normally or abnormally.

Application programs using SPINIT to initialize GDDM can request that transaction-independent services be used, by setting the CICTIF option in an encoded UDSL in the SPINIT call; see the *GDDM Base Application Programming Reference* book.

One of two options may be chosen:

- CICTIF=YES specifies transaction-independent mode. In this mode, GDDM uses CICS storage and program services in such a way that storage and program resources are not released at task or transaction termination.
- CICTIF=EXT specifies extended transaction-independent mode. In this mode, GDDM requests that all storage and program resources be allocated above 16MB, in addition to being retained at task or transaction termination. The function of the ASREAD call is also changed in this mode and becomes a read-only operation. CICS pseudoconversations written in this mode show significant performance improvements over those written for transaction-dependent mode.

Care must be taken when using these options, to ensure that resources are eventually released in all situations including abnormal termination of the task or transaction. The audit trail functions described in the following section can be used to monitor and control the status of the resources.

Using the resource audit trails

Care must be taken when requesting transaction-independent services as described above to ensure that resources are released in all situations including abnormal termination of the task or transaction.

Application programs requesting such services can also request resource audit trails, by specifying the CICAUD option in an encoded UDSL in the SPINIT call; see the *GDDM Base Application Programming Reference* book. The application program can use this option to provide the addresses of 4-byte audit trail anchors for storage and program resources.

The storage audit trail is maintained as follows:

- All blocks of storage acquired but not yet released by GDDM are chained together by 4-byte pointers at offset +0 in each storage block.
- The storage audit trail anchor, addressed by the CICAUD option, is set by GDDM to locate this chain of storage blocks.
- The 4-byte pointer in the last storage block in the chain is set to the initial value of the storage audit trail anchor, as defined by the application program.

- If all storage blocks were released (as at termination), the storage audit trail anchor is reset by GDDM to its initial value.

Thus, if abnormal termination occurs, the storage audit trail anchor can be used to locate those blocks of storage that are not yet released by GDDM. To be effective, the audit trail anchor should be initialized to an identifiable value, such as 0.

The program audit trail is maintained as follows,

- At initialization, GDDM allocates a “program hold” table of 41 entries, each eight bytes in length. All but the last entry are initialized to blanks. The last entry is an “end-of-table” marker and is initialized to a value of `X'FFFFFFFFFFFFFFFF'`
- The program audit trail anchor addressed by the CICAUD option is set by GDDM to address this program hold table.
- Whenever GDDM loads a program, it replaces a blank entry in the program hold table with the program name.
- Whenever GDDM deletes a program, it resets the corresponding entry in the program hold table to blanks.

Thus, if abnormal termination occurs, the program hold table can be used to determine the names of those programs that are not yet deleted by GDDM.

Note that the program hold table itself is in a storage block in the storage audit chain. Therefore, any processing of this table should be performed before processing the storage audit chain.

Using GDDM with Basic Mapping Support

It is possible to write a CICS transaction that uses both Basic Mapping Support (BMS) and GDDM functions to manage the screen. Three methods for doing this are described below. Note that GDDM uses CICS terminal control facilities to manage the screen directly. For this reason, GDDM pictures displayed on the terminal cannot be paged using BMS paging mechanisms.

An application program that uses both CICS terminal control and GDDM functions for input/output operations is subject to the same considerations. However, after GDDM is initialized, no transmissions should be sent by CICS terminal control that would alter the state of the device, other than the screen buffer. In particular, no structured fields to alter the state of PS sets (other than those reserved by the GDDM PSRSV call) should be transmitted.

Using GDDM and Basic Mapping Support consecutively

When GDDM has formatted the screen and displayed data by means of calls to ASREAD, or FSFRCE, or both of these, the displayed panel can be replaced with one generated by BMS using a command such as:

```
EXEC CICS SEND MAP('map-name')...ERASE
```

The ERASE option should be specified, because BMS is not aware of the GDDM screen interactions that occurred since the last BMS interaction.

The BMS map can use any of the field description functions supported by CICS, including references to PS sets loaded by GDDM calls. The application program can then read data entered by the terminal user using BMS.

When the BMS interactions are completed, GDDM can be called again to present the original or updated data. A call to FSREST(0) should be issued before calling FSFRCE or ASREAD, because GDDM would not be aware of the BMS screen interactions. GDDM interactions can then continue until the application program calls BMS again.

Using GDDM and BMS concurrently without coordination mode

It is possible to use GDDM and BMS to display data at the same time on the same screen. In this type of operation, it is recommended that GDDM be used only to **output** graphics data, and that BMS be used for all alphanumeric input/output processing. Specifically, the GDDM ASMODE function should **not** be used to set the character reply mode.

The GDDM picture should be presented first, using FSREST(0) if necessary to clear any preceding BMS data. The BMS map(s) should then be transmitted, omitting the ERASE option. The map(s) should be defined so that all screen areas used by GDDM for graphics are in protected fields with normal attributes (nonhighlighted, nonselectable, neutral color, normal intensity, and standard character set). The application program can then read data entered by the terminal user using BMS.

On completion of terminal data entry, the GDDM FSREST(0) call should again be used on resuming GDDM operations.

If the FSCOPY call is used to copy a panel containing both GDDM and BMS data, only the GDDM data is printed, because GDDM is unaware of the BMS data.

Using GDDM and BMS concurrently with coordination mode

Note: BMS is not supported with CICS pseudoconversational modes.

The difficulty with the above method of using both BMS and GDDM is that whenever GDDM rewrites the screen it may choose to totally erase the screen and start afresh. This, of course, also removes any existing BMS output.

This problem is avoided if the device used for output is explicitly opened with the DSOPEN statement and the “coordination” mode of operation selected.

When GDDM generates the data streams for such a device, it **never** totally erases the screen when an FSFRCE or ASREAD is issued. Instead it just rewrites the contents of the area covered by the graphics field. Any screen erasure required then becomes the responsibility of the application using either Terminal Control or BMS requests.

The following points should be noted:

- GDDM protects the graphics field by a column of attribute bytes to its left, or at the end of the preceding row if the graphics field is positioned in the first column.

  The BMS maps should not use the area used by these attribute bytes. If they do, the results are unpredictable.

- GDDM locks the keyboard when the device is opened, to interrogate the device properties. Therefore, any BMS request to release the keyboard should be issued after calling GDDM to open the device.
- GDDM writes only to the area of the screen covered by the graphics field. Further, no alphanumeric fields, even if they are within the graphics field, are written to the screen.
- ASREAD does not wait for input – it behaves as FSFRCE.
- Programmed symbol (PS) sets may still be loaded within coordination mode.
- The application program must erase the screen before issuing the first GDDM output request, to establish either the default or alternate screen size.
- After receipt of a CLEAR key the application should rewrite the BMS portions of the screen before issuing FSREST and FSFRCE calls to reestablish the GDDM picture.
- The action of the default error exit is to erase the screen and display a prompting message. This causes disruption of the BMS-managed screen layout. Therefore, the application should use the FSEXIT function to redefine the handling of errors.

CICS GDDM default error exit

The function of the GDDM Default Error Exit is generally described under “Specifying an error exit and threshold, using call FSEXIT” on page 137. When GDDM is running under CICS, the Default Error Exit operates as follows:

- The screen is cleared, and diagnostic messages describing the error are displayed.
- Another message, describing the other actions available to the terminal user, is displayed.
- If the terminal user presses the CLEAR key at this point, the screen is cleared and GDDM returns control to the point in the application program where the error exit was invoked. GDDM also retransmits the screen buffer contents on the next terminal input/output-related call.
- If the terminal user uses any key other than CLEAR, GDDM calls the CICS Command Level ABEND facility with an ABCODE of “G000”, indicating that the ABEND is in response to an error message displayed on the terminal.

In either of the above cases, GDDM tries to write one or more error-log records to the CICS transient data destination ADML, if it was specified in the CICS destination control table. The error-log records contain the diagnostic messages displayed on the terminal, prefixed by transaction identification information, as follows:

Byte 0...3	4	5...8	9	10...13	14 15	16...
Transaction ID		Task Number		Terminal ID		Diagnostic Text

Note that in the special case of initialization errors a choice of action is not available to the terminal user after the diagnostic message is displayed. For these

errors, GDDM unconditionally ABENDS, with an ABCODE of “G000”, after displaying the corresponding diagnostic message on the terminal.

Display terminal conventions

In general, the CLEAR key and all PA and PF keys are available to be returned as terminal input by means of the GDDM ASREAD function. However, specific PA keys that were defined in the CICS system initialization table (SIT) for other purposes, such as printing, are not available for GDDM purposes.

If the terminal keyboard has a PA3 key, the default action when pressing it is to activate user control. If user control is not available, or if a key other than PA3 has been designated for activating user control, then PA3 causes the screen to be refreshed. PA3 is never passed to the application.

Using the GDDM nonreentrant interface

GDDM provides a mechanism for using the nonreentrant interface form under CICS while still allowing GDDM and its invoking application program to be quasi-reentrant. To do this, the application programmer should reserve an area of 8 bytes in the associated Transaction Work Area (TWA). This may require changes in the corresponding transaction definition in the CICS program control table (PCT). The programmer should then define an external control section (CSECT) named ADMUOFF, to be link-edited with the application program and the GDDM nonreentrant interface module. This should contain a full-word defining the offset in the TWA of the area reserved for GDDM’s use.

Thus, for application programs that would not otherwise require a TWA, the following would be sufficient:

1. Define a TWA of length 8 bytes by specifying the corresponding option in the transaction definition in the CICS program control table.
2. Define an ADMUOFF CSECT containing a full-word of value zero, to be link-edited with the application program.

The ADMUOFF CSECT can be defined using standard Assembler language facilities. Thus:

```
ADMUOFF CSECT
INIT    DC   F'0'
        END
```

Alternatively, high-level language constructs can be used, where such are available. In PL/I, the CSECT can be generated by a declaration of the form:

```
DECLARE ADMUOFF STATIC EXTERNAL FIXED BINARY (31) INITIAL(0);
```

In C/370, the CSECT can be generated by adding the following statement to the source code of a GDDM application program.

```
extern long int ADMUOFF=0;
```

GDDM uses the area reserved in the TWA to store an Application Anchor Block (AAB), in the format described for the reentrant interface in the *GDDM Base Application Programming Reference* book. When the nonreentrant interface is

invoked, GDDM verifies that the value contained in ADMUOFF is consistent with the length of the TWA defined for the invoking transaction.

Through this mechanism, GDDM operates in a quasi-reentrant way. Although the GDDM nonreentrant interface module is **not** read-only, it does not prevent an invoking transaction from servicing more than one CICS terminal at the same time.

Using the GDDM system programmer interface with dynamic load

If an application uses only the system programmer interface, all invocations of GDDM are through the entry point ADMASP. This entry point can be resolved on MVS by link-editing the application with the GDDM interface module ADMASPC, as described under "Link-editing GDDM applications with CICS on MVS" on page 584. On VSE this entry point can be resolved by link-editing the application with the GDDM interface modules ADMASLC and ADMASP as described under "Link-editing a GDDM application with CICS on VSE" on page 584.

Or, the application can avoid these linkage-edit considerations by using CICS facilities (EXEC CICS LOAD) to load dynamically a GDDM interface module ADMASPLC containing the ADMASP entry point as follows:

```
EXEC CICS LOAD PROGRAM(ADMASPLC) ENTRY(admasp-addr)
     SET(dummy-var)
```

Data sets and file processing

When running under CICS, GDDM Base and GDDM-PGF use three types of data sets:

- VSAM key-sequenced data sets
- CICS transient data queues
- CICS temporary storage data sets.

GDDM-IMD uses additional types of file processing; for more information, see the *GDDM Interactive Map Definition* book.

VSAM key-sequenced data sets

GDDM uses VSAM key-sequenced data sets for:

- Image Symbol Sets (ISS) and Vector Symbol Sets (VSS), as required by calls GSLSS, PSLSS, PSLSSC, SSREAD, and SSWRT, and through the Image Symbol Editor.
- Device-dependent pictures, as required by calls to FSSAVE, FSSHOR, and FSSHOW.
- GDDM-IMD-generated mapgroups, as required by calls to MSPCRT, MSQADS, MSQGRP, MSQMAP, and MSREAD.
- Graphics Data Format (ADMGDF) files, as required by calls to GSSAVE and GSLOAD.
- Image files, as required by calls to IMAPT and IMAGT.

GDDM maintains ADMSAVE, ADMGGMAP, ADMSYMBL, ADMIMG, ADMPROJ, and ADMGDF files as keyed records in VSAM key-sequenced data sets shared by transactions running in the CICS subsystem. These data sets must be defined in

the CICS file control table (FCT). The VSAM data sets must be opened, either when CICS is initialized, or dynamically, before GDDM requires access to them. The underlying MVS or VSE data sets must have characteristics as shown in Table 17 on page 580. Procedures for creating and initializing suitable VSAM data sets are described in the *GDDM/MVS Program Directory* or the *GDDM/VSE Program Directory*.

The default VSAM data set names are as defined in Table 17 on page 580. These names can be changed, if required, after installation, as described in the *GDDM System Customization and Administration* book.

The use of the VSAM data sets can be controlled by the ESLIB call, the syntax of which is described in the *GDDM Base Application Programming Reference* book. This call establishes the set of VSAM data sets that are to be used to store or retrieve a given type of object. The VSAM data sets used are identified to this call by a list of file names.

The VSAM data sets identified are searched in the order given in an attempt to find an object. An object is stored only by means of the first data set name of the list, even though it may have been retrieved from another one. If no data set name list is provided, only the default data set name is used for retrieving and storing GDDM objects.

GDDM ensures the integrity of data as it is written or read on the VSAM data sets. Specifically, GDDM ensures that the particular records defining the content of a symbol set, picture, or generated mapgroup cannot be updated by one transaction while being read by another. If additional control of the use of the VSAM data sets is required (such as restricted write access), this should be implemented by security mechanisms external to GDDM, such as described in the *CICS Facilities and Planning Guide*.

GDDM symbol sets, pictures, generated mapgroups, and ADMGDF files are stored on the VSAM data sets as 400-byte records, with an embedded key in the first 20 bytes, as follows:

Byte 0.....7	8....15	16....19	20....
Name	Type	Record sequence number	Data

Name is that specified in the GDDM call as “symbol-set-name”, “picture-name”, “group-name”, or “name”, subject to the character-substitution rules described in “Selecting symbol sets by device type” on page 395.
Type is an 8-byte character string identifying the type of the record, for example, “symbol set” or “picture”, and is defined in Table 17 on page 580.
Record sequence number is a 4-byte binary full-word that sequences and uniquely identifies each record within a symbol set or picture.

This key format is such that, if required, all of the records defining a specific symbol set or picture can be deleted without calling GDDM. This can be done by using the CICS file control GENERIC DELETE function:

```
EXEC CICS DELETE DATASET (VSAM-data-set-name)
          RIDFLD (first-16-bytes-of-key)
          KEYLENGTH(16)
          GENERIC
```

The Interactive Chart Utility (part of GDDM-PGF) includes a directory function that supports list, delete, and copy operations on GDDM objects such as symbol sets, pictures, generated mapgroups, and ADMGDF files.

Table 17 (Page 1 of 2). GDDM data-set characteristics for CICS

Type of data	GDDM default name or record type	Data-set type	Data characteristics
Symbol sets	Data set name = ADMF Record type = ADMSYMBL	Records in VSAM data set	RECORDSIZE (400 400) KEYS(20 0)
Pictures saved using GSSAVE	Data set name = ADMF Record type = ADMGDF	Records in VSAM data set	RECORDSIZE (400 400) KEYS(20 0)
Pictures saved using FSSAVE	Data set name = ADMF Record type = ADMSAVE	Records in VSAM data set	RECORDSIZE (400 400) KEYS(20 0)
Generated mapgroup	Data set name = ADMF Record type = ADMGGMAP	Records in VSAM data set	RECORDSIZE (400 400) KEYS(20 0)
Image files	Data set name = ADMF Record type = ADMIMG	Records in VSAM data set	RECORDSIZE (400 400) KEYS(20 0)
Image projection files	Data set name = ADMF Record type = ADMPROJ	Records in VSAM data set	RECORDSIZE (400 400) KEYS(20 0)
Object modules	Queue name = ADMD	Transient data queue	Fixed-length records, length 80 bytes
System printer output	Queue name = ADMS	Transient data queue	Variable-length records, length 142 bytes or greater (see note 3)
Queued printer files	(assigned by GDDM)	Temporary storage data set	(assigned by GDDM)
Trace records	Queue name = ADMT	Transient data queue	Variable-length records, maximum length 137 bytes (including 4-byte RDW)
Error log records	Queue name = ADML (cannot be modified)	Transient data queue	Variable-length records, maximum length 120 bytes
External defaults files	Queue name = ADMDxxxx (xxxx is the CICS terminal identifier)	Temporary storage data set	Variable-length records, maximum length 256 bytes
Pseudoconversational saved device information	Queue name = ADMQxxxx (xxxx is the CICS terminal identifier)	Temporary storage data set	Assigned by GDDM
CDPDS	(none)	Temporary storage data set	Variable length records, maximum length 8200 bytes

Table 17 (Page 2 of 2). GDDM data-set characteristics for CICS

Type of data	GDDM default name or record type	Data-set type	Data characteristics
AFPDS	(none)	Temporary storage data set	Variable length records, maximum length 8202 bytes

notes:

Record types for data stored in VSAM data sets cannot be changed.

1. For Transient Data VSE disk output data sets, another 8 bytes, required by LIOCS for creation of the count field, should be added to the block size.
2. The definition of Transient Data queues for System Printer Output should indicate the use of ASA control characters,
   for MVS RECFORM = VARUNBA or VARBLKA
   for VSE CTLCHR = YES
3. The record length specified for System Printer Output queues should be enough to contain the 4-byte Record Descriptor Word (RDW), the ASA control character, any Translation Reference Character (TRC) for 3800 devices, and the maximum number of columns for the type of System Printer selected by the application. The value of 142 is enough for any of the System Printer device-characteristic tokens supplied with GDDM.
4. The output for all 3800 devices should contain table reference characters (TRCs) and so, for MVS, the parameter DCB=OPTCD=J must be included in the output JCL. Under MVS or VSE, additional DCB or SETPRT parameters, such as CHARS, FLASH, FORMS, and so on, may be required.
5. For more information, see the *MVS JCL* manual or the *VSE System Control Statements* manual.

Transient data queues

GDDM uses CICS transient data queues for:

- Object modules resulting from requests from the Image Symbol Editor
- Output destined for a system printer device as the result of calls to DSOPEN and DSCLS
- Trace records resulting from the FSTRCE function
- Error log records resulting from invocation of the GDDM CICS Default Error Exit.

Object modules are written consecutively to a single transient data destination. This must be defined in the CICS destination control table (DCT), typically in a manner that would route the object modules to a predefined extrapartition data set. Each object module generated contains a control section (CSECT) with the name as specified by the appropriate utility, and has a form suitable for link-editing with an application program for subsequent reference, typically using the GSDSS or PSDSS calls.

System printer device output is written to the transient data destination identified using the DSOPEN call. This must be defined in the CICS destination control table (DCT), typically in a manner that would route the output to a predefined extrapartition spool data set. If so routed, the definition should indicate the presence of ASA control characters in the data generated by GDDM.

GDDM ensures that system printer output resulting from a single DSOPEN...DSCLS sequence remains contiguous, and is not interleaved with the output from another CICS transaction. The application programmer should ensure that the use of these facilities in multiple transactions does not introduce excessive transaction delays or interlocks.

Trace records are written to a single transient data destination. This must be defined in the CICS DCT, typically in a manner that would route the output to a predefined extrapartition spool data set. If so routed, the definition should indicate the presence of ASA control characters in the records generated by GDDM.

Trace records from different transactions may be interleaved. For this reason, each record contains the corresponding transaction name and terminal identifier. For a description of the use of the FSTRCE function, and of the format of the trace records, see the *GDDM Diagnosis* book.

For information on the trace facilities obtainable with the GDDM external default TRCESTR, see the *&bapr.* manual.

The above transient data destination names are as defined in Table 17 on page 580. These names can be changed, if required, after installation (by specifying a value for the CICTRCE option, as described in the table of GDDM external defaults for CICS in the *GDDM Base Application Programming Reference* book.

Error log records are written as they occur, to a single transient data destination, which must be defined in the CICS destination control table (DCT), in a manner to suit the installation's requirements. Typically, the destination would be defined as an extrapartition destination, which would route the error log records to an external data set for subsequent printing.

Error log records from different transactions may be interleaved. For this reason, each record contains the corresponding transaction name, number, and terminal identifier. The format of these error log records is described under "CICS GDDM default error exit" on page 576. The Transient Data destination name for error log records is ADML, and cannot be changed.

The programmer should ensure that the Transient Data destination names required are all defined in the appropriate CICS tables. The underlying MVS or VSE data sets must have characteristics as shown in Table 17 on page 580.

Temporary storage data sets

GDDM creates CICS temporary storage data sets to hold intermediate data sets used in the processing of calls to DSOPEN, DSCLS, FSOPEN, and FSCLS for queued printer output. The temporary data sets created are read by the GDDM CICS Print Utility (ADMOPUC), and after output to the printer is completed, the data sets are purged.

By default, for queued printer output, GDDM selects temporary storage queue names beginning with the prefix "ADMT". This prefix can be changed, if required, by specifying a value for the CICTSPX option, as described in the *GDDM Base Application Programming Reference* book.

GDDM also reads a CICS temporary storage data set containing a temporary External Defaults file. Such a file is intended to be used for problem determination purposes only. For more information, see the *GDDM Diagnosis* book.

By default, for External Defaults files, GDDM assumes temporary storage queue names beginning with the prefix "ADMD". This prefix can be changed, if required, by specifying a value for the CICDFPX option, as described in the table of external

defaults for CICS subsystems in the *GDDM Base Application Programming Reference* book.

Also, GDDM uses temporary storage to hold Device Query data when running in transaction-dependent pseudoconversational mode. The queue name is formed from a prefix “ADMQ”, which can be changed, if required, by specifying a value for the CICTQRY option (as described in the table of external defaults for CICS subsystems in the *GDDM Base Application Programming Reference* book), and also the terminal identifier.

Programming under CICS on extensions of MVS

This section describes the special programming considerations for 31-bit mode GDDM applications, and provides general information on GDDM code in application programs that can run under CICS on the MVS/ESA or VSE/ESA operating systems.

GDDM code above 16 megabytes

Under suitable subsystems and operating systems, the main body of GDDM code can reside above 16MB.

Under CICS, the only exceptions are the Call Format Descriptor Module and the APL Request Codes Module. These modules have RMODE(24) to ensure addressability from 24-bit mode applications.

Application code above 16 megabytes

Under suitable releases of CICS/ESA or CICS/VSE, GDDM applications can reside above 16MB.

AMODE(31) applications and application parameters above 16 megabytes

Under CICS, GDDM applications can run in 31-bit mode and pass to GDDM parameters that are located above 16 megabytes.

If GDDM is called in 31-bit mode, it assumes that any parameter addresses that are passed represent 31-bit addresses.

Application programming considerations

Under all MVS/ESA or VSE/ESA systems, a GDDM application program may have any valid AMODE attribute, and may call GDDM in any mode (24-bit or 31-bit) consistent with its location. In fact, it is possible (though not recommended) for an application program to call GDDM in both 24-bit and 31-bit modes in the same session.

User exits

A number of other user exits can be defined for programs using the SPI. These exits and the special consideration for their use on MVS/ESA or VSE/ESA are described in the *GDDM Base Application Programming Reference* book.

Compiling and link-editing GDDM application programs

Examples of JCL that can be used to compile and link-edit application programs written in COBOL, C/370, PL/I, and Assembler are listed in "Example of JCL for compiling and linking PL/I GDDM/CICS applications on MVS" on page 586 through "Example: JCL for GDDM under CICS/VSE using Assembler" on page 593 at the end of this appendix.

Compiling a PL/I program

If you use the GDDM-supplied declarations in your program, you must access the libraries containing them before compiling.

Link-editing a GDDM application program under CICS

An application program using GDDM under CICS must be link-edited with CICS command-level (EXEC) stubs in the usual way, as described in the *CICS/ESA System Definition Guide*. Unless the application program uses dynamic load facilities to access GDDM using the System Programmer Interface (see "Using the GDDM system programmer interface with dynamic load" on page 578), the program must also be link-edited with an appropriate GDDM interface module or modules.

Link-editing GDDM applications with CICS on MVS

For MVS, the required interface module can be explicitly included in the link-edit process. Or, if the application program uses one of the other FSINIT entry points described in the *GDDM Base Application Programming Reference* book. the interface module can be included by linkage editor automatic library call facilities. The following is a list of GDDM CICS interface modules for MVS:

Interface	Interface module	FSINIT alternative entry
Nonreentrant	ADMASNC	FSINNC
Reentrant	ADASRC	FSINRC
System Programmer	ADMASPC	—

The GDDM interface modules for CICs on MVS are supplied prelinked with the CICS language stub DFHELII. This can cause message IEW0241 to be issued for CSECTS with names beginning DFH and DLZ when you link edit your application with the GDDM interface modules. You can safely ignore this message.

Link-editing a GDDM application with CICS on VSE

For VSE, two GDDM interface modules are required, and they should be explicitly included in the link-edit process. The first interface module should be selected according to the form of interface used by the application program as follows:

Interface	Interface module
Nonreentrant	ADMASNB

Interface	Interface module
Reentrant	ADMASRB
System programmer	ADMASP

The second GDDM interface module required is ADMASLC.

In the absence of an explicit ENTRY statement, it is important to include the application program module before the relevant GDDM interface modules, to ensure that the application program entry point is correctly identified.

Note: The ADMASNO and ADMASRO interface modules for GDDM-PGF have been deleted. All the stub information for GDDM Base and PGF is now contained in ADMASNB and ADMASRB. References to ADMASNO and ADMASRO should be deleted from the jobstreams you use to link GDDM applications.

Example of JCL for compiling and linking PL/I GDDM/CICS applications on MVS

```
//******************** CICS/ESA ***************************************
//*
//* Example of JCL to translate, compile, and link-edit an MVS/GDDM/CICS
//* sample program or user-written application.
//*
//* This JCL assumes the use of the CICS-supplied
//* cataloged procedure "DFHEITPL".
//*
//*********************************************************************
//*
//jobname    JOB   accounting info,..........
//           EXEC  PROC=DFHEITPL
//*
//* Translation step
//*
//TRN.SYSIN  DD     DISP=OLD, DSN=your_application source (member)
//* Remember to define ADMUOFF if the program uses the nonreentrant
//* interface.  (See "Using the GDDM nonreentrant interface" on page 577)
//*
//* Compilation step
//*
//* Add a SYSLIB DD statement to reference library containing GDDM sample
//* PL/I declarations, as shown and any additional user libraries
//* required, for example libraries containing GDDM-IMD ADSs, as shown.
//*
//PLI.SYSLIB DD  DSN=GDDM.SADMSAM,DISP=SHR
//           DD   DSN=user.gddm.ads-lib,DISP=SHR
//*
//* Link-edit step
//*
//* Insert a SYSLIB DD statement to reference library containing GDDM
//* interface modules, as shown.
//*
//* In the specified INCLUDE statement,
//* leave   ADMASNC unchanged  if using the nonreentrant interface
//* replace ADMASNC by ADMASRC if using the reentrant interface
//*            or   by ADMASPC if using the system programmer interface
//*
//LKED.SYSLIB DD  DSN=GDDM.SADMMOD,DISP=SHR
//LKED.SYSIN DD *
 INCLUDE INCLIB(ADMASNC)
 NAME xxxxxxxx(R)        Sample Program or Application Name
//*
```

Example: JCL to compile and link COBOL GDDM/CICS applications on MVS

```
//******************** CICS/ESA COBOL ********************************
//*
//* Example of JCL to translate, compile, and link-edit a MVS/GDDM/CICS
//* sample program or user-written application.
//*
//* This JCL assumes the use of the CICS-supplied
//* cataloged procedure “DFHEITCL”.
//*
//*********************************************************************
//*
//jobname    JOB   accounting info,..........
//           EXEC  PROC=DFHEITCL,PARM.COB='as-required-by-CICS'
//*
//* Translation step
//*
//TRN.SYSIN  DD    DSP=OLD, DSN=your_application source (member)
//* Remember to define ADMUOFF if the program uses the nonreentrant
//* interface.  (See “Using the GDDM nonreentrant interface” on page 577)
//*
//* Compilation step
//*
//* Add a SYSLIB DD statement to reference any additional user
//* libraries required, for example libraries containing GDDM-IMD ADSs,
//* as shown.
//*
//COB.SYSLIB DD   DSN=user.gddm.ads-lib,DISP=SHR
//*
//* Link-edit step
//*
//* Insert a SYSLIB DD statement to reference library containing GDDM
//* interface modules, as shown.
//*
//* In the specified INCLUDE statement,
//* leave   ADMASNC unchanged  if using the nonreentrant interface
//* replace ADMASNC by ADMASRC if using the reentrant interface
//*            or   by ADMASPC if using the system programmer interface
//*
//LKED.SYSLIB DD  DSN=GDDM.SADMMOD,DISP=SHR
//LKED.SYSIN DD *
 INCLUDE INCLIB(ADMASNC)
 NAME xxxxxxxx(R)        Sample Program or Application Name
//*
```

Example of JCL for compiling and linking C/370 GDDM/CICS applications on MVS

```
//******************** CICS/ESA C/370**********************************
//*
//* Example of JCL to translate, compile, and link-edit an MVS/GDDM/CICS
//* sample program or user-written application.
//*
//* This JCL assumes the use of the CICS-supplied
//* cataloged procedure "DFHEITDL".
//*
//*********************************************************************
//*
//jobname    JOB   accounting info,..........
//           EXEC  PROC=DFHEITDL
//*
//* Translation step
//*
//TRN.SYSIN  DD     DISP=OLD, DSN=your_application source (member)
//* Remember to define ADMUOFF if the program uses the nonreentrant
//* interface.  (See "Using the GDDM nonreentrant interface" on page 577)
//*
//* Compilation step
//*
//* Add a SYSLIB DD statement to reference library containing GDDM sample
//* PL/I declarations, as shown and any additional user libraries
//* required, for example libraries containing GDDM-IMD ADSs, as shown.
//*
//C.SYSLIB DD  DSN=GDDM.SADMSAM,DISP=SHR
//           DD   DSN=user.gddm.ads-lib,DISP=SHR
//*
//* Link-edit step
//*
//* Insert a SYSLIB DD statement to reference library containing GDDM
//* interface modules, as shown.
//*
//* In the specified INCLUDE statement,
//* leave   ADMASNC unchanged  if using the nonreentrant interface
//* replace ADMASNC by ADMASRC if using the reentrant interface
//*            or   by ADMASPC if using the system programmer interface
//*
//LKED.SYSLIB DD  DSN=GDDM.SADMMOD,DISP=SHR
//LKED.SYSIN DD *
 INCLUDE INCLIB(ADMASNC)
 NAME xxxxxxxx(R)        Sample Program or Application Name
//*
```

Example: JCL to assemble and link-edit Assembler GDDM/CICS applications on MVS

```
//******************** CICS/ESA ASSEMBLER **************************
//*
//* Example of JCL to translate, assemble, and link-edit a GDDM/CICS
//* sample program or user-written application.
//*
//* This JCL assumes the use of the CICS-supplied
//* cataloged procedure "DFHEITAL".
//*
//*********************************************************************
//*
//jobname    JOB   accounting info,..........
//           EXEC  PROC=DFHEITAL
//*
//* Translation step
//*
//TRN.SYSIN  DD    DSP=OLD, DSN=your_application source (member)
//* Remember to define ADMUOFF if the program uses the nonreentrant
//* interface.  (See "Using the GDDM nonreentrant interface" on page 577)
//*
//* Assemble step
//*
//* Add a SYSLIB DD statement to reference any additional user libraries
//* required, for example libraries containing GDDM-IMD ADSs, as shown.
//*
//ASM.SYSLIB DD
//           DD
//           DD   DSN=user.gddm.ads-lib,DISP=SHR
//*
//* Link-edit step
//*
//* Insert a SYSLIB statement to reference library containing GDDM
//* interface modules, as shown.
//*
//* In the specified INCLUDE statement,
//* leave   ADMASNC unchanged  if using the nonreentrant interface
//* replace ADMASNC by ADMASRC if using the reentrant interface
//*            or   by ADMASPC if using the system programmer interface
//*
//LKED.SYSLIB DD  DSN=GDDM.SADMMOD,DISP=SHR
//LKED.SYSIN DD *
 INCLUDE INCLIB(ADMASNC)
 NAME xxxxxxxx(R)        Sample Program or Application Name
//*
```

Example: JCL to compile and link PL/I GDDM/CICS applications on VSE

```
******************** CICS/VSE ****************************************
*
* Example of JCL to translate, compile, and link-edit a GDDM/CICS
* sample program or user-written application.
*
* This JCL assumes that DLBL, EXTENT, and LIBDEF statements have
* already been used to:
*   - Define the GDDM sample source statement libraries
*   - Define the GDDM relocatable libraries
*
* Add additional statements to define any additional user source
* statement libraries required (for example, libraries containing
* GDDM-IMD ADSs).
**********************************************************************
*
// JOB     jobname
// DLBL    IJSYSPH,'PL/I.TRANSLATION',yy/ddd
// EXTENT  SYSPCH,balance of extent information
ASSGN  SYSPCH,DISK,VOL=volid,SHR
// EXEC    DFHEPP1$
*PROCESS   INCLUDE;
      . . . .
 Source deck here.
 Remember to define ADMUOFF if the program uses the nonreentrant
 interface.  (See “Using the GDDM nonreentrant interface” on page 577)
      . . . .
/*
CLOSE      SYSPCH,PUNCH
// DLBL    IJSYSIN,'PL/I.TRANSLATION',yy/ddd
// EXTENT  SYSIPT
ASSGN  SYSIPT,DISK,VOL=volid,SHR
// OPTION  CATAL
   PHASE   phase-name,*
   INCLUDE DFHPL1I
*
* In the following INCLUDE statement,
* leave   ADMASNB unchanged  for GDDM using nonreentrant interface
* replace ADMASNB by ADMASRB for GDDM using reentrant interface
*            or   by ADMASPC if using the system programmer interface
*
   INCLUDE ADMASNB
   INCLUDE ADMASLC
// EXEC    PLIOPT
// EXEC    LNKEDT
/&
// JOB     RESET
CLOSE      SYSIPT,SYSRDR
/&
```

Example: JCL to compile and link COBOL GDDM/CICS applications on VSE

```
******************** CICS/VSE COBOL ******************************
*
* Example of JCL to translate, compile, and link-edit a GDDM/CICS
* sample program or user-written application.
*
* This JCL assumes that DLBL, EXTENT, and LIBDEF statements have
* already been used to:
*   - Define the GDDM sample source statement libraries
*   - Define the GDDM relocatable libraries
*
* Add additional statements to define any additional user source
* statement libraries required (for example, libraries containing
* GDDM-IMD ADSs).
**********************************************************************
*
// JOB     jobname
// DLBL    IJSYSPH,'COBOL.TRANSLATION',yy/ddd
// EXTENT  SYSPCH,balance of extent information
ASSGN  SYSPCH,DISK,VOL=volid,SHR
// EXEC    DFHECP1$
 CBL LIB
      . . . .
 Source deck here.
 Remember to define ADMUOFF if the program uses the nonreentrant
 interface.  (See “Using the GDDM nonreentrant interface” on page 577)
      . . . .
/*
CLOSE      SYSPCH,PUNCH
// DLBL    IJSYSIN,'COBOL.TRANSLATION',yy/ddd
// EXTENT  SYSIPT
ASSGN  SYSIPT,DISK,VOL=volid,SHR
// OPTION  SYM,ERRS,NODECK,CATAL
   PHASE   phase-name,*
   INCLUDE DFHECI
// EXEC    FCOBOL
*
* In the following INCLUDE statement,
* leave   ADMASNB unchanged  for GDDM using nonreentrant interface
* replace ADMASNB by ADMASRB for GDDM using reentrant interface
*            or   by ADMASPC if using the system programmer interface
*
   INCLUDE ADMASNB
   INCLUDE ADMASLC
// EXEC    LNKEDT
/&
// JOB     RESET
CLOSE      SYSIPT,SYSRDR
/&
```

Example: JCL to compile and link C/370 GDDM/CICS applications on VSE

```
********************** CICS/VSE ****************************************
*
* Example of JCL to translate, compile, and link-edit a GDDM/CICS
* sample program or user-written application.
*
* This JCL assumes that DLBL, EXTENT, and LIBDEF statements have
* already been used to:
*   - Define the GDDM sample source statement libraries
*   - Define the GDDM relocatable libraries
*
* Add additional statements to define any additional user source
* statement libraries required (for example, libraries containing
* GDDM-IMD ADSs).
*********************************************************************
*
// JOB     jobname
// DLBL    IJSYSPH,'PL/I.TRANSLATION',yy/ddd
// EXTENT  SYSPCH,balance of extent information
ASSGN  SYSPCH,DISK,VOL=volid,SHR
// EXEC    DFHEPP1$
*PROCESS   INCLUDE;
      . . . .
 Source deck here.
 Remember to define ADMUOFF if the program uses the nonreentrant
 interface.  (See “Using the GDDM nonreentrant interface” on page 577)
      . . . .
/*
CLOSE      SYSPCH,PUNCH
// DLBL    IJSYSIN,'C/370.TRANSLATION',yy/ddd
// EXTENT  SYSIPT
ASSGN  SYSIPT,DISK,VOL=volid,SHR
// OPTION  CATAL
   PHASE   phase-name,*
   INCLUDE DFHPL1I
*
* In the following INCLUDE statement,
* leave   ADMASNB unchanged  for GDDM using nonreentrant interface
* replace ADMASNB by ADMASRB for GDDM using reentrant interface
*            or   by ADMASPC if using the system programmer interface
*
   INCLUDE ADMASNB
   INCLUDE ADMASLC
// EXEC    PLIOPT
// EXEC    LNKEDT
/&
// JOB     RESET
CLOSE      SYSIPT,SYSRDR
/&
```

Example: JCL for GDDM under CICS/VSE using Assembler

```
******************** CICS/VSE ASSEMBLER *************************
*
* Example of JCL to translate, compile, and link-edit a GDDM/CICS
* sample program or user-written application.
*
* This JCL assumes that DLBL, EXTENT, and LIBDEF statements have
* already been used to:
*   - Define the GDDM sample source statement libraries
*   - Define the GDDM relocatable libraries
*
* Add additional statements to define any additional user source
* statement libraries required (for example, libraries containing
* GDDM-IMD ADSs).
*
*********************************************************************
*
// JOB     jobname
// DLBL    IJSYSPH,'ASM.TRANSLATION',yy/ddd
// EXTENT  SYSPCH,balance of extent information
ASSGN  SYSPCH,DISK,VOL=volid,SHR
// EXEC    DFHEAP1$
      . . . .
      . . . .
 Source deck here.
 Remember to define ADMUOFF if the program uses the nonreentrant
 interface.  (See “Using the GDDM nonreentrant interface” on page 577)
      . . . .
      . . . .
/*
CLOSE      SYSPCH,PUNCH
// DLBL    IJSYSIN,'ASM.TRANSLATION',yy/ddd
// EXTENT  SYSIPT
ASSGN  SYSIPT,DISK,VOL=volid,SHR
// OPTION  SYM,ERRS,NODECK,CATAL
   PHASE   phase-name,*
   INCLUDE DFHEAI
*
* In the following INCLUDE statement,
* leave   ADMASNB unchanged  for GDDM using nonreentrant interface
* replace ADMASNB by ADMASRB for GDDM using reentrant interface
*            or   by ADMASPC if using the system programmer interface
*
   INCLUDE ADMASNB
   INCLUDE ADMASLC
// EXEC    ASSEMBLY
// EXEC    LNKEDT
/&
// JOB     RESET
CLOSE      SYSIPT,SYSRDR
/&
```

Appendix F. Programming with GDDM using VSE batch mode

GDDM application programs can be run in batch mode under VSE, provided the only devices that they open are page printers – in GDDM terms, family-4 devices.

GDDM page printer output takes the form of a file containing either a primary data stream, a page segment or an overlay.

A primary data stream is a complete document suitable for processing by a print-driver program – the Print Services Facility (PSF) for AFPDS output or the Composed Document Print Facility (CDPF) for 4250 output, or equivalent programs. Conversely, a page segment must be imbedded into a document by a formatting program such as SCRIPT/VS, which in turn produces a complete document for processing by the print-driver program.

More information about printing on VSE systems is given in the *GDDM System Customization and Administration* book.

In addition to user-written application programs, three GDDM utilities can run in VSE batch mode:

- The Image Print Utility
- The VSE Print Job Utility
- The Composite Document Print Utility

Instructions for running these are given in the *GDDM Base Application Programming Reference* book.

Link-editing

Before an application program can be run in VSE batch mode, it must be link-edited with two GDDM interface modules. One of these, ADMASLD, supports VSE batch mode. Here is some model job control language (JCL) for a link-edit job:

© Copyright IBM Corp. 1982, 1996

```
************************************************
* This JCL assumes that DLBL, EXTENT, and LIBDEF*
* statements have already been used to define   *
* the GDDM relocatable libraries                *
************************************************
*
// JOB jobname
// OPTION CATAL
   PHASE phase-name,*
   INCLUDE phase-name
*
* In the following INCLUDE statement,
* leave   ADMASNB unchanged for GDDM Base using
*                 nonreentrant interface
* replace ADMASNB by ADMASRB for GDDM Base using
*                 reentrant interface
*   or by ADMASP  if using the system programmer
*                 interface
   INCLUDE ADMASNB
   INCLUDE ADMASLD
// EXEC LNKEDT
/*
/&
```

Using the system programmer interface with dynamic load

If an application uses only the system programmer interface, all calls to GDDM are through the entry point ADMASP. This entry point can be resolved by link-editing the application with the GDDM interface modules ADMASP and ADMASLD, as described above.

Alternatively, the application can avoid these linkage-edit considerations by using system facilities (the VSE LOAD function) to dynamically load a GDDM interface module (ADMASPLD). The main entry point for this module is defined both with its load module name and with the name ADMASP.

Large 4250 page segments

A formatting program such as SCRIPT/VS can imbed a page segment in two ways: it can either include the complete segment inline, which means physically putting it into its output file; or include the name of the segment, leaving the printer driver program to physically imbed it in the final output.

The CDPF program limits the size of inline page segments to 40K bytes. If you have larger page segments, they cannot be passed to CDPF inline. Instead, they must be stored in a VSAM ESDS file, from where CDPF reads them when required. However, GDDM stores any page segments that it creates in a phase library, not in a VSAM file. To overcome this problem, there is a GDDM utility called ADMUP2VD that copies page segments from the phase library to a VSAM ESDS file.

ADMUP2VD should not be used in the shared virtual area (SVA).

Here is an example of JCL to copy a page segment from a phase library to a VSAM ESDS file:

```
* $$ JOB JNM=CPYPHASE,CLASS=0,DISP=D
* $$ LST CLASS=A,DISP=D,DEST=(node,userid),JSEP=1
// JOB CPYPHASE
// DLBL gddm,'gddm.library.name'
// EXTENT ,volid
// DLBL libname,'phase.library.name'
// EXTENT ,volid
// LIBDEF *,SEARCH=(libname.sublib,gddm.sublib)
// DLBL IJSYSUC,'user.catalog.name',,VSAM
// DLBL fname,'vsam.file.name',,VSAM
// EXEC IDCAMS,SIZE=AUTO
 DELETE (vsam.file.name)   -
        CLUSTER
/*
// EXEC IDCAMS,SIZE=AUTO
 DEFINE CLUSTER                      -
          (NAME(vsam.file.name)      -
           NONINDEXED                -
           RECORDFORMAT(V)           -
           RECORDSIZE(4000 8202)     -
           TRACKS(5 5)               -
           VOL(volid))               -
           DATA                      -
           ( NAME(data.file.name) )
/*
IF $RC>4 THEN
GOTO $EOJ
// EXEC ADMUP2VD,SIZE=ADMUP2VD,PARM='fname'
//*
//&
* $$ EOJ
```

Only the name of the phase to be copied must be specified on the `PARM='fname'` parameter (up to eight characters long). The type `PHASE` must not be included.

Spill files

GDDM uses spill files when creating output for page printers, unless told otherwise in a processing option. This is true whether the processing is done by a user-written application or a GDDM utility. The spill files need to be defined. An example of some JCL for doing this is shown below.

```
* $$ JOB JNM=DEFSPILL,CLASS=0,DISP=D
* $$ LST CLASS=A,DISP=D,DEST=(node,userid),   *
         JSEP=1
* $$ LST CLASS=A,DISP=D,LST=1A0,              *
         DEST=(node,userid),JSEP=1
// JOB DEFSPILL
// DLBL IJSYSUC,'user.catalog.name',,VSAM
// DLBL ADM0001,'ADM00001.SPILL.FILE',,VSAM
// DLBL ADM0002,'ADM00002.SPILL.FILE',,VSAM
// EXEC IDCAMS,SIZE=AUTO
 DELETE (ADM00001.SPILL.FILE)        -
         CLUSTER
 DEFINE CLUSTER                      -
        (NAME(ADM00001.SPILL.FILE)   -
         NONINDEXED                  -
         REUSE                       -
         RECORDSIZE(1000 2000)       -
         RECORDS(10 10)              -
         VOL(PAC371))                -
         DATA                        -
        (NAME(ADM00001.SPILL.DATA))
 DELETE (ADM00002.SPILL.FILE)        -
         CLUSTER
 DEFINE CLUSTER                      -
        (NAME(ADM00002.SPILL.FILE)   -
         NONINDEXED                  -
         REUSE                       -
         RECORDSIZE(1000 2000)       -
         RECORDS(10 10)              -
         VOL(PAC371))                -
         DATA                        -
        (NAME(ADM00002.SPILL.DATA))
/*
/&
* $$ EOJ
```

You must decide how you want to use spill files. Either one spill file can be deleted and defined in each print job (as shown above) or several can be defined before a print job is run.

If you define several spill files before the print job is run, use the NOALLOC option in the define statement to save space. Spill files that have not been emptied correctly (as a result of a previous job not ending cleanly) should be erased periodically.

Glossary

This glossary defines technical terms used in GDDM documentation. If you do not find the term you are looking for, refer to the index of the appropriate GDDM manual or view the *IBM Dictionary of Computing*, located on the Internet at:

`http:\\www.networking.ibm.com/nsg/nsgmain.htm`

A

AAB. Application anchor block.

ACB. Application control block.

active operator window. In GDDM, the operator window with the highest priority in the viewing order.

active partition. The partition containing the cursor. Contrast with *current partition*.

advanced function printing. The ability of licensed programs to use the all-points-addressable concept to print text and illustrations.

adjunct. In mapped alphanumerics, one of a set of optional subfields in an application data structure that specifies some attribute of a data field; for example, that it is highlighted. An adjunct enables the attribute to be varied at run time.

ADMGDF. See *graphics data format (GDF)*.

ADS. Application data structure.

AFPDS. Advanced-function presentation data stream.

AIC. Application interface component.

alphanumeric character attributes. In GDDM, the highlighting, color, and symbol set to be used for individual characters.

alphanumeric cursor. A physical indicator on a display. It can be moved from one hardware cell to another.

alphanumeric field. A field (area of a screen or printer page) that can contain alphabetic, numeric, or special characters. In GDDM, contrast with *graphics field*.

alphanumeric field attributes. In GDDM, the intensity, highlighting, color, and symbol set to be used for field type, field end, output conversion, input conversion, translate table assignment, transparency, field outlining, and mixed-string fields.

alphanumerics. Pertaining to alphanumeric fields. In GDDM there are three types of alphanumerics:

- Procedural alphanumerics
- Mapped alphanumerics
- High performance alphanumerics (HPA)

alternate device. In GDDM, a device to which copies of the primary device's output are sent. Usually the alternate device is a printer or plotter. See also *primary device*.

annotation. An added descriptive comment or explanatory note.

APA. All points addressable.

aperture. See *pick aperture*.

API. Application programming interface.

APL. One of the programming languages supported by GDDM.

application data structure (ADS). A structure created by GDDM-IMD that contains an entry for each variable field within a *map*. The data to be displayed in a mapped field is placed into the application data structure by the user's program.

application image. In GDDM, an image contained in GDDM main storage, and independent of any device or GDDM page. Contrast with *device image*.

application programming interface (API). The formally defined interface used by an application programmer to pass commands to, and get responses from, an IBM system control program or licensed program.

area. In GDDM, a shaded shape, such as a solid rectangle. It is created by opening the area, defining its outline, and closing the area.

aspect ratio. The width-to-height ratio of an area, symbol, or shape.

attention identifier. A number indicating which button the operator pressed to satisfy a read operation. For example, 0 (returned from GDDM to the application program) means that the operator pressed the Enter key.

attribute byte. The screen position that precedes an alphanumeric field on a 3270-family device and holds the attribute information. See also *trailing attribute byte*.

© Copyright IBM Corp. 1982, 1996

attributes. Characteristics or properties that can be controlled, usually to obtain a required appearance; for example, the color of a line. See also *alphanumeric character attributes*, *alphanumeric field attributes*, and *graphics attributes*.

axis. In a chart, a line that is drawn to indicate units of measurement against which items in the chart can be viewed.

A3. A paper size, more common in Europe than in the U.S. It measures 297mm by 420mm, and is twice the size of A4. See also *A4*.

A4. A paper size, more common in Europe than in the U.S. It measures 210mm by 297mm, and is half the size of A3. Compare with *quarto*. See also *A3*.

B

background color. Black on a display, white on a printer. The initial color of the display medium. Contrast with *neutral color*.

bar code. A code representing characters by sets of vertical parallel bars of varying thickness and separation that are read optically by transverse scanning.

BASIC. One of the programming languages supported by GDDM.

BDAM. Basic Direct Access Method.

bi-level image. An image in which each pixel is either black or white (value 0 or 1). Contrast with *gray-scale image* and *halftone image*.

BMS. Basic Mapping Support (CICS).

BPAM. Basic Partitioned Access Method.

business graphics. The methods and techniques for presenting commercial and administrative information in chart form; for example, the creation and display of a sales bar chart. Contrast with *general graphics*.

C

CALS. Continuous Acquisition and Life-Cycle Support.

CDPDS. Composite Document Presentation Data Stream.

CDPF. Composed Document Print Facility.

CDPU. Composite Document Print Utility.

CECP. Country-extended code page.

cell. See *character cell*.

CGM. Computer Graphics Metafile. A file that contains information about the content of a picture, and conforms to the International Standard, ISO 8632, or is of a similar format.

channel-attached. Pertaining to devices that are attached directly to a computer by means of data (I/O) channels. Synonymous with *local*. Contrast with *link-attached*.

character. A letter, digit, or other symbol.

character attributes. See *alphanumeric character attributes*. See also *graphics text attributes*.

character box. In GDDM, the rectangle or (for sheared characters) the parallelogram boundaries that govern the size, orientation, spacing, and italicizing of individual symbols or characters to be shown on a display screen or printer page.

The box width, height, and, if required, shear are specified in world coordinates and can be program-controlled. See also *character mode*. Contrast with *character cell*.

character cell. The physical, rectangular space in which any single character or symbol is displayed on a screen or printer device. The size and position of a character cell are fixed. Size is usually specified in pixels on a given device; for example, 9 by 12 on an &3279. Model 3 display. Position is addressed by row and column coordinates. Synonymous with *hardware cell* and *symbol cell*. Contrast with *character box*.

character code. The means of addressing a symbol in a symbol set, sometimes called *code point*.

The particular form and range of codes depends on the GDDM context. For example:

- For the Image Symbol Editor, a hexadecimal constant in the range `X'41'` through `X'FE'`, or its EBCDIC character equivalent
- For the Vector Symbol Editor, a hexadecimal constant in the range `X'00'` through `X'FF'`, or its EBCDIC character equivalent
- For the GDDM API, a decimal constant in the range 0 through 239, or subsets of this range (for example, a marker symbol code range of 1 through 8)

character grid. A notional grid that covers the *graphics field*. The size of the grid determines the basic size of the characters in all text constructed by presentation graphics routines. It is the fundamental measurement in chart layout, governing the spacing of mode-2 characters and the size of mode-3 characters. It also governs the size of the chart margins and thus the plotting area.

character matrix. Synonym for *dot matrix*.

character mode. In GDDM, the type of characters to be used. There are three modes:

- Mode-1 characters are loadable into PS and are of device-dependent fixed size, spacing, and orientation, as are hardware characters.
- Mode-2 characters are image (ISS) characters. Size and orientation are fixed. Spacing is variable by program.
- Mode-3 characters are vector (VSS) characters. Box size, position, spacing, orientation, and shear of individual characters are variable by program.

chart. In GDDM, usually means business chart; for example, a *bar chart*.

choice device. A logical input device that enables the application program to identify keys pressed by the terminal operator.

CICS. Customer Information Control System. A subsystem of MVS or VSE under which GDDM can be used.

clipping. In computer graphics, removing parts of a display image that lie outside a viewport. Synonymous with *scissoring*.

CMS. Conversational Monitor System. A time-sharing subsystem that runs under VM/SP.

COBOL. One of the programming languages supported by GDDM.

code page. Defines the relationship between a set of code points and graphic characters. This relationship covers both the standard alphanumeric characters and the national language variations. GDDM supports a set of code pages used with typographic fonts for the &4250. page printer.

code point. Synonym for *character code*.

Composite Document Presentation Data Stream (CDPDS). A data stream containing graphics, image, and text that is the input to the GDDM Composite Document Print Utility (CDPU).

Composed Document Print Facility (CDPF). An IBM licensed program for processing documents destined for the &4250. page printer.

composed-page image file. An intermediate form, residing on disk, of a picture destined for a page printer.

composed-page printer. See *page printer*.

composed-page printer format. A general term describing the format of print data destined for output by using either *CDPF* or *PSF*.

composite document. A document that contains both formatted text, such as that produced by the DCF program, and graphic or image data, such as that produced by GDDM. It is a combination of text and pictures on a page or set of pages. The pictures can be computer graphics or images created by scanning paper originals.

Composite Document Print Utility (CDPU). A utility that can print or display composite documents

compressed data stream. A data stream that has been made more compact by use of a data-compression algorithm.

constant data. In GDDM, data that is defined in a map and need not be known to the application program.

correlation. The translation (by GDDM) of a screen position into a part of the user's picture. This follows a *pick* operation.

country-extended code page (CECP). An extension of a normal EBCDIC code page that includes definitions of all code points in the range `X'41'` through `X'FE'`. Each code page contains the same 190 characters, but the mapping between code points and graphics characters depends on the country for which the code page is defined. This is a method of marking a GDDM object so that the environment in which it was created can be identified. It enables automatic translation to a different environment.

CSD. (1) Under MVS or VSE, CICS system definition. (2) In personal computer systems, Corrective Service Diskette; the means by which service is applied to the personal computer system.

current partition. The partition selected for processing by the application program. Contrast with *active partition*.

current position. In GDDM, the end of the previously drawn primitive. Unless a "move" is performed, this position is also the start of the next primitive.

cursor. A physical indicator that can be moved around a display screen. See *alphanumeric cursor* and *graphics cursor*.

CUT. Control unit terminal.

D

DASD. Direct access storage device.

data stream compatibility (DSC). In &8100. systems, the facility that provides access to System/370 applications that communicate with &3270. Information Display System terminals.

data stream compression. The shortening of an I/O data stream for the purpose of more efficient transmission between link-attached units.

data set. The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

DBCS. Double-byte character set.

DCF. Document Composition Facility.

DCSS. Discontiguous saved segment (VM/SP).

DCT. Destination control table (CICS).

default value. The value of an attribute chosen by GDDM when no value is explicitly specified by the user. For example, the default line type is a solid line. The default value is sometimes device-dependent. See also *drawing default* and *standard default*.

denibblized data. The decoded data stream used between the GDDM DOS Support feature in the host and GDDM-PCLK on the workstation.

designator character. The first byte of a light-pen-detectable field that indicates whether or not the field has been selected.

device echo. A visual identification of the position of the graphics cursor. The form of the device echo is defined by the application program.

device family. In GDDM, a device classification that governs the general way in which I/O will be processed. See also *processing option*. For example:

- Family 1: 3270 display or printer
- Family 2: queued printer
- Family 3: system printer (alphanumerics only)
- Family 4: page printer

device image. In GDDM, an image contained in a device or GDDM page. Contrast with *application image*.

device suffix. In GDDM-IMD, a suffix to a mapgroup name that indicates the device class.

device token. In GDDM, an 8-byte code giving entry to a table of pre-established device hardware characteristics that are required when the device is opened (initialized).

DIF. In GDDM terms, data interchange format.

digital image. A two-dimensional array of picture elements (pixels) representing a picture. A digital image can be stored and processed by a computer, using bits to represent pixels. In GDDM, pixels have the value black or white. Often called simply *image*.

direct transmission. In GDDM image processing, the transfer of image data direct from a source outside GDDM to an image device, including manipulation by a projection in the device, and without GDDM maintaining a copy or buffer of the data.

display device. Any output unit that gives a visual representation of data; for example, a screen or printer. More commonly, the term is used to mean a screen and not a printer.

display point. Synonym for *pixel*.

display-point matrix. Synonym for *dot matrix*.

display terminal. An input/output unit by which a user communicates with a data processing system or subsystem. It usually includes a keyboard and always provides a visual presentation of data. For example, an &3179. display.

DL/1. Data language 1. A language for database processing operations.

dot matrix. In computer graphics, a two-dimensional pattern of dots used for constructing a display image. This type of matrix can be used to represent characters by dots. Synonymous with *character matrix* and *display-point matrix*.

double-byte characters. See *double-byte character set (DBCS)*.

double-byte character set (DBCS). A set of characters in which each character occupies two byte positions in internal storage and in display buffers. Used for oriental languages; for example, *Kanji* or *Hangeul*. Contrast with *single-byte character set*.

DPCX. Distributed Processing Control Executive. An &8100. system control program.

DPPX. Distributed Processing Programming Executive. An &8100. system control program.

drawing default. The value of a graphics attribute chosen by GDDM when no value is explicitly specified

by the user. The drawing default may be altered by the user.

DSC. Data stream compatibility.

dual characters. See *double-byte characters*.

dummy device. An output destination for which GDDM does all the normal processing but for which no actual output is generated. Used, for example, to test programming for an unavailable output device.

E

EBCDIC. Extended binary coded decimal interchange code. A coded character set consisting of 8-bit coded characters.

echo. In interactive graphics, the visible form of the locator or other logical input device.

ECSA. Extended character set adapter.

edit. To enter, modify, or delete data.

editing grid. In the GDDM Image and Vector Symbol Editors, a grid used as a guide for editing a symbol. In the Image Symbol Editor, it is a dot matrix. In the Vector Symbol Editor, it is a grid of lines.

enterprise. An organization or company that undertakes local, national, or international business ventures.

extended data stream. For &3179., 3192, 3278, 3279, and 3287 devices, input/output data formatted and encoded in support of color, programmed symbols, and extended highlighting. These features extend the &3270. data stream architecture.

extended highlighting. The emphasizing of a displayed character's appearance by blinking, underscore, or reverse video.

external defaults. GDDM-supplied values that users can change to suit their own needs.

extracted image. In GDDM, an image on which transform element calls operate. It may imply the whole source image or just a part of it, depending on whether a define sub-image transform element has been applied in its derivation.

F

FCT. File control table (CICS).

field. An area on the screen or the printed or plotted page. See *alphanumeric field*, *graphics field*, and *mapped field*.

field attributes. See *alphanumeric field attributes*.

field list. The high performance alphanumerics data structure used to define alphanumeric fields.

fillet. A curve that is tangential to the end points of two adjoining lines.

flat file. A file that contains only data; that is, a file that is not part of a hierarchical data structure. A flat file can contain fixed-length or variable-length records.

floating area. The part of a page reserved for *floating maps*.

floating map. A map whose absolute position on the GDDM page is not fixed. During execution, a floating map takes the next available space that satisfies its specification.

floating-point feature. A processing unit feature that provides four 64-bit floating-point registers to perform floating-point arithmetic calculations.

foil. A transparency for overhead projection.

font. A particular style of typeface (for example, Gothic English). In GDDM, a font can exist as a programmed symbol set.

formatted document. A type of file containing text, images, and graphics.

FORTRAN. One of the programming languages supported by GDDM.

four-button cursor. A hand-held device, with cross-hair sight, used on the surface of a *tablet* to indicate position on a screen. Synonymous with *puck*.

frame. In GDDM-IMD, a synonym for *panel*.

full-screen alphanumeric operation. Full-screen processing operations on alphanumeric fields.

full-screen mode. A form of screen presentation in which the contents of an entire terminal screen can be displayed at once. Full-screen mode is often used for fill-the-blanks prompting, and is an alternative to line-by-line I/O.

full-screen processor. A host software component that, together with display terminal functions, supports display terminal input/output in full-screen mode.

G

GDDM. Graphical Data Display Manager. A series of IBM licensed programs, running in a host computer, that manage communications between application programs and display devices, printers, plotters, and scanners for graphics applications.

GDDM-GKS. GDDM Graphical Kernel System. A member of the GDDM family that runs under TSO and CMS and provides an alternative graphics programming interface to that of the GDDM base product. It is an implementation of the Graphical Kernel Standard, ISO 7942, of the International Organization for Standardization.

GDDM/graPHIGS. A member of the GDDM family used for creating hierarchical three-dimensional structures on the &5080gs.. It is based on the proposed ANSI standard for the Programmer's Hierarchical Interactive Graphics System (PHIGS).

GDDM Interactive Map Definition. GDDM-IMD. A member of the GDDM family of licensed programs. It enables users to create alphanumeric layouts at the terminal. The user defines the position of each field within the layout and may assign attributes, default data, and associated variable names to each field. The resultant map can be tested from within the utility.

GDDM-IVU. GDDM Interactive View Utility. A member of the GDDM family of licensed programs. It enables users to view, create, modify, store, and print images.

GDDM-OS/2. A licensed program that enables IBM PS/2 and other personal-computer systems with OS/2 installed to run GDDM application programs in the host computer.

GDDM-PCLK. A licensed program that enables IBM PS/2 and other personal computers with graphics-display adapters, and &3270. terminal emulators to run GDDM application programs in the host computer.

GDDM-PGF. GDDM-Presentation Graphics Facility. A member of the GDDM family of licensed programs. It is concerned with business graphics, rather than general graphics.

GDDM storage. The portion of host computer main storage used by GDDM.

GDF. Graphics data format.

general graphics. The methods and techniques for converting data to or from graphics display in mathematical, scientific, or engineering applications; that is, in any application other than business graphics. See also *business graphics*.

generated mapgroup. The output produced when a source GDDM-IMD mapgroup is generated. It contains the information needed by GDDM at execution to position the mapped fields on the GDDM page.

GIF. Graphics Interchange Format.

GKS. Graphical Kernel System. See *GDDM-GKS*.

GL. Graphical Language.

Graphical Data Display Manager. See *GDDM*.

graphics. A picture defined in terms of *graphics primitives* and *graphics attributes*.

graphics area. Part of a mapped field that is reserved for later insertion of graphics.

graphics attributes. In GDDM, color selection, color mix, line type, line width, graphics text attributes, marker symbol, and shading pattern definition.

graphics cursor. A physical indicator that can be moved (often with a joystick, mouse, or stylus) to any position on the screen.

graphics data format (GDF). A picture definition in an encoded order format used internally by GDDM and, optionally, providing the user with a lower-level programming interface than the GDDM API.

graphics data stream. The data stream that produces graphics on the screen, printer, or plotter.

graphics field. A rectangular area of a screen or printer page, used for graphics. Contrast with *alphanumeric field*.

graphics input queue. A queue associated with the graphics field onto which elements arrive from logical input devices. The program can remove elements from the queue by issuing a graphics read.

graphics primitive. A single item of drawn graphics, such as a line, arc, or graphics text string. See also *graphics segment*.

graphics read. A form of read that solicits graphics input or removes existing elements from the graphics input queue.

graphics segment. A group of graphics primitives (lines, arcs, and text) that have a common window and a common viewport and associated attributes. Graphics

segments allow a group of primitives to be subject to various operations. See also *graphics primitive*.

graphics text attributes. In GDDM, the symbol (character) set to be used, character box size, character angle, character mode, character shear angle, and character direction.

graPHIGS. See *GDDM/graPHIGS*.

gray-level. A digitally encoded shade of gray, normally (and always in GDDM) in the range 0 through 255. See also *gray-scale image*.

gray-scale image. An image in which the gradations between black and white are represented by discrete gray-levels. Contrast with *bi-level image* and *halftone image*.

green lightning. The name given to the flashing streaks on an &3270. screen while a programmable symbol set is being loaded.

H

halftone image. A bi-level image in which intermediate shades of gray are simulated by patterns of adjacent black and white pixels. Contrast with *gray-scale image*.

Hangeul. A character set of symbols used in Korean ideographic alphabets.

hardware cell. Synonym for *character cell*.

hardware characters. Synonym for *hardware symbols*.

hardware symbols. The characters that are supplied with the device. The term is loosely used also for GDDM mode-1 symbols that are loaded into a PS store for subsequent display. Synonymous with *hardware characters*.

hexadecimal. Pertaining to a numbering system with base sixteen.

host. See *host computer*.

high performance alphanumerics. The creation of alphanumeric displays using field list data structures. Contrast with *procedural* and *mapped* alphanumerics.

host computer. The primary or controlling computer in a multiple-computer installation.

I

ICU. Interactive Chart Utility.

identity projection. In GDDM image processing, a projection that is transferred from source image to target image without any processing being performed on it.

image. Synonym for *digital image*.

image data stream. The internal form of the GDDM data in an image environment.

image field. A rectangular area of a screen or printer page, used for image. Contrast with *alphanumeric field* and *graphics field*.

Image Object Content Architecture (IOCA). An architected collection of constructs used to interchange and present images.

image symbol. A character or symbol defined as a dot pattern.

Image Symbol Editor (ISE). A GDDM-supplied interactive editor that enables users to create or modify their own image symbol sets (ISS).

image symbol set (ISS). A set of symbols each of which was created as a pattern of dots. Contrast with *vector symbol set (VSS)*.

IMD. See *GDDM Interactive Map definition*.

IMS/VS. Information Management System/Virtual Storage. A subsystem of MVS under which GDDM can be used.

include member. A collection of source statements stored as a library member for later inclusion in a compilation.

input queue. See *graphics input queue*.

integer. A whole number (for example, –2, 3, 457).

Intelligent Printer Data Stream (IPDS). A structured-field data stream for managing and controlling printer processes, allowing both data and controls to be sent to the printer. GDDM uses IPDS to communicate with the IBM 4224 printer.

Interactive Chart Utility (ICU). A GDDM-PGF menu-driven program that allows business charts to be created interactively by nonprogrammers.

interactive graphics. In GDDM, those graphics that can be moved or manipulated by a user at a terminal.

Interactive Map definition. A member of the GDDM family of licensed programs. It enables users to create alphanumeric layouts at the terminal. The operator defines the position of each field within the layout and may assign attributes, default data, and associated variable names to each field. The resultant map can be tested from within the utility.

interactive mode. A mode of application operation in which each entry receives a response from a system or program, as in an inquiry system or an airline reservation system. An interactive system can also be conversational, implying a continuous dialog between the user and the system.

interactive subsystem. (1) One or more terminals, printers, and any associated local controllers capable of operation in interactive mode. (2) One or more system programs or program products that enable user applications to operate in interactive mode; for example, CICS.

intercept. In a chart, a method of describing the position of one axis relative to another. For example, the x axis can be specified so that it intercepts (crosses) the y axis at the bottom, middle, or top of the plotting area of a chart.

inter-device copy. The ability to copy a page or the graphics field from the current primary device to another device. The target device is known as the alternate device.

IOCA. See *Image Object content Architecture*.

IPDS. See *Intelligent Printer Data Stream*.

ISE. Image Symbol Editor.

ISO. International Organization for Standardization.

ISPF. Interactive System Productivity Facility.

ISS. Image symbol set.

IVU. Image View Utility. See *GDDM-IVU*.

J

joystick. A lever that can pivot in all directions in a horizontal plane, used as a *locator* device.

K

Kanji. A character set of symbols used in Japanese ideographic alphabets.

Katakana. A character set of symbols used in one of the two common Japanese phonetic alphabets; Katakana is used primarily to write foreign words phonetically. See also *Kanji*.

key. In a legend, a symbol and an associated data group name. A key might, for example, indicate that the blue line on a graph represents “Predicted Profit.” See also *legend*.

key symbol. A small part of a line (from a line graph) or an area (from a shaded chart) used in a legend to identify one of the various data groups.

L

Latin. Of or pertaining to the Western alphabet. In GDDM, a synonym for *single-byte character set*.

legend. A set of symbolic keys used to identify the data groups in a business chart.

line attributes. In GDDM, color, line type, and line width.

link pack area. An MVS term that describes an area of shared storage.

link-attached. Pertaining to devices that are connected to a controlling unit by a data link. Synonymous with *remote*. Contrast with *channel-attached*.

local. Synonym for *channel-attached*.

local character set identifier. A hexadecimal value stored with a GDDM symbol set, which can be used by symbol-set-loading means other than GDDM in the context of local copy on a printer.

locator. A logical input device used to indicate a position on the screen. Its physical form may be the alphanumeric cursor or a graphics cursor moved by a joystick.

logical input device. A concept that allows application programs to be written in a device-independent manner. The logical input devices to which the program refers may be subsequently associated with different physical parts of a terminal, depending on which device is used at run time.

LPA. Link pack area.

LTERM. In IMS/VS, logical terminal.

M

map. A predefined format of alphanumeric fields on a screen. Usually constructed outside of the application program.

map specification library (MSL). The data set in which maps are held in their source form.

mapgroup. A data item that contains a number of maps and information about the device on which those maps are to be used. All maps on a GDDM page must come from the same mapgroup.

mapped alphanumerics. The creation of alphanumeric displays using predefined maps. Contrast with *procedural alphanumerics* and *high performance alphanumerics*.

mapped field. An area of a page whose layout is defined by a map.

mapped graphics. Graphics placed in a graphics area within a mapped field.

mapped page. A GDDM page whose content is defined by maps in a mapgroup.

mapping. The use of a map to produce a panel from an output record, or an input record from a panel.

marker. In GDDM, a symbol centered on a point. Line graphs and polar charts can use markers to indicate the plotted points.

MDT. Modified data tag.

menu. A displayed list of logically grouped functions from which the user can make a selection. Sometimes called a menu panel.

menu-driven. Describes a program that is driven by user response to one or more displayed menus.

MFS. Message format service.

MICR. Magnetic ink character recognition.

mixed character string. A string containing a mixture of *Latin* (one-byte) and *Kanji* or *Hangeul* (two-byte) characters.

Mixed Object Document Content Architecture (MO:DCA). An architected, device-independent data stream for interchanging documents.

mode-1/-2/-3 characters. See *character mode*.

mountain shading. A method of shading surface charts where each component is shaded separately from the base line, instead of being shaded from the data line of the previous component.

mouse. A device that a user moves on a flat surface to position a pointer on a screen.

MSHP. Maintain System History Program. A software process for installing licensed programs on VSE systems.

MSL. Map specification library.

MVS. IBM Multiple Virtual Storage. A system under which GDDM can be used.

MVS/XA. Multiple Virtual Storage/Extended Architecture. A subsystem under which GDDM can be used.

N

name-list. A means of identifying which physical device is to be opened by a GDDM program. It can be used as a parameter of the DSOPEN call, or in a *nickname*.

National Language Support (NLS). A special feature that provides translations of the ICU panels and some of the GDDM messages into a variety of languages, including US English.

negate. In bi-level image data, setting zero bits to one and one bits to zero.

neutral color. White on a display, black on a printer. Contrast with *background color*.

nibblized data. The encoded data stream used between the GDDM DOS Support feature in the host and GDDM-PCLK on the workstation.

nickname. In GDDM, a means of referring to a device, the characteristics and identity of which have been already defined.

NLS. National Language Support.

nonqueriable printer. A printer about which GDDM cannot obtain any information.

NSS. Named saved system (VM/XA and VM/ESA).

null character. An empty character represented by `X'00'` in the EBCDIC code. Such a character does not occupy a screen position.

O

operator reply mode. In GDDM, the mode of interaction available to the operator (display terminal user) with respect to the modification (or not) of alphanumeric character attributes for an input field.

operator window. Part of the display screen's surface on which the GDDM output of an application program can be shown. An operator window is controlled by the end user; contrast with *partition*. A *task manager* may create a window for each application program it is running.

outbound structured field. An element in &3270. data streams from host to terminal with formatting that allows variable-length and multiple-field data to be sequentially translated by the receiver into its component fields without the receiver having to examine every byte.

P

page. In GDDM, the main unit of output and input. All specified alphanumerics and graphics are added to the current page. An output statement always sends the current page to the device, and an input statement always receives the current page from the device.

page printer. A printer, such as the &3820. or &4250., to which the host computer sends data in the form of a succession of formatted pages. Such devices can print pictorial data and text, and can position all output to pixel accuracy. The pixel density and the general print quality both often suffice as camera-ready copy for publications. Also known as *composed-page printer*.

page segment. A picture file in a form that can be printed. It can only be printed if it is embedded in a primary document. Also known as a *PSEG*o file.

panel. A predefined display that defines the locations and characteristics of alphanumeric fields on a display terminal. When the panel offers the operator a selection of alternatives it may be called a menu panel. Synonymous with *frame*.

partition. Part of the display screen's surface on which a page, or part of a page, of GDDM output can be shown. Two or more partitions can be created, each displaying a page, or part of a page, of output. A partition is controlled by the GDDM application; contrast with *operator window*.

partition set. A grouping of partitions that are intended for simultaneous display on a screen.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

PCB. In GDDM, program communication block (IMS/VS).

PCLK. See *GDDM-PCLK*.

PDS. Partitioned data set (MVS).

pel. Picture element. See *pixel*.

PGF. Presentation Graphics Facility. A member of the GDDM family of licensed programs. It is concerned with business graphics, rather than general graphics.

PHIGS. Programmer's Hierarchical Interactive Graphics System.

pick. The action of the operator in selecting part of a graphics display by placing the graphics cursor over it.

pick aperture. A rectangular or square box that is moved across the screen by the graphics cursor. An item must lie at least partially within the pick aperture before it can be picked.

pick device. A logical input device that allows the application to determine which part of the picture was selected (or picked) by the operator.

picture interchange format (PIF) file. In graphics systems, the type of file, containing picture data, that can be transferred between GDDM and an &3270pcg., /GX, or /AT workstation.

picture space. In GDDM, an area of specified aspect ratio that lies within the graphics field. It is centered on the graphics field and defines the part of the graphics field in which graphics will be drawn.

PIF. Picture interchange format.

pixel. The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonymous with *display point*, *pel*, and *picture element*.

PL/I. One of the programming languages supported by GDDM.

plotter. An output device that uses pens to draw its output on paper or transparency foils.

pointings. Pairs of x-y coordinates produced by an operator defining positions on a screen with a locator device, such as a *mouse*.

polar chart. A form of business chart where the x axis is circular and the y axis is radial.

polyfillet. In GDDM, a curve based on a sequence of lines. It is tangential to the end points of the first and last lines, and tangential also to the midpoints of all other lines.

polyline. A sequence of adjoining lines.

popping. A method of ordering data whereby each item in a list or sequence takes the value of the previous item in the list or sequence, and is then removed from the list; when this happens, the list or sequence of data is said to be “popped.”

ppi. Pixels per inch.

PQE. Printer queue element.

presentation graphics. Computer graphics products or systems, the functions of which are primarily concerned with graphics output presentation. For example, the display of business planning bar charts.

preview chart. A small version of the current chart that can be displayed on ICU menu panels.

primary device. In GDDM, the main destination device for the application program’s output, usually a display terminal. The default primary device is the user console. See also *alternate device*.

primitive. See *graphics primitive*.

primitive attribute. A specifiable characteristic of a graphics primitive. See *graphics attributes* and *graphics text attributes*.

Print Services Facility (PSF). An IBM licensed program for processing documents destined for the &3800m3. page printer.

print utility. A subsystem-dependent utility that sends print files from various origins to a queued printer.

procedural alphanumerics. The creation of alphanumeric displays using the GDDM alphanumeric API. Contrast with *mapped alphanumerics* and *high performance alphanumerics*.

processing option. Describes how a device’s I/O is to be processed. It is a device-family-dependent and subsystem-dependent option that is specified when the device is opened (initialized). An example is the choice between CMS attention-handling protocols.

procopt. Processing option.

profile. In GDDM, a file that contains information about how GDDM is to process requests for services to devices or other functions.

program library. (1) A collection of available computer programs and routines. (2) An organized collection of computer programs.

programmed symbols (PS). Dot patterns loaded by GDDM into the PS stores of an output device.

projection. In GDDM image processing, an application-defined function that specifies operations to be performed on data extracted from a source image. Consists of one or more *transforms*. See also *transform element*.

PS. Programmed symbols.

PS overflow. A condition where the graphics cannot be displayed in its entirety because the picture is too complex to be contained in the device’s PS stores.

PSB. Program specification block (IMS).

PSEG. See *page segment*.

PSF. Print Services Facility.

PSP bucket. A database containing descriptions of faults found in programs. Used by Service personnel.

PS/2. Personal System/2.

puck. Synonym for *four-button cursor*.

PUT. Program update tape.

Q

quarto. A paper size, more common in the U.S. than in Europe. It measures 8.5 inches by 11.0 inches. Also known as A size. Compare with *A4*.

queued printer. A printer belonging to the subsystem under which GDDM runs, to which output is sent indirectly by means of the GDDM Print Utility program. In some subsystems, this may allow the printer to be shared between multiple users. Contrast with *system printer*.

R

raster device. A device with a display area consisting of dots. Contrast with *vector device*.

rastering. The transforming of graphics primitives into a dot pattern for line-by-line sequential use. In GDDM PS devices, this is done by transforming the primitives into a series of programmed symbols (PS).

real device. A GDDM device that is not being windowed by means of operator window functions. Contrast with *virtual device*.

reentrant. The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

remote. Synonym for *link-attached*.

reply mode. See *operator reply mode*.

resolution. In graphics and image processing, the number of pixels per unit of measure (inch or meter).

reverse clipping. Where one graphics primitive overlaps another, removing any parts of the underlying primitive that are overpainted by the overlying primitive.

reverse video. A form of alphanumeric highlighting for a character, field, or cursor, in which its color is exchanged with that of its background. For example, changing a red character on a black background to a black character on a red background.

REXX. Restructured Extended Executor Language. One of the programming languages supported by GDDM.

Roman. Relating to the *Latin* type style, with upright characters.

S

SBCS. Single-byte character set.

scanner. A device that produces a digital image from a document.

scissoring. Synonym for *clipping*.

scrolling. In computer graphics, moving a display image vertically or horizontally in a manner such that new data appears at one edge as existing data disappears at the opposite edge.

SCS. SNA character string.

segment. See *graphics segment*.

segment attributes. Attributes that apply to the segment as an entity, rather than to the individual primitives within the segment. For example, the visibility, transformability, or detectability of a segment.

segment library. The portion of auxiliary storage where segment definitions are held. These definitions are GDDM objects in graphics data format (GDF) and are managed by GDDM API calls. GDDM handles the file accesses to and from auxiliary storage.

segment priority. The order in which segments are drawn; also the order in which they are detected.

segment transform. The means to rotate, scale, and reposition segments without re-creating them.

selector adjunct. A subfield of an application data structure that qualifies a data field.

shear. The action of tilting graphics text so that each character leans to the left or right while retaining a horizontal baseline.

single-byte character set (SBCS). A set of characters in which each character occupies one byte position in internal storage and in display buffers. Used for example, in most non-Oriental symbols. Contrast with *double-byte character set.*

SMP/E. System Modification Program/Extended. A software process for installing licensed programs on MVS systems.

SNA. System Network Architecture.

source image. An image that is the data input to image processing or transfer.

spill file. A means of reducing storage requirements at the cost of processing time, when creating high-resolution output files for page printers, for example.

stand-alone (mode). Operation that is independent of another device, program, or system.

standard default. The value of a graphics attribute chosen by GDDM when no value is explicitly specified by the user. The standard default cannot be altered by the user, although it may be overridden by the user.

string device. A logical input device that enables an application program to process character data entered by the terminal operator.

stroke device. A logical input device that enables an application program to process a sequence of x,y coordinate data entered by the terminal operator.

stylus. A pen-like pointer used on the surface of a tablet to indicate position on a screen.

surface chart. A chart similar to a line graph, except that no markers appear and the areas between successive lines are shaded.

swathe. A horizontal slice of printer output, forming part of a complete picture. Page printer images are often constructed in swathes to reduce the amount of storage required.

symbol. Synonymous with *character*. For example, the following terms all have the same meaning: vector symbols, vector characters, vector text.

symbol cell. Synonym for *character cell*.

symbol matrix. Synonym for *dot matrix*.

symbol set. A collection of symbols, usually but not necessarily forming a font. GDDM applications may use the hardware device's own symbol set. Alternatively, they can use image or vector symbol sets that the user has created.

symbol set identifier. In GDDM, an integer (or the equivalent EBCDIC character) by which the programmer refers to a loaded symbol set.

system printer. A printer belonging to the subsystem under which GDDM runs, to which output is sent indirectly by use of system spooling facilities. Contrast with *queued printer*.

T

tablet. (1) A locator device with a flat surface and a mechanism that converts indicated positions on the surface into coordinate data. (2) The IBM 5083 Tablet Model 2, which, with a four-button cursor or stylus, allows positions on the screen to be addressed and the graphics cursor to be moved without use of the keyboard.

tag. In interactive graphics, an identifier associated with one or more primitives that is returned to the program if such primitives are subsequently picked.

target image. An image that is the destination of processed or transferred data.

target position. In the GDDM Vector Symbol Editor, the grid coordinates of a point on the editing grid to which a vector is to be drawn.

task manager. A program that supervises the concurrent running of other programs.

temporary graphics. Graphics created outside a segment.

terminal. A device, usually equipped with a keyboard and a display unit, capable of sending and receiving information over a link. See also *display terminal*.

terminal emulator. A program that enables a device such as a personal computer system to enter and receive data from a host computer system as if it were a particular type of attached terminal.

test symbol. In the GDDM Image and Vector Symbol Editors, an area on the Symbol Edit panel in which the currently chosen symbol is displayed.

text. Characters or symbols sent to the device. GDDM provides alphanumeric text and graphics text.

text attributes. See *graphics text attributes*.

tilted pie chart. A pie chart drawn in three dimensions, which has been tilted away from full face to reveal its three-dimensional properties.

trailing attribute byte. The screen position following an alphanumeric field. This attribute byte can specify, for example, that the cursor should auto-skip to the next field when the current field is filled.

transfer operation. In GDDM image processing, an operation in which a projection is applied to a source image, and the result placed in a target image. The source and target images can be device or application images in any combination, or one or other of them (but not both) can be image data within the application program.

transform. (1) The action of modifying a picture for display; for example, by scaling, rotating, or displacing. (2) The object that performs or defines such a modification; also referred to as a *transformation*. (3) In GDDM image processing, a definition of three aspects of the data manipulation to be done by a projection:

1. A transform element or sequence of transform elements
2. A resolution conversion or scaling algorithm
3. A location within the target image for the result

Only the third item is mandatory.

See also *projection* and *transform element*.

transform element. In GDDM image processing, a specific function in a transform, which can be one of the following: define sub-image, scale, orient, reflect, negate, define place in target image.

A given transform element can be used only once in a *transform*.

transformable. A segment must be defined as transformable if it will subsequently be moved, scaled, or rotated.

transparency. (1) A document on transparent material suitable for overhead projection. (2) An alphanumeric attribute that allows underlying graphics or image to show.

TSO. Time Sharing Option. A subsystem of MVS under which GDDM can be used.

TWA. Transaction work area.

U

UDS. User default specification.

UDSL. A list of user default specifications (UDSs).

unformatted data. In GDDM image processing, compressed or uncompressed binary image data that has no headers, trailers, or embedded control fields other than any defined by the compression algorithm, if applicable. The data is in row major order, beginning with the top left of the picture.

User Control. A GDDM function that enables the terminal or workstation to perform some functions without the need for application programming. The actions include: moving and zooming graphics; manipulating windows; printing, plotting, and saving pictures.

user default specification (UDS). The means of changing a GDDM external default value. The external default values that a UDS can change are those of the GDDM or subsystem environment, GDDM user exits, and device definitions.

user exit. A point in GDDM execution where a user routine will gain control if such has been requested.

V

variable cell size. In most devices, the hardware cell size is fixed, but the &3290. Information Panel has a cell size that can be varied. This, in turn, causes the number of rows or columns on the device to alter.

vector. (1) In computer graphics, a directed line segment. (2) In the GDDM-PGF Vector Symbol Editor, a straight line between two points.

vector device. A device capable of displaying lines and curves directly. Contrast with *raster device*.

vector symbol. A character or shape composed of a series of lines or curves.

Vector Symbol Editor. A program supplied with GDDM-PGF, the function of which is to create and edit vector symbol sets (VSS).

vector symbol set (VSS). A set of symbols, each of which was originally created as a series of lines and curves.

Venetian blind effect. The name given to the appearance of bars across shaded patterns on an &3270pc. when GDDM tries to match the image symbol sets.

Venn diagram. A form of business chart in which, in GDDM, two or more populations and their intersection are represented by overlapping circles.

viewport. A subdivision of the picture space, most often used when two separate pictures are to be displayed together.

virtual device. A GDDM device that is being windowed by use of operator window functions. Contrast with *real device*.

virtual screen. The presentation space viewed through an *operator window*.

VM/ESA. IBM Virtual Machine Enterprise Systems Architecture.

VM/SP CMS. IBM Virtual Machine/System Product Conversational Monitor System; a system under which GDDM can be used.

VMXA. IBM Virtual Machine Extended Architecture; a system under which GDDM can be used.

VSE. Virtual storage extended; an operating system consisting of VSE/Advanced Functions and other IBM programs.

Note: In GDDM, the abbreviation VSE has sometimes been used to refer to the Vector Symbol Editor, but to avoid confusion, this usage is deprecated.

VSS. Vector symbol set.

W

Ward. One of the 190 matrices used to contain the symbols of a double-byte character set. The value in the first byte of each double-byte character code refers to the ward in which the character is contained. The value in the second byte denotes the character's position in the matrix.

window. In GDDM, the term window has three distinct meanings:

1. The "graphics window" is the coordinate space used for defining the primitives that make up a graphics picture. By default, both x and y coordinates run from 0 through 100. The graphics window can be regarded as a set of coordinates that are overlaid on the viewport.
2. An "operator window" is an independent rectangular subdivision of the screen. Several can exist at the same time, and each can receive output from, and send input to, either a separate GDDM program or a separate function of a single GDDM program.

3. The “page window” defines which part should be displayed of a page that is deeper or wider than its partition.

workstation. A display screen together with attachments such as a local copy device or a tablet.

world coordinates. The user application-oriented coordinates used for drawing graphics. See also *window*.

wrap-around field. An alphanumeric field that extends to the right-hand edge of the page and continues at the start of the next row.

WTP. Write-to-programmer.

Index

Special Characters

¢ sign 242
$ sign 242

Numerics

1403 system printer 404
16M, GDDM code above this location 550, 556
3117 scanner
  brightness control call 340
  contrast control call 341
  image conversion algorithms 342
  input image width restriction 360
  ISLDE call, effect of 89
3118 scanner
  brightness control call 340
  contrast control call 341
  document loading 89
  image conversion algorithms 342
  input image width restriction 360
  introduction to 85
  programming for 87
  resolutions 89
3193 Display Station 368
  introduction to 85
  local operations 350
  multiple extract restrictions 361
  multiple placing restrictions 361
  programming for 87
  rectangle placing restriction 362
  resolution 90
  scaling factors restriction 361
3211 system printer 404
3270 hardware attributes 74
3270 terminals
  plotters 433
3270-PC/G and /GX
  graphics attributes 49
  graphics text 69
  processing options
    retained and nonretained modes 224
  retained and nonretained modes 224
  supported colors 49
  symbol sets 255
  underpaint mode not supported 49
3270-PC/G, and /GX
  GDF files 192
  PIF files 192
  transferring PIF and GDF files 192
3270-PC/GX
  alphanumerics and graphics on two screens 284, 333
3270-PC/GX *(continued)*
  dual-screen configuration 71
  mapping 284, 333
3279 displays
  graphics text 69
3290 information panel
  partitions 453
  programming example 454, 477
  scrolling 473
  variable cell size 474
3800 page printer
  graphics-text output 70
3800 printer 404
  model 1 404
  system printer
    loadable symbol sets 423
3812 printer 404
3816 printer 404
3820 printer 404
3825 printer 404
4028 printer 404
  printing an image on 366
4224 printer 369, 404
  introduction to 85
4234 printer 404
4250 printer 404
  alphanumerics not supported 54
  graphics-text output 70
  page segments, large, under VSE 596
  typographic fonts 424
  using symbol sets 256
5080 graphics system
  alphanumerics 82
  alphanumerics and graphics on two screens 71, 284, 333
  graphics text 69
  mapping 284, 333
5550 Multistation 265
6090 graphics system
  alphanumerics and graphics on two screens 71, 284
  graphics text 69
  mapping 284
64-color pattern set 39
737x plotters 433

A

AAB (application anchor block) 4
activate (open) a device 382
activate stroke device 205

© Copyright IBM Corp. 1982, 1996

active operator window 480, 484, 490
active partition 459
Address command 16
Address gddm 16
Address link 16
adjunct
  See alphanumerics, mapped
ADM4CDUx 419
ADMASLD 595
ADMASNB 595
ADMASNO and ADMASRO interface modules 585
ADMASP 595
ADMASRB 595
ADMCDATA files
  code-page conversion 252
ADMCDEFM files
  code-page conversion 252
ADMCFORM files
  code-page conversion 252
ADMCGM
  TSO 542
  VM/CMS 532
ADMCOLn
  TSO 542
  VM/CMS 532
ADMCOLn files 430
ADMCOLSD 39
ADMCOLSN 39
ADMCOLSR 39
ADMDATRN 248
ADMDECK
  TSO 542
  VM/CMS 532
ADMDEFS
  TSO 542
  VM/CMS 532
ADMDHIPK symbol set 428
ADMDVECP 248
ADMDVSSx 248
ADMDxxxx (CICS termid) 580
ADMG transient data queue 294
ADMGDF
  CICS 580
  TSO 542
  VM/CMS 532
ADMGDF files 173
ADMGGMAP
  CICS 580
  TSO 542
  VM/CMS 532
ADMGGMAP ddname 294
ADMGGMAP FCT name 294
ADMGGMAP filetype 293
ADMGNADS ddname 294
ADMIMAGE
  TSO 542
ADMIMAGE *(continued)*
  VM/CMS 532
ADMIMAGE files 406
ADMLIST
  TSO 542
  VM/CMS 532
ADMLISTCDP
  VM/CMS 532
ADMMCOLT macro 429
ADMMNICK statement 375
ADMnnnnn color tables 429
ADMOPRT sequential file print program 421
ADMOPUC (CICS print utility) 582
ADMOPUV CMS graphics print utility 418
ADMPATTC 38
ADMPLOT plotter name 433
ADMPRINT
  files 252
  VM/CMS 532
ADMPRINT files 402
ADMPRINTQ 542
ADMQPOST EXEC procedure 418
ADMSAVE
  CICS 580
  TSO 542
  VM/CMS 532
ADMSYMBL 532
  CICS 580
  TSO 542
  VM/CMS 532
ADMTRACE
  IMS 559
  TSO 542
  VM/CMS 532
ADMUAIMC 317
ADMUCIMC 317
ADMUFO, user fast option
  bypass parameter checking 140
    restriction 141, 540, 557
ADMUOFF (CSECT for nonreentrant CICS) 577
ADMUP2VD 596
ADMUPIMC 317
ADMUSB1 C/370 sample program 519, 520
ADMUSB2 C/370 sample program 519, 520
ADMUSB3 C/370 sample program 519
ADMUSC1 COBOL sample program 519, 520
ADMUSC2 COBOL sample program 519, 520
ADMUSF1 FORTRAN sample program 519, 520
ADMUSF2 FORTRAN sample program 519, 520
ADMUSP1 PL/I sample program 519, 520
ADMUSP1I PL/I sample program (for IMS) 520
ADMUSP2 PL/I sample program 519, 520
ADMUSP2I PL/I sample program (for IMS) 520
ADMUSP3 PL/I sample program 519
ADMUSP3 sample program 520

ADMUSP4 PL/I sample program 519
ADMUSP4 sample program 521
ADMUSP7 sample program 252
ADMUTMAT 522
ADMUTMAV 522
ADMUTMCT 522
ADMUTMCV 522
ADMUTMDT 522
ADMUTMDV 522
ADMUTMIT 522
ADMUTMIV 522
ADMUTMPT 522
ADMUTMPV 522
ADMUTMST 522
ADMUTMSV 522
ADMUTMT PL/I sample program (for TSO) 519
ADMUTMT sample program 521
  compiling 522
  ending 523
  interaction with User Control 523
  link-editing 522
  running 522
  running your own programs 523
  using 523
ADMUTMTT 522
ADMUTMTV 522
ADMUTMV PL/I sample program (for CMS) 519
ADMUTMV sample program 521
  compiling 522
  link-editing 522
  running 523
ADS (application data structure) 283
  See also alphanumerics, mapped
Advanced Function Presentation Data Stream 400
advanced-function printers
  See also page printers
  graphics-text output 70
  using symbol sets 256
advanced-function printing
  mixed-object output 407
  print file
    GOCA 407
    IOCA 407
    PTOCA 407
  secondary data stream
    overlay 409
    page segment 409
AFPDS 400
AFTCxxxx code pages 425
AFTxxxxx fonts 424
AID translation 326
alarm
  mapped output 315
  procedural call 80
alphanumeric defaults module
  translation tables 248
alphanumeric text
  selecting symbols for 53
alphanumerics
  comparison of the different types 56
  device considerations 396
  procedural
    processing modified fields 261
alphanumerics taking precedence over graphics 61
alphanumerics,
  high-performance 273
alphanumerics, high-performance
  support on plotters 433
alphanumerics, introduction to 54
alphanumerics, mapped 283
  adjunct 307
    base attribute 315
    color 327
    cursor 318
    extended highlighting 327
    length 321
    on input 310, 311
    on output 307, 311
    selector 307, 311, 321
    symbol set 327
  AID translation 326
  alarm 315
  and graphics
    programming example 333
  application data structure (ADS) 283, 285
    adjuncts 307
    conversion for C/370 288
    conversion for GDDM-REXX 284
    creating 293, 294
    receiving data 289
    transmitting data 289
  character attributes 330
  compared with high-performance alphanumerics 56
  compared with procedural alphanumerics 56
  constant data fields 283
  copying between devices 422
  cursor position 318
  CURSR SEL key 321
  default data 293, 307
  examples
    AID translation 327
    color adjunct 327
    cursor adjunct 319, 321
    cursor menu selection 322
    floating maps 301
    graphics and mapping 333
    light pen menu selection 322
    multiple fixed maps 297
    PF key selection from menu 322
    selector adjunct 308
    selector and cursor adjuncts 322
    simple program using ASREAD 291
    simple program using MSREAD 285, 291

alphanumerics, mapped *(continued)*
field attributes 315
blinking 327
color 327
data type 317
defining and testing 293
highlight 316, 327
intensity 316
light pen 317
MDT bit 317
non-display 316
protected, unprotected, and autoskip 315
reverse video 327
symbol set 327
unprotected field changed to protected 305
field naming 293
floating area 299
folding input 332
generating mapgroup 293
graphics and mapping 332
initial data 293
Interactive Map Definition product (GDDM-IMD) 283, 285
overview of operations 291
quick-path tutorial 285
introduction 54, 55
justifying input 332
light pen 321
designator character 321
detectable attribute 317
mapgroup 289, 304
device suffixes 304
maps 283
cursor receiver 321
fixed 296
floating 296, 299
graphic area 332
multiple 296
positioning on page 289, 296, 297, 299
testing 293, 297
mixed with procedural alphanumerics 289
MSDFLD (create a mapped field) 289
MSGET (retrieve data from a map) 289
adjuncts 310
character attributes 330, 332
setting adjuncts 311
MSPCRT (create a page for mapping) 289
MSPUT (place data into a mapped field) 289
alarm and keyboard locking 315
base attribute adjunct 316
reject 311
reject operations 311
selector adjunct 310
write and rewrite 310, 311
MSQMOD (query modified fields) 302
MSREAD (present mapped data) 286

alphanumerics, mapped *(continued)*
null characters 321
query calls 306
support on plotters 433
variable data
receiving 289
transmitting 289
variable data fields 283
alphanumerics, procedural 71, 260
See also alphanumerics, mapped
See also graphics text
and graphics 54
ASFBDY - define field outline 270
ASFTRA - define field-transparency attribute 82
ASFTRN - set translation-tables attribute 75
attributes 74
on printer 413
auto-skip fields 74, 75
blinking fields 75
character attributes 76
color 76
highlight 77
input of 77, 242
symbol set 239
compared with high-performance alphanumerics 56
compared with mapped alphanumerics 56
copying between devices 422
field attributes 74, 76, 264
blank-to-null 75
color 75
DBCS (double-byte character set) 266
double-byte characters 266
field end 75
Hangeul 266
highlight 75
intensity 75
Kanji 266
light pen 264
multiple definition 262
null-to-blank 75
outlining 270
setting defaults 262
symbol set 75, 239
translation tables 75
transparency 82
type 72, 74
fields 71
multiple definition 260
query modified 261
setting to modified or unmodified 263
input 72
introduction 54
light pen 264
menu example 257
mixed with mapped fields 289
multiline fields 72

alphanumerics, procedural *(continued)*
output 72
precedence over graphics 80
overriding on 3270-PC/G and /GX, 3179-G, 3192-G, 3472-G, 82
overriding on 3812, 3816 and 4224 printers 82
programming example 77
reverse-video fields 75
summary of function 71
support on plotters 433
symbol sets 233
trailing attribute bytes 74
translation tables 75
underscored fields 75
alternate device 382, 402, 412
copying graphics to printer 414
angles, rotation, and shear
graphics segments 149
text and symbols 66
animation effect 42
annotating graphics 53
APDEF (define high-performance alphanumeric field) 277
aperture, pick
See pick
API call parameters
data types of 11
APL characters
code page for 253
APL feature
TSO, nonqueriable displays 548
VM/CMS, nonqueriable APL displays and printers 536
APL2 3
high-performance alphanumerics (HPA)
restrictions 280
programming example 502
application anchor block (AAB) 4
application code page 249
application data structure (ADS) 283, 285
See also alphanumerics, mapped
application groups (windowing) 498
application image
creating an (IMACRT) 89
definition of 85
application programming
image 368
application programming interface (API) 3
See also external interfaces
arcs
circular 29
elliptic 29
area 30
change attributes inside 45
shading algorithm 31
array parameters 17, 19
ASCCOL (specify character colors within a field) 76
ASCGET (get field contents) 72
ASCHLT (specify character highlights within a field) 77
ASCPUT (specify field contents) 72
ASCSS (specify character symbol sets within a field) 239
ASDFLD (define or delete a single field) 71
ASDFLT (set default field attributes) 262
ASDTRN (define I/O translation tables) 75
ASFBDY (define field outline) 270
ASFCOL (define field color) 75
ASFCUR (position the cursor) 73
ASFEND (define field-end attribute) 75
ASFHLT (define field highlighting) 75
ASFIN (define input null-to-blank conversion) 75
ASFINT (define field intensity) 75
ASFMOD (change field status) 263
ASFOUT (define output blank-to-null conversion) 75
ASFPSS (define primary symbol set for a field) 75, 239
ASFSEN (define field mixed-string attribute) 267
ASFTRA (define field-transparency attribute) 82
ASFTRN (assign translation table set to field) 75
ASFTYP (define field type) 74
ASMODE (define the operator reply mode) 77
aspect ratio
of copied graphics 414
ASQCOL (query character colors for a field) 77
ASQCUR (query cursor position) 73
ASQHLT (query character highlights for field) 77
ASQMOD (query modified fields) 261
ASQNMF (query number of modified fields) 261
ASQSS (query character symbol sets for a field) 77, 242
ASRATT (define field attributes) 262
ASREAD (device output/input) 9, 80
mapping 289
partitions 458
ASRFMT (redefine fields) 260
assembler
format of call to GDDM 3
assembler language
ADMUAIMC 317
error code in register 15 136
parameter declarations 11
assigning data to alphanumeric field 72
ASTYPE 248
asynchronous interrupt on VM/CMS 534
attention interrupts under TSO 545
attribute adjunct
See alphanumerics, mapped
attribute adjuncts 315
attribute bytes on 3270-type hardware 74
attributes
default
standard 47

attributes, alphanumeric
  See alphanumerics
attributes, graphics
  See graphics
attributes, segment
  See graphics segments
auto-skip fields
  See also alphanumerics
  mapped 315
  procedural alphanumerics 75
automatic closure of queued printer devices 417
auxiliary device 433

B

background
  color 35, 42
  color mixing 44
badge reader
  translation into alphanumeric input 326
base attribute adjunct 315
  See also alphanumerics, mapped
BASIC 3
basic edit process for IMS 560
Basic Mapping Support 574
batch mode, VSE 595
batch processing
  MVS 550
  TSO 549
  VM/CMS 538
bibliography xxvii
black, special treatment of 42
blank-to-null conversion 75
blinking
  See also alphanumerics
  ASFHLT (define field highlighting) 75
  mapped data 327, 330
BMS and GDDM 574
books, list of xxvii
bottom right cell of screen 113
breaking lines of graphics text 64
browsing
  composite documents
    application control of 420
    device 420
bundle list
  update rule 280
bypass parameter checking
  ADMUFO 140
    restriction 141

C

C/370
  declaration of GDDM entry points 11
  format of call to GDDM 3
C/370 *(continued)*
  parameter declarations 11
calling segments 165
  inherited attributes 169
candidate operator window 484
capturing pictures 217
CDPDS 419
CDPF (Composed Document Printing Facility) 596
CDPU (Composite Document Print Utility) 419
CECPINP
  external default 253
CECPs 248
cell size, variable 474
  programming example 477
cent sign 242
CGLOAD (load CGM) 192
CGM
  code page for 179
  conversion profiles 179
    general purpose 179
  tailored for applications 179
CGM (computer graphics metafile) 173
  loading onto GDDM page
    CGLOAD call 192
  saving graphics as
    CGSAVE call 178
CGSAVE call 178
chained segment attribute 148
change field status 263
change resolution flag of an image (IMARF) 105
changes to GDDM
  compatibility of release 1.4 with earlier
    releases 177
changes to this book
  for Version 3 Release 1 xxix
changing image resolution (IMARES) 106
changing pictures 193
character
  See also symbol, alphanumerics
  box 64
    on 3270-PC/G and /GX 69
  code 233
  comparison of modes 61
  graphics 57
  GSCHAR (draw character string at specified
    point) 57
  mode 58
  national use 242
  shear 66
  space 65
  strings 57
    ways of displaying 53
character attributes 76, 330
  See also alphanumerics
character box
  on advanced function printers 70

character box *(continued)*
on IBM 4250 printers 70
plotter cell 438
character code page
See code page
character modes
mode-1 61
mode-2 61
mode-3 61
character size, variable 474
programming example 477
chart files, ICU, code-page conversion 252
choice input 202
associated with graphics field 214
data keys
GDDM-OS/2 Link 202
enabling and disabling device 206, 230
data key 202
enabling data keys for 202
initializing device 211
input data 202
querying 202
CICS
ADMASXC (COBOL error-exit name) 139
ADMUOFF control section 577
BMS and GDDM 574
compiling GDDM application programs 584
GDDM default error exit 576
print utility 582
using GDDM 567
CICS conversational programs 567
CICS pseudoconversational control 510
CICS/ESA
example of JCL
compile Assembler 589
compile C/370 588
compile COBOL 587
compile PL/I 586
GDDM code above 16MB 583
CICS/VSE
example of JCL
compile Assembler 593
compile C/370 592
compile COBOL 591
compile PL/I 590
circle displayed as oval 28
circular arcs 29
clear
graphics field 147
clear a rectangle in an image (IMACLR) 105
CLEAR key
enabling as logical input device 207, 230
terminates plotting 438
translation into alphanumeric input 326
clipping 113, 125
after GSLOAD 187
clipping *(continued)*
and GSLOAD 185
by GSSAVE 177
close device 389
close segment 145
closure of an area, automatic 31
CMS
ADMASXV (COBOL error-exit name) 139
CMSINTRP processing option 534
CMSTRCE, VM trace filename/filetype 135
COBOL
ADMUCIMC 317
error exits 138
format of call to GDDM 3
parameter declarations 11
code page 425
conversion 247, 248
for 4250 253
SBCS Japanese input 251
country-extended 248
definition 248
for CGM data 179
code page translation
for SBCS Japanese input
FSTRAN 251
code point 248
code, character 233
color 35, 327, 330
See also alphanumerics
See also multi-colored
3270-PC/G and /GX 49
ASFCOL (define field color) 75
changing inside an area 45
codes 75
GSBMIX (set current background color-mixing mode) 44
GSMIX (set current foreground color-mixing mode) 40
mapped data 327, 330
mixing 40
of line bounding an area 40
on plotters 446
on workstations
GDDM-OS/2 Link 49
GDDM-PCLK 49
unsupported by device 36
color mixing
on workstations
GDDM-OS/2 Link 49
GDDM-PCLK 49
color-separation masters 428
range of colors 51
combining segments 163
comment order, GDF 193
compatibility of release 1.4 with earlier releases
GDF (graphics data format) 177

compiling a GDDM program 14
mapped alphanumerics 288, 295
sample programs 524
under CICS 584
under VM/CMS 529
compiling and running GDDM programs 14
complex pictures, checking 397
Composed Document Printing Facility (CDPF) 596
Composite Document Print Utility
invoking from applications
CDPU call 419
composite documents
displaying on screen 419
DisplayWrite/370
CDPDS 419
output of
on display devices 420
specifying device 420
printing 419
computer graphics metafile (CGM) 173, 178, 192
IMS restriction 553
concatenating graphics text 65
console, user 371
constant data fields 283, 293
construction lines of polyfillet 30
control echoing of scanner image (ISESCA) 88, 90
controlling image quality (ISCTL, ISXCTL) 359
convert the resolution attributes of an image (IMARES) 106
coordinates
current
See current position
querying
current position 32
locator input 201, 204
pick input 201
coordinating device (windowing) 480
coordination exit routine 480, 495
copying graphics 160
copying pictures between devices and systems 193
copying screen output
to a plotter 441
to a printer 413, 414
with control of size and position 412
to an alternate device 382
copying, inter-device
using GSSAVE and GSLOAD 186, 188
correlation by workstation 197
correlation of tag to primitive (GSCORR) 225
Country Extended Code Pages
querying 395, 396
country-extended code pages 248
CPSPOOL processing option 418
CPTAG processing option 418
create an empty projection (IMPCRT) 95, 103
create an image (IMACRT) 89, 92
create graphics field 112
create picture space 113
create viewport 114
current device 371
current operator window 484, 489, 490
modify
WSMOD 490
query
WSQRY 494
current page 111
current partition 459
current position 8
querying 32
cursor
always within scrolling window 474
for selecting from menu 322
positioning
in ASREAD output 9, 73
in FSFRCE output 10, 73
in GSREAD output 214
in mapped ASREAD output 318
in MSREAD output 318
querying
See pick, locator, stroke
querying position
mapped alphanumerics 320
procedural alphanumerics 73
specifying type 211
cursor adjunct 318
See also alphanumerics, mapped
CURSR SEL key
mapped fields 321

D

data (gray) keys
enabling as logical input device 207
input to GSREAD 203
data buffer
update rule 280
updating a data buffer 280
data characteristics
CICS data sets 580
IMS data sets 559
VM/CMS files 532
data entry example 454
data sets and file processing
CICS 578
IMS 559
TSO 540
data type field attribute 317
DBCS 274
alphanumerics 266
graphics text 244

DBCSLNG default parameter 246
DCB characteristics for TSO data sets 542
DCSS (Discontiguous Shared Segment) 530, 531
debugging aids 131
declaration of GDDM entry points in C/370 11
declaration of GDDM entry points in PL/I 11
default
  device 371, 383
  drawing 47
  error exit 137
  error threshold 137
  external 47
    for trace facilities 134
  field attributes 262
  graphics attributes 47
  standard 47
  symbol set 237
default data in mapping 293
  See also alphanumerics, mapped
default error exit
  CICS 576
  IMS 559
default, external
  for CECP support
    CECPINP 253
defaults module and file 135
  nicknames 135
  nicknames in 375
  parameters for GDDM call tracing 135
  parameters for Kanji graphics text 246
defensive programming 391
define bi-level conversion algorithm (IMRCVB) 342
define brightness conversion algorithm (IMRBRI) 340
define contrast conversion algorithm (IMRCON) 341
define high-performance alphanumeric field (APDEF) 277
define image field (ISFLD) 365
define place position in pixel coordinates (IMRPL) 97
define place position in real coordinates (IMRPLR) 97
define rectangular sub-image in pixel coordinates (IMREX) 96
define rectangular sub-image in real coordinates (IMREXR) 96
delete
  segment 147
delete graphics field 112
delete projection (IMPDEL) 98
delete the image associated with the identifier (IMADEL) 91
designator of light pen field
  mapping 321
  procedural alphanumerics 264
detectable field attribute 317
  procedural alphanumerics 265
detectable segment 148, 200
device 108
  alternate 382
  characteristics 392
  close 389
  conceptual 371
    querying characteristics 392
  current 371
  default 371
  definition
    altered by nicknames 380
    changed by end user 374
    DSOPEN 372
    nicknames and DSOPEN 374
    querying nicknames (ESQUNS) 376
  definition tables 373
  dependency 391
  dummy 387
  errors in full-screen mode (TSO) 546
  independence 391
  logical input 197
    See also choice, locator, pick, string, stroke
  mapgroup suffixes 304
  more than one 383
  physical 373
    querying characteristics 392
  primary 382
  properties 372
  query 392
  reinitialize 390
  support 371
  symbol set suffixes 242
  token 372, 404
    for 3825 printers 404
    in nickname statement 375
  usage 382
    reinitialize 390
  variation 391
device class for a map 292
device code page 249
device image
  creating an (IMACRT) 89
  definition of 85
device independence 391
device independent programs 391
device variations 391
DEVTOK nickname parameter 375
dialed devices with GDDM/VMXA 536
digitizing
  See also stroke input
  example 217
direct transmission, of image 356, 364
direction of graphics text 66
disable clipping, GSCLP 125
disabling image cursors (ISENAB) 348
disabling logical input device 207, 230
  See also GSENAB

disabling logical input device *(continued)*
image (FSENAB) 348
when advisable 214
Discontiguous Shared Segment (DCSS) 530, 531
discontinue device usage (DSDROP) 383
displacing segment origin 158
displacing segments 149
display format
See alphanumerics, mapped
display-device conventions
CICS 577
TSO 544
VM/CMS 533
dividing the screen 115
DL/I
databases 562
GDDM interface 560
dollar sign 242
dots, use of 17
double-byte character set (DBCS) 244, 266
double-byte characters
for graphics text 244
dragging segments
example 219
problems 221
draw
circular arc 29
elliptic arc 29
graphics area 30
graphics marker 30
image 32
polyfillet 29
series of lines 29
several graphics markers 30
straight line 28
text at current position 57
text at specified point 57
drawing chain 145
drawing defaults 47
drawing graphics text 57
drawing interactively on the screen 197
example program 217
drawing order 164
drop (discontinue) device (DSDROP) 383
DSCLS (close a device) 389
DSCOPY 368
DSCOPY (send transformed picture to alternate device) 412
DSDROP (discontinue device usage) 383
DSOPEN
query effects of
DSQDEV 392
DSOPEN (open a device) 371
for a plotter 433
simplifying the call 374
to print color masters 430
DSOPEN (open a device) *(continued)*
use for operator windows 480
DSQDEV (query DSOPEN parameters) 392
DSQUSE (query device usage) 392
DSRNIT (reinitialize a device) 390
DSUSE (specify device usage) 382
for alternate device 412
dual-screen terminals 71
mapping 284, 333
dummy device 387
duplicate identifiers 387
dynamic load of system programmer interface
IMS 557
TSO 539

E

EBCDIC 247, 253
echo
locator device 211
querying 215
segment, how drawn 220
stroke device 213
editing pictures 193
electro-erosion printers 404
ellipse displayed instead of circle 28
ellipses, drawing 29
elliptic arc, draw (GSELPS) 29
elliptic arcs
direction of drawing 29
enable or disable image cursor (ISENAB) 348
enabling clipping 125
enabling logical input device 200, 206, 230
and initializing 211, 214
image (FSENAB) 348
pick, locator and stroke together 214
querying 215
encapsulated PostScript (EPS) 409
enlarging segments 149
ENTER key
enabling as logical input device 207, 230
translation into alphanumeric input 326
entry-points to GDDM 11
EPSBIN files 409
erasing by overpainting in black 42
error
checking picture complexity 397
exits 137
handling 131
messages 132
processing 10
query last 133
record 133
return codes 132
in register 15 136

error exits
CICS default error exit 576
COBOL 138
FORTRAN 138
IMS 559
PL/I 138
user-defined
COBOL 138
error handling
COBOL
user-defined exit routines 138
return error record using FSQERR 138
error-log record 576
errors in full-screen mode (TSO) 546
ERXMENU
GDDM-REXX sample program
output from 260
ERXMENU REXX sample program 519
ERXMODEL REXX sample program 519
ERXOPWIN REXX sample program 519
ERXORDER REXX sample program 519
ERXPROTO REXX sample program 519
ERXTRY
interactive learning tool 10
ERXTRY REXX sample program 519
ESACRT (create application group) 498
ESADEL (delete application group) 498
ESAQRY (query current application group) 498
ESASEL (select application group) 498
ESQUNL 379
ESQUNL (query length of user's nickname information) 376
ESQUNS 379
ESQUNS (query user's nickname information) 376
ESSUDS (specify source-format user default specification) 381
example JCL
copy page segments from phase library to VSAM file 597
defining spill files 597
example of program
AID translation 327
changing graphics-text attributes 58
color adjunct 327
concatenating graphics text 65
copying screen output to a printer 416
creating masters for color printing 430
cursor adjunct 319, 321
cursor selection 322
directly-attached printer as primary device 401
graphics and mapping 333
light pen selection 322
mapped menu 322
PF key selection from menu 322
plotting a saved picture 439
queued printer 402
example of program *(continued)*
selector adjunct 308
setting the symbol set attributes for alphanumerics 240
to accept procedural alphanumeric input and display graphics 80
to create partitions that can be scrolled 477
to create partitions with different cell sizes 477
to display a bank balance using procedural alphanumerics 77
to display a menu using procedural alphanumerics 257
to draw a spider with graphics image 32
to draw a street map using GDDM graphics calls 25
to print an image on the 4224 367
to provide an operator window for user dialog 481
to provide two operator windows (each a virtual device) 485
to read in data in separate partitions 454
to scale an image to fit display screen 346
to trim images interactively 351
to trim images interactively with part-screen image field 354
to use a dummy device 388
to use a multipart graphics area 31
to use two primary devices 384
using 4250 fonts 426
using a symbol set for graphics text 237
using a system printer 404
using advanced-function printers 404
example of programming
with graphics text 62
example programs
calling segments 166
correlation 226
dragging segments 219
floating maps 301
freehand drawing or digitizing 217
graphics menu 198
image scanning, displaying and saving 87
multiple maps 297
picking symbol primitives 198
redefining graphics windows and viewports 121
restoring a projection and saving an image 98
simple mapping 285
simple mapping program 291
stroke input 205
viewports 115
exclusive-OR drawing mode 220
executing a GDDM program 14, 529
mapped alphanumerics 288
mapping 295
exporting pictures 193
Extended Binary Coded Decimal Interchange Code
See EBCDIC

extended set image quality-control parameters
(ISXCTL) 363
external default
always unlock keyboard
AUNLOCK 572
control size of output
IOBFSZ 561
control validation of GDDM call parameters
FRCEVAL 281
controlling error information
ERRFDBK 136
for APL data analysis feature
CMSAPLF 537
for CECP conversion
CECPINP 253
for trace facilities
CMSTRCE 135, 532
TRCESTR 135, 582
for trace facilities (IMS)
IMSTRCE 559
queued printing on IMS
IMSPRNT 562
release storage when abend
CICAUD 572
system printing on IMS
IMSSYSP 562
external defaults
passing to GDDM 135
external interfaces 4
nonreentrant 4
reentrant 4
system programmer 5

F

FAM nickname parameter 375
family of device
printer 399
family of output 372
in nickname statement 375
family-4 output
retrieval by application
FSGETS, FSGET, FSGETE 409
family-4 printing
spill file 407
fast-path processing (ADMUFO) 140
field attributes
See alphanumerics
field end, procedural, setting attribute (ASFEND) 75
field list
update rules 279
fields
mapped 283
procedural alphanumeric 71
fields, introduction 54
file processing
IMS 559
file, defaults 135, 375
file, graphics 173
files, GDDM, code-page conversion 252
files, non-GDDM
printing 421
fixed-point GDF 177
floating-point GDF 177
folding mapped input 332
FONT4250 code pages 425
FONT4250 default file name/filetype
TSO 542
FONT4250 default filename/filetype
VM/CMS 532
FONT4250 fonts 424
fonts 58, 234
3800 printer symbol sets 423
4250 typographic 424
for alphanumerics 266
foreground color
exclusive-OR mix (XOR) mode 41
mix mode 40, 41
overpaint mode 40
transparent mode 41
underpaint mode 41
format
for data sets/files
CICS 580
required data sets/files
IMS 559
TSO 542
VM/CMS 532
format of call to GDDM 3
format of GDDM error record 133
formatting the screen
See alphanumerics, high-performance
See alphanumerics, mapped
FORTRAN
error exits 138
format of call to GDDM 3
limited mapping support 284
parameter declarations 11
freehand drawing example 217
FSALRM (sound the alarm) 80, 315
FSCHEK (check picture complexity before output) 397
FSCOPY (send page to alternate device) 413
FSENAB (enable/disable device input) 210, 348
FSEXIT (specify error exit) 137
FSFRCE (update the display) 10
mapping 305
partitions 458
FSGET
retrieval of family-4 datastream
example 409

FSINIT (initialize GDDM processing) 7
FSLOG (send character string to alternate device)
  send text to queued printer 414
FSPCRT (create a page)
  effect on cell size of 3290 475
FSPWIN (set page window) 473
  effect on cell size of 3290 475
FSQERR (query last error) 133
FSQURY (query device characteristics) 392
  image related 339
FSTERM (terminate GDDM processing) 7
FSTRAN (translate code page of user input) 251
full-screen mode errors under TSO 546

G

GDDM
  CICS 567
  code resident above 16M 550, 556, 583
  MVS Batch 550
  TSO batch 549
  using under MVS
    batch 550
  using under TSO
    batch 549
  VM/CMS 529
GDDM application programs
  compiling, loading and running
    under TSO 14
    under VM/CMS 14, 529
GDDM concepts
  hierarchy of 107
GDDM image objects
  See stored image
GDDM Internet home page xxvi
GDDM object files with code page tag 252
GDDM objects
  code-page conversion 252
GDDM symbol sets 248
GDDM-IMD
  See Interactive Map Definition
GDDM-OS/2 Link
  IMS restriction 553
GDDM-OS/2 Link workstations
  color mixing 49
  shading patterns
    restriction 49
  supported colors 49
GDDM-PCLK workstations
  color mixing 49
  shading patterns
    restriction 49
  supported colors 49
GDDM-REXX
  application data structure (ADS)
    GXSET subcommand 284
GDDM-REXX *(continued)*
  format of call to GDDM 3
  high-performance alphanumerics (HPA)
    restrictions 280
  introduction 16
  mapped alphanumerics 284
  output from sample program
    ERXMENU 260
  parameters 18
  programming example 508
  starting to use 17
  trace facility 134
GDDM/VMXA 538
GDDMREXX command
  INIT 16
  TERM 16, 22
GDF (graphics data format) 173
  printer spill file 407
  storing in files 173
general light-pen fields 265
generating mapgroup 293
  See also alphanumerics, mapped
geometric pattern set - ADMPATTC 39
get and reserve a unique image identifier (IMAGID) 92
get and reserve a unique projection identifier
 (IMPGID) 105
getting data from an image
 (IMAGTS,IMAGT,IMAGTE) 358
GIF file data characteristics
  CMS 532
GL
  See IBM GL
GLOBAL commands needed under VM/CMS 529
GOCA 407
GRAF option 418
graphic area of mapped display 332
graphics
  and alphanumerics 54
  and mapping 332
    programming example 333
  area 30
  attributes 35, 146
    changing 193
    changing default values 47
    changing inside an area 45
    default values 8
    pushing and popping values 46
    querying 46
  character strings 57
  clipping 125
  concepts 107
  coordinate system 117
  device 108
  device considerations 393
  drawing on the screen 197
  field 112
    clear 147

graphics *(continued)*
field *(continued)*
effect on logical input devices 214
giving precedence to alphanumerics 61
graphics window 118
image 32
input 208
interactive 197
library 173
markers 30
multiple markers 30
page 111
plotter considerations 446
positioning when copying 422
primitive 25
background overlapping other primitive(s) 44
changing 193
identifier 197
outside segment 170, 171
query tag when picked 200
tag 197
scrolling 476
segment
See also graphics segments
viewing limits 128
storing 173
text 57
See also graphics text
using PS with 397
variable cell size on 3290 476
window 118
See also window, graphics
graphics concepts
hierarchy of 107, 387, 393
graphics data format (see GDF)
graphics field
and GSLOAD 180
clipping 125
graphics hierarchy 387
graphics markers
user-defined 36
graphics menu example 198
graphics orders
CGM
saved 174
GDF
ignoring 182
saved 173
graphics segments 8, 145
as locator echo 220
example 219
attributes 147, 200
chained 148
detectable 148, 200
highlighting 148
nonchained 148
transformable 148
graphics segments *(continued)*
attributes *(continued)*
visibility 148
attributes modification 148
called
loading 182
calling 165
inherited attributes 169
closing 145
copying 160
deleting 147
displacing 149
dragging by end user 219
drawing chain 145
library 173
moving 149
moving and transforming 223
origin 150
moving 158
querying 159
picking example 215
query identifier when picked 200
querying 165
reference point 222, 223
relation to graphics hierarchy 120
reopen, not permitted 146
rotating 149
saving 176
scaling 149
segment origin 221, 223
shearing 149
storing 173
structure 145
example 167
transformations 149
unnamed 149
loading 182
renumbering 182
untagged primitives
loading 182
tagging 182
with zero identifier 149
graphics stored as CGM
loading into programs 192
graphics stored as GDF
loading into programs 180
graphics symbol sets 233
graphics text 8, 57
alignment in text box 67
angle of slope 66
appearance of 58
attributes 58
character
angle of rotation 66
italic 66
mode 58, 64
shearing 66

graphics text *(continued)*
character-positioned mode 60, 61
characters
proportional spacing of 68
comparison of modes 61
device considerations 395
device variations 69
direction 66
drawing, at current position 57
drawing, at specified position 57
enlarging 64
input 204
introduction 53
line break 64
loading symbol sets 235
mode 58
character-positioned 60
string-positioned 59
stroke-positioned 60
mode-1 59, 60
mode-2 60
curing unexpected overlap 65
mode-3 60
on 3179-G 69
on 3192-G 69
on 3270-PC/G and /GX 69
on 3472-G 69
on advanced function printers 70
readability 64
reverse-video 44
rotation 66
rounding errors 61
selecting symbols for 53
shearing 66
size 65
string-positioned mode 61
string-positioning mode 60
stroke-positioned mode 61
symbol set example 237
symbol sets for 235
text box
outlining 66
unexpectedly upside-down 119
using symbol sets 235
gray keys
See data keys
grid, partition set 453
GSAM (set attribute mode) 46
GSARC (draw a circular arc) 29
GSARCC (specify aspect-ratio control (for copy))
with FSCOPY 423
with GSCOPY 414
GSAREA (start a shaded area) 30
GSBMIX (set current background color-mixing mode) 44
GSBND (define a data boundary) 125
GSCA (set current character angle) 66
GSCALL (call a segment) 165
GSCB (set character-box size) 64
on advanced-function printers 70
GSCBS (set character-box spacing) 65
GSCD (set current character direction) 66
GSCH (set current character shear) 66
GSCHAP (draw a character string at current position) 57
GSCHAR (draw a character string at a specified point) 57
GSCLP (enable and disable clipping) 125
GSCLR (clear graphics field) 147
GSCM (set current character mode) 58
GSCOL (set current color) 35
GSCOPY (send graphics to alternate device) 414
GSCORR (explicit correlation of tag to primitive) 225
GSCORS (explicit correlation of structure) 228
GSCPG (set current code page) 425
GSCS (set current symbol set) 236
GSDSS (load a graphics symbol set from the application program) 243
GSELPS (draw an elliptic arc) 29
GSENAB (enable or disable a logical input device) 206, 230
and initialization calls 211, 214
enabling pick, locator and stroke together 214
when to issue 214
GSFLD (define the graphics field) 112
GSFLW (set current fractional line width) 36
GSGET (retrieve graphics data) 193
GSGETE (end retrieval of graphics data) 193
GSGETS (start retrieval of graphics data) 193
GSIDVF (initial data value, float) 212, 213
GSIDVI (initial data value, integer) 212
GSILOC (initialize locator) 211
GSIMG (draw a graphics image) 32
GSIMGS (draw a scaled graphics image) 32, 34
GSIPIK (initialize pick device) 213
GSISTR (initialize string device) 213
GSLINE (draw a straight line) 28
GSLOAD (load segments) 180
GSLSS (load a graphics symbol set from auxiliary storage) 235
GSLSS (load graphics symbol set from auxiliary storage)
on 3270-PC/G and /GX 255
GSLT (set current line type) 36
GSLW (set current line width) 36
GSMARK (draw a marker symbol) 30
GSMB (set marker scale) 37
GSMIX (set current foreground color-mixing mode) 40
GSMOVE (move without drawing) 28
inside an area 31

GSMRKS (draw series of marker symbols) 30
GSMS (set the current type of marker symbol) 36
GSPAT (set current shading pattern) 37
GSPFLT (draw a curved fillet) 29
GSPLNE (draw series of lines) 29
GSPOP (restore attributes) 47
GSPS (define the picture space) 113
GSQAGA (query all geometric attributes) 156
GSQCB (query character-box size) 65
GSQCHO (query choice device data) 202
GSQCOL (query current color) 46
GSQCP (query current position) 32
GSQLID (query logical input device) 215
GSQLOC (query graphics locator data) 201
GSQLW (query current line width) 46
GSQORG (query segment origin) 159
GSQPIK (query pick data) 200
GSQPKS (query pick structure) 201
GSQPRI (query segment priority) 164
GSQPS (query picture-space definition) 115
GSQSIM (query existence of simultaneous queue entry) 209
GSQSTK (query stroke data) 204
GSQSTR (query string data) 204
GSQTB (query the text box) 67
GSQTFM (query segment transform) 156
GSREAD (await graphics input) 208
  partitions 458
GSSAGA (set all geometric attributes) 150
GSSATI (set initial segment attributes) 148
GSSATS (modify segment attributes) 148
GSSAVE (save a segment) 176
GSSCLS (close the current segment) 145
GSSCPY (copy a segment) 160
GSSCT (set current transform) 45, 159
GSSDEL (delete a segment) 147
GSSEG (create a segment) 8, 145
GSSINC (include a segment) 162
GSSORG (set segment origin) 158
GSSPOS (set segment position) 153
GSSPRI (set segment priority) 164
GSSTFM (set segment transform) 150, 151, 154
GSSVL (define segment viewing limits) 128
GSTA (set text alignment) 67
GSTAG (set current primitive tag) 199
GSUWIN (define a uniform graphics window) 28, 118
GSVIEW (define a viewport) 114
GSWIN (define a graphics window) 117
GXSET subcommand
  application data structure (ADS)
    REXX programming 284

H

Hangeul
  alphanumerics 266
hardcopy of graphics output 403
hardware attribute bytes 74
hardware symbols 60, 61
  as default symbol set 239
header pages for printer 403
hidden surface 164
hierarchy of GDDM concepts 107, 387, 393
high-performance alphanumerics 273
  introduction 55
  restrictions
high-performance alphanumerics (HPA)
  attributes of characters 277
  attributes of fields 277
    changing 278
  bundle list
    declaration 274
    initialization 274
  character attributes 277
  compared with mapped alphanumerics 56
  compared with procedural alphanumerics 56
  data buffer
    declaration 274
    initialization 274
    setting up 277
  defining fields 276
  displaying fields again 279
  double- and single-byte characters 274
  dynamic fields 280
  enlarging structures 280
  field attributes 277
    changing 278
  field definition 276
  field list
    declaration 273
    definition 277
    initialization 273
    update rules 279
  field list cursor position 276
  field list depth 276
  field list status 276
  field list width 276
  how to use 273
  input 279
  locate mode 278
  mode 278
    cursor 278
    locate mode 278
    move mode 278
  output 273
  programming example 274
  reshow 279
  restrictions
    FRCEVAL external default 281
    use of shared storage 281
    with interpreted languages 280
  SBCS and DBCS characters 274

high-performance alphanumerics (HPA) *(continued)*
status of field list 276
updating a bundle list 280
updating a field list 279
use with read-only storage 281
validation 281
forced by default 281
high-resolution printers
See page printers
highlighting
ASFHLT (define field highlighting) 75
mapped data 327, 330
segment attribute 148
home page for GDDM xxvi
host offload, to image devices 359, 368
HPA
See high-performance alphanumerics

I

IBM GL
plot files 437, 446
CICS restriction 567
IMS restriction 553
ICU (Interactive Chart Utility)
ADMGDF files 173
identifier
symbol-set 235
identifier,
See also tag
primitive 197
identify device to GDDM 373
IMACLR (clear a rectangle in an image) 105
IMACRT (create an image of the specified size, type, and resolution) 89, 92
IMADEL (delete the image associated with the identifier) 91
image 32
ADMIMG file 91
ADMPROJ file 97
application programming 368
aspect ratio, preserving 346
attributes of target 101
bi-level, definition of 340
box cursor 348
enabling/disabling (ISENAB) 348
initializing (ISIBOX) 350
querying (ISQBOX) 349
size or shape change, keys for 351
brightness conversion (IMRBRI) 340
changing resolution values (IMARES) 106
clearing an (IMACLR) 105
clipping to target rectangle 97
completing image transform (IMRPL) 97
completing image transform (IMRPLR) 97
compressions supported 344, 356
image *(continued)*
contrast conversion (IMRCON) 341
converting resolutions (IMARES) 106
creating a target, implicitly 90, 97
creating an (IMACRT) 89, 92
cross cursor 348
enabling/disabling (ISENAB) 348
initializing (ISILOC) 350
querying (ISQLOC) 349
cursors 348
initializing 350
movement, keys for 351
type selection 350
data transfer to/from your program 355
definition of 85
deleting an (IMADEL) 91
device variations 370
direct transmission 356
direct transmission from a scanner 364
direct transmission to the 3193 364
display station (3193)
introduction to 85
programming for 87
display station (3193), end use of 350
editing without transfer 105
entering data into an (IMAPTS,IMAPT,IMAPTE) 356
extracted image
definition of 93
extracting a sub-image (IMREX) 96
extracting a sub-image (IMREXR) 96
field, defining (ISFLD) 364, 365
file
See image, stored
file format
use of your own 86
filename of stored 91
formats supported 344, 356
getting data from GDDM 358
gray-scale
conversion to binary 340
gray-scale to bi-level conversion (IMRCVB) 342
gray-scale, definition of 340
halftone, definition of 340
halftoning 342
identifiers 86
obtaining (IMAGID) 92
reuse 91
value range 92
identity projection
definition of 87
implicit creation of target 90, 97
improved performance
host offload 359, 368
input device enabling/disabling (FSENAB) 348
input/output synchronization (ASREAD) 91

image *(continued)*
interactive input 348
example 351
inverting an (IMRNEG) 102
locator cursor 370
See also image, cross cursor
on PS displays 370
merging 97
multiple extraction 361
multiple placing 361
negating an (IMRNEG) 102
on plotters 449
performance/function trade-offs 359
positioning in target image (IMRPL) 97
positioning in target image (IMRPLR) 97
printer (4028)
programming for 366
printer (4224)
introduction to 85
printing 366
on an IPDS printer 366
projection 97
applying a 98
changing a 104
completion of 97
contents, explanation of 93
creating a (IMPCRT) 95
definition of 87
deleting a (IMPDEL) 98
effect on source image 87
evaluation order 104
example code to define and save 95
explanation of contents 93
extract, scale, and save example 95
identifiers reuse 98
identifiers value range 105
identifiers, obtaining (IMPGID) 105
identity, definition of 87
illustration of 94
invoking a 98
library 95
multiple transform example 104
operations making up a 93
order of evaluation 104
restoring from auxiliary storage (IMPRST) 98
saving on auxiliary storage (IMPSAV) 97
use in IMARST call 92
use in IMASAV call 104
uses of 93
putting data to GDDM 356
quality control parameter, setting extended (ISXCTL) 363
quality-control parameters (ISCTL) 362
quality, controlling (ISCTL, ISXCTL) 359
querying attributes (IMAQRY) 93
querying compressions (ISQCOM) 344

image *(continued)*
querying device characteristics (FSQURY) 339
querying formats (ISQFOR) 343
querying resolutions (ISQRES) 345
querying scanner device (ISQSCA) 339
querying scanner status (FSQURY) 339
reflecting an (IMRREF) 102
reorienting an (IMRORN) 101
resolution type
changing the (IMARF) 105
resolution/scaling algorithm
description of alternatives for 103
during resolution change (IMARES) 106
setting the (IMRRAL) 102
restoring from auxiliary storage (IMARST) 92
retrieving data from an (IMAGTS,IMAGT,IMAGTE) 358
reversed polarity 357
same source and target, using (IMXFER) 106
saving on auxiliary storage (IMASAV) 91
scaling algorithm, control of 361
scaling an (extracted) image (IMRSCL) 97
scaling and conversion, control of 361
scaling to fit 346
scan, display, and save example 87
scanner
brightness control (IMRBRI) 340
contrast control (IMRCON) 341
device identifier for 89
echo control (ISESCA) 88, 90
image conversion to bi-level (IMRCVB) 342
loading and ejecting paper (ISLDE) 89, 91
order of conversion calls 343
paper size 90
programming for 87
querying status (FSQURY) 339
querying status (ISQSCA) 339
scanner (3118)
introduction to 85
resolutions 89
size change by scaling (IMRSCL) 97
size rounding, control of 360
stored
definition of 86
target rectangles, control of 362
transfer operations 87, 92, 98, 356
editing without 105
effects on image attributes 101
transferring data (IMXFER) 90, 106
transferring into your program 355
transferring out of your program 355
transform
calls sequence 103
contents 94
definition of 93
illustration of 94
mandatory call 97

image *(continued)*
transform *(continued)*
sequence of calls 103
transform element
introduction to 93
operations 94
trimming (IMATRM) 105
example 351
turning an
See image, reorienting an (IMRORN)
type conversion to bi-level (IMRCVB) 342
undefined resolution
changing to defined (IMARF) 105
with graphics or text 365
Image Print Utility 366
image processing
advanced 339
introduction to 85
Image Symbol Editor 36, 37, 38, 233
image symbols 61, 233
curing unexpected overlap 65
on plotters 449
spacing 65
two types of 233
image text 57
IMAGID (get and reserve a unique image identifier) 92
IMAGT (retrieve image data from an image) 359
IMAGTE (end retrieval of data from an image) 359
IMAGTS (start retrieval of data from an image) 358
IMAPT (enter data into an image) 357
IMAPTE (end data entry into an image) 357
IMAPTS (start data entry into an image) 356
IMAQRY (query attributes of an image) 93
IMARES (convert the resolution attributes of an image) 106
IMARF (change resolution flag of an image) 105
IMARST (restore image from auxiliary storage) 92, 104
IMASAV (save image on auxiliary storage) 91, 104
IMATRM (trim an image down to the specified rectangle) 105
IMPCRT (create an empty projection) 95, 103
IMPDEL (delete projection) 98
IMPGID (get and reserve a unique projection identifier) 101, 105
importing pictures 193
improved performance
for error-free code
ADMUFO 140
IMPRST (restore projection from auxiliary storage) 98
IMPSAV (save projection on auxiliary storage) 97
IMRBRI (define brightness conversion algorithm) 340
IMRCON (define contrast conversion algorithm) 341
IMRCVB (define bi-level conversion algorithm) 342
IMREX (define rectangular sub-image in pixel coordinates) 96, 103
IMREXR (define rectangular sub-image in real coordinates) 96, 103
IMRNEG (negate the pixels of an extracted image) 102
IMRORN (orient extracted image) 101
IMRPL (define place position in pixel coordinates) 97, 103
IMRPLR (define place position in real coordinates) 97, 103
IMRRAL (set current resolution/scaling algorithm) 102
IMRREF (reflect extracted image) 102
IMRSCL (scale extracted image) 97
IMS
ADMASXI (COBOL error-exit name) 139
application program structure 554
basic edit, use of 560
databases, use of 562
default error exit 559
DL/I interface 560
dynamic load and SPI 557
example of JCL
compile COBOL 566
compile PL/I 565
GDDM code above 16MB 556
message format service (MFS), use of 560
message queues 561
message size of segments 561
object import/export utility 564
PCB (program communication block) 554
PSB (program specification block) 557
restrictions 553
sample program for
ADMUSP1I 520
ADMUSP2I 520
SCS printers 560
using GDDM 553
with GDDM-PGF utilities 563
IMXFER (transfer data between two images, applying a projection) 90, 101, 106
including graphics 162
initial data in mapping 293
See also alphanumerics, mapped
initializing GDDM 7
initializing image cursors (ISILOC and ISIBOX) 350
initializing logical input device 211
and enabling 211, 214
input
processing (IMS and MFS) 560
input/output
See also logical input device
basic (ASREAD and FSFRCE) 9
for interactive graphics (GSREAD) 197, 208
introduction 9
mapped (ASREAD) 289
mapped (MSREAD) 286
of character attributes 77

input/output *(continued)*
of procedural alphanumeric data 72
partitions 458
insert-mode key 75
installation code page 250
instances of GDDM and GDDM-REXX 21
intensity
ASFINT (define field intensity) 75
mapped field attribute 316
inter-device picture transfer 193
inter-system picture transfer 193
interactive graphics 197
device considerations 396
with more than one partition 229
interactive learning
GDDM programming
ERXTRY 10
Interactive Map Definition (GDDM-IMD) 54, 283
Interactive Map Definition product (GDDM-IMD) 291
interfaces
external 4
nonreentrant
CICS 577
interfaces to GDDM (reentrant, nonreentrant, system-programmer) 4
internal DSOPEN 383
Internet home page for GDDM xxvi
interrupt
from partitioned screen 458
from windowed device 490
handling by ASREAD 9
handling by GSREAD 208
interrupt on VM/CMS 534
introduction
GDDM-REXX 16
inverting an image (IMRNEG) 102
inverting the graphics window 119
invisible field attribute
See alphanumerics
INVKOPUV processing option 418
IOCA 407
IPDS printers 51
IPDSBIN processing option 375
ISCTL (set image quality-control parameters) 359
ISENAB (enable or disable image cursor) 348
ISESCA (control echoing of scanner image) 88, 90
ISFLD (define image field) 364, 365
ISIBOX (initialize image box cursor) 350
ISILOC (initialize image locator cursor) 350
ISLDE (load external read-only image) 89, 91
ISQBOX (query image box cursor) 349
ISQCOM (query image compressions supported by the device) 344
ISQFLD (query image field) 366
ISQFOR (query image formats supported by the device) 343
ISQLOC (query image locator cursor position) 349
ISQRES (query supported image resolutions) 345
ISQSCA (query image scanner device) 339
ISXCTL (extended set image quality control parameters) 359

J

Japanese device support
extended (Katakana) code page 290
translation 251
JCL examples
copy page segments from phase library to VSAM file 597
defining spill files 597
justifying mapped input 332

K

Kanji
alphanumerics 266
graphics text 244
Katakana 248
keyboard, locking and unlocking 315
when screen partitioned 458

L

languages, programming 3
layout of the screen
See alphanumerics, high-performance
See alphanumerics, mapped
learning
GDDM-REXX parameters 18
multiple instances 21
learning interactively
GDDM programming
ERXTRY 10
length adjunct 321
See also alphanumerics, mapped
library, graphics 173, 188
light pen
enabling as logical input device 230
mapping 317, 321
procedural fields 264
translation into alphanumeric input 326
line
changing inside an area 45
GSLINE (draw a straight line) 28
multicolored area boundary 40
on plotters 449
type 36
width 36
line break
in graphics text 64

line width 113
link-editing GDDM application programs
  in VSE batchmode 595
  sample programs 525
  under CICS 584
  under IMS 557
  under TSO 539
linking fields in GDDM-IMD 293
  See also alphanumerics, mapped
load external read-only image (ISLDE) 89, 91
load graphics symbol sets 235, 243
loading graphics from ADMGDF files 180
loading graphics from CGM files 192
loading graphics from external storage 179
locator input 201
  associated with graphics field 214
  dragging segment 219
  enabling and disabling device 206, 230
  initializing device 211
  locator with pick and stroke devices 214
  querying 201
  segment transforms as locators
    GDDM-OS/2 Link 212
  triggering 203
locking and unlocking keyboard 315
  when screen partitioned 458
logical input device
  See choice, locator, pick, string, stroke
logical input devices 197
  associated with graphics field 214
  for GDDM-OS/2 Link 212
  querying 200, 215
long plots 436

M

manuals, list of xxvii
mapped alphanumerics
  Assembler variables for base attribute adjuncts 317
    ADMUAIMC. 317
  C/370 variables for base attribute adjuncts 317
    ADMUBIMC. 317
  COBOL variables for base attribute adjuncts 317
    ADMUCIMC. 317
  introduction 54
  PL/I variables for base attribute adjuncts 317
    ADMUPIMC. 317
mapping 283, 307
  See also alphanumerics, mapped
margins for FSLOG and FSLOGC 403
markers 30
  color 37
  set the type 36
matrix, transformation 152
  querying 156
  setting 154
MDT bit 317
menu
  graphical 198
  mapped 322
  procedural alphanumeric 257
merging images 97
message inserts 133
message segments, size of (IMS) 561
messages
  from WTP (write-to-programmer) 549
messages, error 10
MFS (message format service) 560
mix mode
  background color mixing 44
  foreground 45, 164
    3270-PC/G and /GX 49
    changing inside an area 45
    changing priorities 164
  foreground color mixing 40
mixing colors
  on plotters 448
mixing DBCS with SBCS characters
  device support for
    querying 396
mixing foreground colors 40
  primitive
    foreground overlapping other primitive(s) 40
mixing graphics and alphanumerics 80
mixing images 97
MIXSOSI default option 246
mnemonic for color codes 75
mode of graphics text 58, 64
mode-1 graphics text 59, 60
  advantages and disadvantages 61
  on 3279 69
mode-2 graphics text 60
  advantages and disadvantages 61
mode-3 graphics text 60
  advantages and disadvantages 61
modified fields
  mapped data 317
  procedural 261
    processing 263
    querying 261
    resetting field status 263
module, defaults 135, 375
moving segment origin 158
moving segments 149
moving the current position (GSMOVE) 28
  inside an area 31
MSCPOS (set cursor position) 318
MSDFLD (create or delete a mapped field) 289
MSGET (retrieve data from a map) 289
  setting adjuncts 310
MSPCRT (create a page for mapping) 111, 289
  effect on cell size of 3290 475

MSPUT (place data into a mapped field) 289, 291, 310, 311
MSQMOD (query modified fields) 302
MSQPOS (query cursor position) 320
MSREAD (present mapped data) 286
  partitions 458
multi-task windowing 494
multicolored
  graphics images 33
  image symbols 242
  markers 37
  shading patterns 40
multiline procedural alphanumeric fields 72
multipart graphics area 31
multiple instances of GDDM 531
multiple instances of GDDM and GDDM-REXX 21
multiple markers 30
multiple pictures 115
multitask windowing 479
MVS Batch 550
MVS/ESA 550, 556, 583
MVS/XA
  application interface
    user exits 551, 584
    user fast option 141

N

name
  symbol set 235
NAME nickname parameter 375
name of device 373
  in nickname statement 375
named segments 145
namelist.
  under CMS 373
  under TSO 373
national use characters 242
native CMS files 531
negate the pixels of an extracted image (IMRNEG) 102
neutral color 35
new function
  Version 3 Release 1 xxix
nicknames 374
  query content
    ESQUNS 379
  query length of
    ESQUNL 379
  query using ESQUNS 376
  sending output to plotter 445
  simplifying DSOPEN 374
  spooling print files under CMS 401, 418
NOEDIT mode under TSO 547
non-display field attribute
  See alphanumerics
non-GDDM device interrupt handling 536
nonchained segment attribute 148
nonqueriable APL displays and printers
  TSO 548
  VM/CMS 537
nonreentrant interface 4
  CICS
    ADMUOFF 577
nonreentrant interface for CICS 4
nonretained mode 224
null-to-blank conversion 75
number of copies to printer 403
numeric input fields
  See also alphanumerics
  mapped 317
  procedural alphanumeric 72

O

object code page 252
object import/export utility (IMS) 564
objects
  GDDM, code-page conversion 252
open a device 371
open graphics segment 145
operator window 480
  active 480, 484, 490
  application group 498
  attribute
    modifying 490
  attributes
    defaults 484
    querying 494
  candidate 484, 489
  compared with partitions 481
  coordinating device 480
  creating 484
    default 480
  current 484, 489, 490
  deleting 485
  DSOPEN use 480
  identifier 491
    default window 484
    querying 493
    use of –1 492, 493
  multitasking 494
  priorities
    changing 486
  priority 481
    changing 481, 491
    querying 492
  reference 492
  user control 481, 486
  viewing order (priorities) 489
  viewing order (priority) 490
  virtual device 489
    interrupts 490

operator window *(continued)*
virtual devices 480
virtual screen 480
operator window viewing priorities
query
WSQWP 492
set
WSSWP 491
option group 373
options list
for device processing 373
OR, exclusive, drawing mode 220
orient extracted image (IMRORN) 101
oriental languages 66
orientation of plotter picture 437
origin of segment
See graphics segments
outline of graphics area 31
outlining fields 270
output
family 372
graphics 173
storing 173
oval displayed instead of circle 28
overlap
of image symbols 65
overlapping multiple pictures 116
overlays 409
overpainting 164
on plotters 448

P

PA keys
enabling as logical input devices 207, 230
translation into alphanumeric input 326
use under CICS
restriction 577
use under CMS 533
processing option for 534
use under TSO 545
PA1 usage
under CMS 534
under TSO 545
page 111
GDDM 9
select 123
mapped 289
page printers 404
page segments 409
large, for 4250, under VSE 596
Page segments (PSEGs) 409
panning and zooming
overview 224
using GSSAVE and GSLOAD 189
paper size, plotter 435
parameter checking
bypassing for performance 140
parameters 18
array 17, 19
constant 80
partition sets 109, 453
partitions 109, 453, 473, 477
and interactive graphics 229
device considerations 396
programming example 454, 477
pattern sets
require storage on 3270-PC/G and /GX 49
samples provided with GDDM 39
patterns 37
PCB (program communication block) 554
pel
See pixel
pen plotters 433
pen-detectable field attribute 317
pen-detectable fields 264
pen-enterable fields 265
pens in plotter
numbers and colors 446
pressure 434, 451
velocity 451
changing 434
control by operator 437
width 434
performance improvement
image programs 368
procedural alphanumerics 270
PF keys
enabling as logical input devices 207, 230
translation into alphanumeric input 326
pick input
altering priorities 164
associated with graphics field 214
compared with GSCORR 226
enabling and disabling device 206, 230
example 198
initializing device 211, 213
pick aperture 199, 213
pick with locator and stroke devices 214
querying 200
segment-picking example 215
triggering 203
picture all in one segment 163
picture complexity 396
checking 397
picture drawing defaults 47
picture interchange format (PIF) 174, 192
picture space 113
and GSLOAD 180
PIF (picture interchange format) files 174, 192, 252

pixel 32
3270-PC/G and /GX 220
image symbols 233
images 32
line width 36
plotter 438
shading patterns 38
PL/I
ADMUPIMC 317
compiling and executing a program 14
mapped alphanumerics 288, 295
declaration of GDDM entry points 11
error exits 138
placing an image (IMRPL) 97
plot files
IBM GL 437, 446
CICS restriction 567
IMS restriction 553
plotter cells 438
plotters 433
alphanumerics not supported 54
roll-feed 436
user pattern sets not supported 51
using symbol sets 256
workstation-attached
as auxiliary device 433
plotting
GDDM API 371
long plots 436
on auxiliary device 433
plotting area
size
control by operator 435
polyfillet call 29
polyline call 29
polyline input 204
polylocator input 204
polymarker input 204
positioning an image (IMRPL) 97
positioning segments 149
PostScript
example 408
PostScript printers 404
pound sign 242
precedence of alphanumerics over graphics 61
prefixed variables 17
presentation area of a map 292
primary colors 40
primary data stream for CDPF, PSF, and PostScript 408
primary device 382
primitive, graphics
See also graphics
of graphics 25
print file
family-4
GOCA 407
print file *(continued)*
family-4 *(continued)*
IOCA 407
PTOCA 407
print utility
for GDDM files 417
on CICS 582
for non-GDDM files 421
print-control options group 402
printer
as a primary device 401
as an alternate device 412
CDPF attached 404
header pages 403
page 404
page size 401
plotter output 445
PostScript 404
processing under VM/CMS 537
PSF attached 404
queued 402
rightmost columns in black and white 401
SCS under IMS 560
system 404
ways of using 399
printing
composite documents
CDPU call 419
device 420
family-4
PostScript 408
truncated viewport 393
GDDM API 371
images
on 4028 366
overview 399
PostScript
example 408
printing images 366
priority of segments and primitives 164
after GSLOAD 180
procedural alphanumerics 71
introduction 54
processing modified fields 261
symbol sets for
default (hardware) symbols 239
processing options
list in DSOPEN 373
plotters 434
retained and nonretained modes 224
PROCOPT nickname parameter 375
production programs
improved performance for 140
PROFILE WTPMSG 549
program communication block (PCB) 554

program specification block (PSB) 557
programmed symbols
  See also PS
  loading symbol sets 397
programming example
  AID translation 327
  alphanumeric menu 257
  graphics text 62
  printing composite documents 419
  system printer 404
  windowing (two windows) 485
programming examples
  4250 fonts 426
  advanced-function printers 404
  breaking a line of graphics text 62
  color adjunct 327
  color masters 430
  concatenating graphics text 65
  copying screen output to a printer 416
  cursor adjunct 319, 321
  cursor selection 322
  data entry 454
  directly-attached printer as primary device 401
  dummy devices 388
  GDDM-REXX 508
  graphics and mapping 333
  graphics and procedural alphanumerics 80
  graphics image 32
  graphics-text attributes 58
  high-performance alphanumerics 274
  HPA 274
  image printing on 4224 367
  image scaling to fit display screen 346
  interactive image trimming 351
  interactive image trimming with part-screen image field 354
  inverting graphics windows 119
  light pen selection 322
  mapped menu 322
  multipart graphics area 31
  opening a device 372
  partitions 454, 477
  PF key selection from menu 322
  plotting a saved picture 439
  procedural alphanumerics 77
  querying graphics attributes 46
  queued printer 402
  redefining graphics windows and viewports 121
  scrolling 477
  selector adjunct 308
  subroutine to draw at specified location 46
  symbol set attributes 240
  symbol set for graphics text 237
  the 64-color set 39
  two primary devices 384
  Using GDDM graphics calls 25
programming examples *(continued)*
  viewports 115
  windowing (one window) 481
programming languages supported 3
projections, image
  See image, projection
proportionally spaced symbols 68
  printing 413
proportions of picture, correcting 28
protected attribute, unexpected 305
protected fields
  See also alphanumerics
  mapped 315
  procedural alphanumeric 72
prototyping 23
PS (programmed symbol) 233
  adjunct 327
  operator windows 497
  store 242
PS overflow 397
PS stores
  complex pictures 396
  exceeded 397
  graphics 397
PS/55 Workstation 265
PSB (program specification block) 557
PSDSS (load a symbol set into a PS store from the application program) 243
PSEGxxxx files 409
PSLSS (load a symbol set into a PS store from auxiliary storage) 238
  workstations
    3270-PC/G and /GX 255
    supported by GDDM-OS/2 Link 255
    supported by GDDM-PCLK 255
PSLSSC (conditionally load a symbol set into a PS store from auxiliary storage) 243
PSQSS (query status of device stores) 254
PSRSS (release a symbol set from a PS store) 254
PSRSV (reserving or releasing a PS store) 254
PTNCRT (create a partition) 109, 457
PTNDEL (delete a partition) 472
PTNMOD (modify the current partition) 472
PTNQRY (query the current partition) 459, 472
PTNQUN (query unique partition identifier) 472
PTNSEL (select a partition) 457
  making partition active 459
PTOCA 407
PTSCRT (create a partition set) 109, 453
PTSDEL (delete a partition set) 472
PTSQPN (query partition numbers) 472
PTSQRY (query partition set attributes) 472
PTSQUN (query unique partition-set identifier) 472
PTSSEL (select a partition set) 457
publications, list of xxvii

pushing/popping attribute values 46
putting data to an image
(IMAPTS,IMAPT,IMAPTE) 356

Q

quality control of images (ISCTL, ISXCTL) 359
quasi-reentrancy
reentrant interface for CICS 4
query
all segments 165
attributes of an image (IMAQRY) 93
character attributes 77
character box 65
current color 46
current line width 46
current operator window
WSQRY 494
current partition 459
current position 32
cursor position
mapped alphanumerics 320
procedural alphanumerics 73
device 373, 392
device characteristics (FSQURY) 339, 392
graphics attributes 46
image box cursor (ISQBOX) 349
image compressions supported by the device
(ISQCOM) 344
image field (ISQFLD) 366
image formats supported by the device
(ISQFOR) 343
image locator cursor position (ISQLOC) 349
image scanner device (ISQSCA) 339
last error 133
logical input device 200, 215
logical input devices
choice 202
locator 201
pick 200
string 204
stroke 204
mapping calls 306
modified procedural fields 261
operator window attributes 494
operator window identifiers 493
WSQWI 493
operator window numbers
WSQWN 493
operator window priorities 492
operator window viewing priorities
WSQWP 492
partition 472
partition set 472
picture space 115
PS stores 254
query *(continued)*
segment origin 159
segment priority 164
supported image resolutions (ISQRES) 345
symbol set character attributes 242
transforms 156
unique partition set identifier 472
query calls 115
queue, graphics input 208
See also input
queued printer 402, 417
as a primary device 403
as an alternate device 412
send logging text to 414
quick-path tutorial of GDDM-IMD 285
quotes 20

R

rastering
when copying 421
RCP (request control parameter) 5
re-raster for different device 422
read screen contents
ASREAD 9
mapped pages 289
GSREAD 208
read symbol set into program 243
readability of graphics text 64
record
error 131
error-log 576
graphics input 208
record initialization 211
for logical device 211
rectangle displayed instead of square 28
redefining a graphics window or viewport 121
reducing segments 149
reentrant interface 4
reference operator window 492
reference point 222, 223
reflect extracted image (IMRREF) 102
refreshing the screen 171
ASREAD and FSFRCE 9
GSREAD 208
regeneration of screen 171
register 15, error code in 136
reject-type MSPUT 310, 311
See also alphanumerics, mapped
release symbol set 254
releases 1, 2, and 3: compatibility with release 1.4
GDF (graphics data format) 177
reply mode for operator (ASMODE) 77
request control parameter (RCP) 5
reserve a PS store 254

restore image from auxiliary storage (IMARST) 92
restore projection from auxiliary storage (IMPRST) 98
restricting level of messages displayed 137
retained/nonretained mode, 3270-PC/G and /GX workstations 224
retrieving alphanumeric data
  mapped 289
  procedural 72
retrieving graphics
  CGM 192
  GDF
    coordinates preserved 184
    picture maximized 185
    same size 187
retrieving graphics from ADMGDF files 180
retrieving graphics from CGM files 192
retrieving graphics from external storage 179
return codes 132
reverse video
  See also alphanumerics
  ASFHLT (define field highlighting) 75
  graphics text 44
  mapped data 327, 330
rewrite-type MSPUT 310
  See also alphanumerics, mapped
REXX
  See also GDDM-REXX
  trace facilities 134
roll-feed plotters 436
rotating
  graphics segments 149
  graphics text 66
rotating a plotter picture 437
RSCS (Remote Spooling Communication Subsystem) 418
running
  a GDDM program 14
  multiple instances of GDDM 531
  programs under CMS 530
  sample programs 527
running a GDDM program 529
  mapping 288, 295

S

sample programs
  C/370
    ADMUSB1 (line graph) 519
    ADMUSB2 (alphanumerics) 519
    ADMUSB3 (display line types, colors and patterns) 519
  COBOL
    ADMUSC1 (line graph) 519
    ADMUSC2 (alphanumerics) 519
  compiling 524
  description 519—521
sample programs *(continued)*
  FORTRAN
    ADMUSF1 (line graph) 519
    ADMUSF2 (alphanumerics) 519
  GDDM-REXX 519
  link-editing 525
  PL/I 521
    ADMUSP1 (line graph) 519
    ADMUSP2 (alphanumerics) 519
    ADMUSP3 (display line types, colors and patterns) 519
    ADMUSP4 (graphics editor) 519
    ADMUSP7 (CECP translation of chart objects) 519
    ADMUTMT (Task Manager for TSO) 519
    ADMUTMV (Task Manager for MVS) 519
  running 527
sampling mouse, puck, or stylus position 204, 213
save
  current page contents 176
  graphics 173
save graphics
  in CGM format 178
  in GDF format 176
save image on auxiliary storage (IMASAV) 91, 104
save projection on auxiliary storage (IMPSAV) 97
saved graphics
  loading into programs 179
  retrieving from external storage 179
    CGM 192
    GDF 180
    PIF 192
SBCS Japanese input
  extended code-page support
    translation 251
scale drawings
  inter-device copy 188
  plotting 443
scale extracted image (IMRSCL) 97
scaling segments 149
scanner
  See also image, scanner
  introduction to 85
scanning
  introduction to 85
scope of symbol sets 387
screen attribute byte 113
screen interrupt
  from partitioned screen 458
  from windowed device 490
  handling by GSREAD 208
screen layout
  See alphanumerics, high-performance
  See alphanumerics, mapped
screen partitions 453

screen redraw
effect on primitives outside segments 171
screen regeneration 171
SCRIPT/VS 596
scrolling 473
(see also panning)
programming example 477
SCS printers in IMS 560
secondary data stream
for CDPF or PSF
page segment (PSEG) 409
for PostScript
encapsulated PostScript (EPS) 409
secondary data stream for CDPF or PSF 409
segment
See also graphics segments
leaving open 123
origin
for segments on libraries 184
relation to graphics hierarchy 120
segment origin 150
See also graphics segment
moving 158
querying 159
segments
page, large, for 4250, under VSE 596
segments, graphics
See graphics segments
SEGSTORE processing option 225
selecting symbol sets by device type 395
selection from menu
See menu
selector adjunct 307
See also alphanumerics, mapped
selector input
See pick
selector pen feature 264
send output and await reply
ASREAD 9
GSREAD 208
MSREAD 286
send output to terminal
FSFRCE 10
mapped pages 289
send text to queued printer 414
sending picture to the device 9
sequence of pictures 10
sequential data sets
TSO 541
sequential non-GDDM files, printing 421
sessions for learning
GDDM-REXX parameters 18
multiple instances 21
set current resolution/scaling algorithm (IMRRAL) 102
set image quality-control parameters (ISCTL) 362
severity of error 132
shading algorithm 31
shading patterns 37
on workstations
GDDM-OS/2 Link 49
GDDM-PCLK 49
use on plotters 450
user-defined 37, 38
shearing
graphics segments 149
text and symbols 66
shift-in (SI) character 245, 267, 274
shift-out (SO) character 245, 267, 274
SI (shift-in) character 245, 267, 274
single-task windowing 480
size of graphics text 64
size of plot 435, 443
size of plotter paper 435
size of segments, changing 149
slide-show effect 10
SO (shift-out) character 245, 267, 274
sound terminal alarm 315
procedural call 80
source image, definition of 87
spacing
text and symbols 65
spill file
under VSE 597
spill files
family-4 printing 407
transformable segment 408
splitting the screen 453
spooling to printer 417, 418
square displayed as rectangle 28
square on screen for pick aperture 199
SSREAD (read a symbol set from auxiliary storage) 243
SSWRT (write a symbol set to auxiliary storage) 243
status
of alphanumeric field 263
of mapped field 317
storage exhausted, possible cause 7
storage factors 396
storage problems 22
stored image
definition of 86
naming of 86
storing graphics 173
storing/restoring attribute values 46
straight line 28
stream input 204
string input 204
associated with graphics field 214
effects on choice input 203
enabling and disabling device 206, 230
initializing device 211, 213

string input *(continued)*
  triggering 203
stroke input 204
  associated with graphics field 214
  effects on choice input 203
  enabling and disabling device 206, 230
  example 217
  initializing device 211, 213
  sampling method 213
  stroke with locator and pick devices 214
  triggering 203
substitution character
  mapgroup 304
  symbol set 242, 413
  symbol-set 395
suffix, device dependent
  See substitution character
supported programming languages 3
suppress
  warning messages 137
swathes 408
symbol set 233, 330
  See also alphanumerics
  ASFPSS (define primary symbol set for a field) 75
  attributes 236
  automatic load
    retrieving GDF 182
  character attributes 239, 240
  default 237
  field attributes 239, 240
  identifier 239
    saved in GDF 175
  loaded for alphanumerics 238
  loaded for graphics text 235
  mapped data 327, 330
  on plotters 450
  reading into program 243
  scope of 387
  scrolling 479
  selecting by device type 395
  selecting symbol sets by device type 395
  type 235, 253
  using PS with graphics 397
  variable cell size on 3290 474
    programming example 477
  writing to auxiliary storage 243
symbol sets
  3800 system printer 423
  4250 typographic fonts 424
symbol, national use 242
system markers 36
system patterns 37
system printer 404
system programmer interface 5
  dynamic load 557
    IMS 557
    restriction 540, 557
system programmer interface *(continued)*
  dynamic load *(continued)*
    TSO 539
  dynamic load of system programmer interface
  restriction on ADMUFO 540, 557

T

table for color-separation masters 429
tag, primitive 197
  See also graphics
tagging untagged primitives 182
target image, definition of 87
task management (windowing) 479, 494
temporary storage data sets 582
terminal interrupt
  from partitioned screen 458
  from windowed device 490
  handling by GSREAD 208
terminal processing, under TSO 544
terminating GDDM 7
termination 22
text
  See also graphics text
  introduction 53
  See also alphanumerics
text box 66
three-dimensional drawing 164
threshold, error 137
TOFAM nickname parameter 380
token, device
  See device
TONAME nickname parameter 380
trace
  on CMS
    CMSTRCE external default 135
    TSOTRCE 135
  on TSO
    TSOTRCE external default 135
trace all GDDM calls
  using FSEXIT 137
trace facilities
  GDDM 134
  GDDM-REXX 134
  REXX 134
tracing drawings 217
tracing GDDM calls
  using external defaults 134
trademarks xxiv
transaction processing (windowing) 485
transaction work area (TWA) 4
transfer data between two images, applying a projection (IMXFER) 90, 101, 106
transferring data from an image (IMAGTS,IMAGT,IMAGTE) 358

transferring data to an image
(IMAPTS,IMAPT,IMAPTE) 356
transferring pictures between systems and
devices 193
transformable segment
attribute 148
with family-4 spill file 408
transforming primitives
setting current transform 45, 159
transforming segments 149, 154
querying 156
transforms, image
See image, transform
transient data queues 580
translation tables for procedural alphanumerics 75
translation, AID 326
transmit output
ASREAD 9
ASREAD and FSFRCE
mapped pages 289
FSFRCE 10
GSREAD 208
mapped pages 289
transparency attribute 82
transporting picture 193
transporting pictures between devices and
systems 193
TRCESTR external default parameter 135
triggering input 203
trim an image down to the specified rectangle
(IMATRM) 105
TSO
ADMASXT (COBOL error-exit name) 139
Batch 549
DCB characteristics 542
direct access data sets 541
example of JCL 552
GDDM code above 16MB 551
NOEDIT mode 547
PA keys under 545
PROFILE WTPMSG 549
sequential data sets 541
using APL feature on nonqueriable displays 548
using GDDM 539
WTP (write-to-programmer) messages 549
TSOTRCE 135
turning (reorienting) an image (IMRORN) 101
tutorial, quick-path, of GDDM-IMD 285
TWA (transaction work area) 4
type-of-field attribute
mapping 317
procedural call (ASFTYP) 74

U

underpainting 41, 164
not supported on 3270-PC/G and /GX 49
on plotters 448
underscore
ASFHLT (define field highlighting) 75
mapped data 327
uniform graphics window, define (GSUWIN) 28, 118
unlocking and locking keyboard 315
when screen partitioned 458
unmodified fields
mapped data 317
procedural alphanumerics
querying 261
setting 263
unnamed segments 145, 149
unprotected field changed to protected 305
unprotected fields
See also alphanumerics
mapped 315
untagged primitives, tagging 182
updating the screen 171
ASREAD and FSFRCE 9
GSREAD 208
upside-down graphics text 119
usage of a device 382
user console 371
user control
of operator windows 481, 486
user exits 137
user fast option, ADMUFO 140
user response
handling by ASREAD 9
User-Control option 373
user-defined markers 36
user-defined patterns 38
using GDDM under TSO 539

V

variable cell size 474
programming example 477
variable data
with protected or autoskip attribute 315
variable data fields 283
Vector Symbol Editor 36, 233
vector symbol sets
default 248
vector symbols 61, 233
vector text 57
Version 3 Release 1, new function xxix
viewing composite documents 420
viewport 114
See also window, graphics

virtual device (windowing) 480, 489
virtual screen (windowing) 480
visibility segment attribute 148
  with family-4 spill file 408
VM/CMS
  compiling and running a GDDM program 14, 529
  GLOBAL commands needed for GDDM 529
  native files 531
  non-GDDM device interrupt handling 536
  using APL feature on nonqueriable printers 537
  using GDDM under VM/CMS 529
VM/XA 538
VSAM ESDS files 597
VSAM key-sequenced data sets (CICS) 578
VSE
  Batch 139
    ADMASXD (COBOL error-exit name) 139
  batch mode 595

W

width of graphics lines 36
  on plotters 449
window
  See also operator window
  operator 480
window for scrolling 473
window, graphics 28, 118
  (see also viewport)
  and GSLOAD 180
  clipping 125
  enlarging to shrink graphics 387
  for graphics libraries 184
  in graphics libraries 189
  inverting 119
  using points outside 125
workstations
  3270-PC/G and /GX 255
  supported by GDDM-OS/2 Link 49
  supported by GDDM-PCLK 49
    color mixing 49
    colors 49
    shading patterns 49
world coordinates
  See window, graphics
wrap-around procedural alphanumeric fields 72
write symbol set to auxiliary storage 243
write-type MSPUT 310
  See also alphanumerics, mapped
WSCRT (create an operator window) 484
WSDEL (delete operator window) 485
WSIO (windowed device input/output) 490
WSMOD (modify current operator window) 490
WSQRY (query current operator window) 494
WSQWI (query operator-window identifiers) 493
WSQWN (query number of operator windows) 493
WSQWP (query operator-window viewing priorities) 492
WSSEL (select an operator window) 489
WSSWP (set operator-window viewing priorities) 491

Z

zooming
  overview 224
  using GSSAVE and GSLOAD 189

Sending your comments to IBM

GDDM

Base Application Programming Guide

SC33-0867-01

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, use the Readers' Comment Form
- By fax:
  - From outside the U.K., after your international access code use 44 1962 870229
  - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: WINVMD(IDRCF)
  - Internet: idrcf@winvmd.vnet.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Readers' Comments

GDDM

Base Application Programming Guide

SC33-0867-01

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name	Address
Company or Organization	
Telephone	Email

GDDM Base Application Programming Guide SC33-0867-01

You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States of America
Belgium	France	Israel	New Zealand	Spain	
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name _____

Company or Organization _____

Address _____

_____

EMAIL _____

Telephone _____

1 Cut along this line

4 Fasten here with adhesive tape

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC33-0867-01