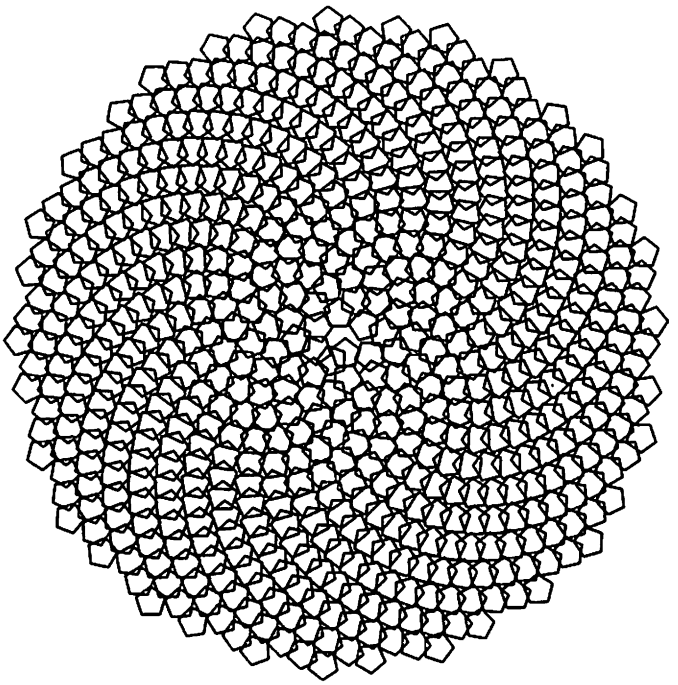
GDDM

Application Programming Guide Volume 1

IBM



Front Cover Pattern: Electronic Sunflower

The pattern on the front and back cover was produced using this GDDM program.

```
      INTEGER TYPE, VAL, COUNT, N, M
      REAL A1, A2, K1, K2, R1, R2, X, Y
      REAL XCEN, YCEN, XS, YS
      K1=5.3333
      K2=1.1
      R1=2
      XCEN=50
      YCEN=50
      CALL FSINIT
      CALL GSPS(1.0,1.0)
      K2=1.1*SQRT(2.4/K1)
      A2=0
      DO 40 M=1, 600
        A2=A2+K1
        R2=K2*(A2**.5)
        XS=R2*COS(A2)+XCEN
        YS=R2*SIN(A2)+YCEN
        DO 30 N=0, 5
          A1=2.*3.142*(FLOAT(N)/5.)+A2
          X=R1*COS(A1)+XS
          Y=R1*SIN(A1)+YS
          IF (N) 20,10,20
10        CALL GSMOVE(X,Y)
20        CALL GSLINE(X,Y)
30      CONTINUE
40    CONTINUE
      CALL ASREAD (TYPE,VAL,COUNT)
      CALL FSTERM
      END
```

SC33-0337-0
File No. S370/4300-40

GDDM

Application Programming Guide

Program Numbers
GDDM/MVS 5665-356
GDDM/VM 5664-200
GDDM/VSE 5666-328
GDDM-PGF 5668-812

Version 2 Release 1

Licensed Programs

Volume 1

IBM

First Edition (September 1986)

This edition applies to Version 2, Release 1, Modification 0 of the following members of the IBM GDDM Series of licensed programs:

GDDM/MVS 5665-356
GDDM/VM 5664-200
GDDM/VSE 5666-328
GDDM Interactive Map Definition 5668-801
GDDM-PGF 5668-812

Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

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 licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

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 is provided at the back of this publication. If the form has been removed, comments may be addressed either to:

International Business Machines Corporation, Department 6R1H,
180 Kost Road, Mechanicsburg, PA. 17055, U.S.A.

or to:

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

What this book is about

The GDDM Application Programming Guide introduces the application programming interfaces of GDDM, the Graphical Data Display Manager.

Who this book is for

This book is for application designers and programmers who are experienced in the following areas:

- Application programming in the language in which the GDDM programs are to be written. For example:

  COBOL
  FORTRAN
  PL/I
  SYSTEM/370 Assembler

- The subsystem under which the GDDM programs are to run. For example:

  CICS/VS
  IMS/VS
  MVS/TSO
  CMS subsystem of VM/SP

- The information contained in *GDDM General Information*, GC33-0319

How to use this book

This Application Programming Guide is in two volumes.

This first volume introduces the Base application programming interface of GDDM.

The second volume introduces the Presentation Graphics Facility application programming interface of GDDM.

You can read the chapters of each volume sequentially, or just read those chapters that concern you. The structure of both volumes is shown on the following page, and detailed in the table of contents of each volume. The last part of each volume is devoted to complete example programs. There is an index at the back of each volume, that you can use for reference.

The GDDM library

Introduction

General Information

GBOF-0058*

*Includes the GDDM brochures. For the General Information manual only, use order number GC33-0319

Release Guide

GC33-0320

General

Installation and System Management for MVS
SC33-0321

Installation and System Management for VM
SC33-0323

Installation and System Management for VSE
SC33-0322

Performance Guide

SC33-0324

Messages

SC33-0325

Diagnosis and Problem Determination Guide

SC33-0326

Programming

Application Programming Guide

(Two volumes)

SC33-0337

this book

Base Programming Reference

(Two volumes)

SC33-0332

GDDM-PGF Programming Reference

SC33-0333

Base Programming Summary (Booklet)

SX33-6053

GDDM-PGF Programming Summary (Booklet)

SX33-6054

User's Guides

Guide for Users

SC33-0327

Interactive Chart Utility (ICU)

SC33-0328

Image Symbol Editor

SC33-0329

Vector Symbol Editor

SC33-0330

Interactive Map Definition (GDDM-IMD)

SC33-0338

Books from related libraries

The Graphics Control Program (GCP), which controls the 3270-PC/G and /GX work stations, is described in:

GCP Work Station Programmer's Guide and Reference, SC33-0208.

The Composed Document Printing Facility (CDPF), a prerequisite IBM program product if you use the IBM 4250 printer, is introduced in:

Composed Document Printing Facility General Information, GC33-6133.

Fonts and code pages for the IBM 4250 Printer are illustrated in:

IBM 4250 Printer Type Font Catalog, G520-0004.

The Print Services Facility (PSF), an IBM program product that directs output to the IBM 3800 Printing Subsystem Models 3 and 8, is introduced in:

IBM 3800 Printing Subsystem Model 3 Introduction, GA32-0049

Fonts for the 3800 Model 3 are illustrated in:

IBM 3800 Printing Subsystem Model 3 Font Catalog, SH35-0053

and for the 3800 Model 8 in:

IBM 3800 Printing Subsystem Model 8 Font Catalog, SH35-0054

The Document Composition Facility (DCF), which handles GDDM-created page segments for composed page printers, is introduced in:

Document Composition Facility and Document Library Facility General Information Manual, GH20-9158.

The DCF Script/VS language, and other DCF functions, such as the Font Library Index Program, are described in:

Document Composition Facility Script/VS Language Reference, SH35-0070.

Book structure

VOLUME 1. Base facilities (this volume)

Part 1. GDDM basics . . . pages 3 through 123
Describes basic graphics and alphanumeric functions and the GDDM hierarchy of concepts. This part tells you how to write and run simple GDDM programs, generally of the menu-driven output graphics type.

Part 2. Advanced graphics . . . pages 125 through 215
Describes advanced graphics functions, including interactive graphics. This part tells you how to structure and store your graphics data, and how to use logical input devices to make your graphics interactive, without making them device-dependent.

Part 3. Advanced text . . . pages 217 through 302
Describes the remainder of the text functions. These include symbol sets, advanced procedural alphanumerics, and mapping.

Part 4. Image processing . . . pages 303 through 364
Introduces the principles behind image hardware, image data, and image processing. This part shows you, through simple example programs, how to capture, manipulate, save, restore, display, and print images.

Part 5. Device support, printing, plotting and windowing . . . pages 365 through 485
Describes device support, printing, and plotting. This part also shows you ways to split a terminal screen into a number of separate logical areas, how to prioritize those areas, and how the end user can interact with those areas.

Part 6. Example programs . . . pages 487 through 504

Appendixes . . . pages 507 through 516

Glossary of GDDM terms . . . pages 517 through 529

Index to Volume 1 . . . page 531 onward.

VOLUME 2. Presentation Graphics Facility

Part 1. Business charts
Describes two ways of producing business charts from an application program.

Part 2. Example programs

Glossary of GDDM terms

Index to Volume 2

Contents

Figures

Summary of amendments

Changes to this manual for Version 2 Release 1

Numerous changes in organization and scope have been made, particularly:

- The guide has been divided into two volumes. See "Book structure" on page vi.
- An appendix listing all GDDM calls has been removed. All the calls in the Base API are listed and explained in the *GDDM Base Programming Reference.* All the calls in the PGF API are listed and explained in the *GDDM-PGF Programming Reference.* Also, all the calls that are covered in the guide are listed in the index at the back of each volume.

The information in this volume has been changed to reflect the introduction of Version 2 Release 1 of GDDM:

- Support for the following devices:
  - 3193 Display Station and 3117 and 3118 Scanners
  - 4224 Printer
  - 5080 Graphics System
  - 6180 Plotter.
- Withdrawn support for the 3277GA terminal.
- Extension of the Base application programming interface (API) to support the input, output, storage, and manipulation of images.
- A wide range of improvements to the Base graphics API.
- More flexible partition support.
- Support for operator windowing. Several applications can share a screen, each one running in its own window. Also, a single application can use several operator windows of its own.
- A call and processing options for handling user control.
- A trace facility for API calls and internal GDDM processing.
- Better operating characteristics on 5550 devices.

Compatibility of Version 2 Release 1 with earlier releases

Programs that were written for Version 1, Releases 1 or 2 will execute on Version 2, Release 1, but will need to be link-edited again if they were not link-edited under Version 1, Releases 3 or 4. Programs written for Version 1, Releases 3 or 4 do not need to be link-edited again. Data streams, chart formats and data, symbol sets, and map groups created under earlier releases can be used with Version 2 Release 1.

Incompatibilities

- Programs that attempt to open a 3277GA terminal will fail because the 3277GA is no longer supported.
- A parameter value of zero on the GSCB or GSMB call will cause the current default value to be used. With previous releases, the dimension was reduced to zero.
- Segment transformations are now honored on family-4 (composed-page) printers, even when a spill file is used. With previous releases, they were ignored.
- NATLANG=K now means Kanji rather than Katakana. On terminals that do not support double-byte character sets, US English will be used instead of Kanji.
- The local mode of operation that was previously available on work stations has been dropped. Its functions have been taken over by user control. If the terminal user presses PA3, then user control will be offered by default instead of local mode. The LCLMODE option now affects only the way panning and zooming is implemented.
- The CHART call is no longer affected by preceding PG routine calls. The exception is CHAREA, which you can use to define the area in which the ICU constructs the chart.
- With the ICU, if the chart area is altered, the size of vector markers alters proportionately. Previously, the size of markers was independent of the chart area.
- The appearance of legends in charts created under the current release of PGF differs from charts created under Version 1 Release 1.
- Print files from earlier releases cannot be processed by Version 2 Release 1 print utilities.
- Call format descriptor and APL request codes modules:

  These modules can no longer be referred to and loaded by name. The only method of accessing them is through the address obtained by a CALLINF external defaults option in a SPINIT call. The meaning of the RCPPPGF flag in the RCPPFLAG field of the call format descriptor has been changed. When set on, it indicates that the call is not available in the GDDM Base programs, instead of indicating that it is available in the Presentation Graphics Feature. The name of the flag has been changed to RCPPOGP.

Changes to this manual for Version 1 Release 4

The guide was changed to reflect the introduction of the following facilities in Version 1 Release 4 of GDDM:

- Support for the following devices:
  - IBM 3179 Models G1 and G2 Color Graphics Display Station
  - IBM 3270 Personal Computer/G and /GX (3270-PC/G and /GX) Work Stations
  - Plotters attached to the IEEE-488 port of a 3270-PC/G or /GX
  - IBM 3800-3 Printing Subsystem Models 3 and 8
  - IBM 5550 Multistation (including support for Kanji alphanumerics).
- String and stroke graphics input devices
- Transformation, copying, and priority control of graphics segments
- Storing graphics segments on external storage and retrieving them
- Uniform graphics window coordinates
- Explicit correlation of graphics segments and primitives
- Kanji graphics text (as well as alphanumerics on the 5550)
- Nicknames to increase the flexibility of device support
- Copying to family-1 and -3 devices (in addition to family-2)
- Fonts and codepages for the IBM 4250 Printer
- Improved printer spooling
- Printing non-graphics data with the GDDM Print Utility
- Tower charts
- Polar charts
- Exploded and three-dimensional pie charts
- Bar charts with numeric axes, hidden bars, and extended bar labeling functions
- Support for missing values in charts
- More flexibility in chart labeling, markers, shading, and outlining.

In addition, numerous changes in organization and scope were made, particularly:

- The book was divided into six parts: a primer, followed by five parts devoted to particular functional areas.
- A chapter giving a short overview of all the text facilities was added.
- A chapter introducing the graphics data format (GDF) was added.
- An appendix summarizing the major types of supported device was added.
- A heading indicating the devices covered was added to each page.

Compatibility of Version 1 Release 4 with earlier releases

Application programs written for use with earlier releases of GDDM and PGF will run under Version 1 Release 4 without modification. They will need link-editing again if they were link-edited under Version 1 Release 1 or 2, but not if they were link-edited under Version Release 3. Data streams, chart formats and data, symbol sets, and map groups created under earlier releases, can be used with Version 1 Release 4.

Print files from earlier releases cannot be processed by Version 1 Release 4 print utilities. Data streams, chart formats and data, and vector symbol sets created under Version 1 Release 4 cannot be used with earlier releases.

Changes to this manual for Version 1 Release 3

The guide was changed to reflect the introduction of the following facilities in Version 1 Release 3 of GDDM:

- Support for the following devices:
  - IBM 3277 Graphics Attachment RPQ (3277GA).
  - IBM 3290 Information Panel.
  - IBM 4250 Composed Page Printer.
- Alphanumeric mapping and the Interactive Map Definition utility (GDDM-IMD)
- Partitioning the screen and scrolling
- Interactive graphics
- Segment attributes
- Primitives outside segments
- Support for color separation masters for in-house printing
- Proportionally spaced graphics text
- Scaled images (GSIMGS call)
- Vector symbol markers and scaling of them (GSMSC call)
- Fractional line widths (GSFLW call)
- Line-width table for PGF charts (CHLW call)
- A call (FSSHOR) that provides similar function to FSSHOW, but also returns some input data
- New fields in the Interactive Chart Utility (ICU) call parameter to support new ICU function
- New libraries of PL/I declarations.

In addition, numerous editorial improvements were made, particularly:

- A COBOL example was added.
- Constant parameters to GDDM calls were presented in a way that shows whether they are fixed or floating point.
- An appendix listing all GDDM calls was added.

Compatibility of Version 1 Release 3 with earlier releases

Application programs written for use with earlier releases of GDDM and PGF will run under Version 1 Release 3 without modification, but they must be link-edited again. Data streams, chart formats and data, and symbol sets created under earlier releases can be used with Version 1 Release 3.

Print files from earlier releases cannot be processed by Version 1 Release 3 print utilities. Data streams, chart formats and data, and vector symbol sets created under Version 1 Release 3 cannot be used with earlier releases.

Volume 1. Base facilities

Part 1. GDDM basics

Chapter 1. Introduction

What this volume describes

GDDM is a family of IBM program products 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 general graphics, alphanumerics, and image, or **base** facilities are introduced in Parts 1 to 4 of this volume. Part 3 of this volume also contains some guidance on the optional GDDM Interactive Map Definition (GDDM-IMD) product, which you can use in conjunction with some of the alphanumeric facilities of GDDM Base.

The **Presentation Graphics Facility (PGF)** is a product that you use to create **business graphics**, for example, line graphs or pie charts. An important part of PGF is the **Interactive Chart Utility (ICU)**, which allows business charts to be drawn on a display screen by people with no programming knowledge. The PGF and ICU are introduced in Volume 2 of this guide.

The GDDM application programming interface

All the base and PGF facilities are accessed by means of a call-type application programming interface (API).

This guide is an introduction to GDDM, rather than a comprehensive reference document. The *GDDM Base Programming Reference, Volume 1* and *GDDM-PGF Programming Reference* manuals have complete descriptions of all the calls and their parameters.

Most of the examples given in the text are coded in PL/I, but the GDDM calls are similar in the other supported languages – COBOL, FORTRAN, and System/370 Assembler. For example, these pairs of calls initialize GDDM and request a screen read:

```
PL/I:          CALL FSINIT;
               CALL ASREAD(TYPE,MOD,COUNT);
FORTRAN:       CALL FSINIT
               CALL ASREAD(TYPE,MOD,COUNT)

COBOL:         CALL 'FSINIT'.
               CALL 'ASREAD' USING TYPE, MOD, COUNT.

ASSEMBLER:     CALL FSINIT,(0),VL
               CALL ASREAD,(TYPE,MOD,COUNT),VL
```

Throughout this guide, floating-point constant parameters are shown with a decimal point (for instance: 3.0), and fixed point without (for instance: 3).

There is an Assembler example program in "Example 2. Assembler language example" on page 493, and a COBOL example in Volume 2.

APL and BASIC programs can also call GDDM routines. However, the support is provided by software associated with the languages, rather than by GDDM. There is an APL example program in "Example 3. An APL2 example" on page 495, and a BASIC one at "Example 4. BASIC example" on page 497. For further information, you will need to refer to the manuals describing this language-related software.

The examples are intended to illustrate particular points about GDDM, not necessarily to demonstrate good programming practice. For instance, a well-written real application program might test the return codes from every GDDM call and take special action to handle any errors. The examples do not in general do this because it would obscure the main points.

All the examples use the GDDM **non-reentrant interface**. Two less commonly used interfaces are available, the **reentrant interface** and the **system programmer interface**. These are fully documented in the *GDDM Base Programming Reference* manual.

Hardware and software

GDDM supports IBM 3179 Model G color display stations, IBM 3270 terminal-attached display units, including the 3270-PC/G and /GX family of work stations, 5550 family multistations, 5080 graphics systems, 3270 terminal-attached printers, IPDS printers, system printers, composed-page printers, and plotters. Overviews of the major types of device are given in Appendix A, "Major types of supported device" on page 507.

The examples and descriptions in this guide typically apply equally to IBM 3179 Model G color display stations (3179-G) or 3279 terminals running under the CMS subsystem of VM/SP, unless otherwise stated or implied. Where appropriate, device variations are listed at the end of chapters. Most of the examples would require little or no change to execute on other terminals and under one of the other supported subsystems, namely CICS/VS, IMS/VS, or TSO. Information about running under these subsystems is given in the *GDDM Base Programming Reference* manual.

All the color illustrations in both volumes of the *GDDM Application Programming Guide* were produced by GDDM programs.

Chapter 2. Drawing a simple picture

This chapter tells you how to write a program that draws a simple picture on the screen of any GDDM-supported graphics display terminal.

When drawing pictures, there are two main types of call to GDDM. One type requests the addition of a **graphics primitive**, such as a line or arc, to the picture:

```
CALL GSLINE(20.0,65.0);        /* Draw a line to (x=20,y=65)     */
```

To address points on the screen, GDDM uses a coordinate system of 0 through 100 in each direction, with the origin in the bottom left-hand corner, unless you specify a different system.

The other type of call changes the value of a **graphics attribute** such as color, line type, or line width:

```
CALL GSCOL(6);                 /* Change current color to yellow */
```

On color terminals this call causes all subsequently drawn primitives to appear in yellow, until the color is changed again. (On monochrome devices, the call has no effect.)

Figure 1 on page 8 shows a simple PL/I graphics program to draw a sketch of a house, complete with a dimension. The output of the program is shown in Figure 2 on page 11. If you like, when you have read the explanation of the calls in the program, you can copy it, and have a go at putting in the calls to draw some windows.

The program introduces several important GDDM calls and concepts. These will now be explained. The explanations refer to statements in the program that are identified by letter. The identifications look like this in the program:

```
/*A*/
```

```
SKETCH: PROC OPTIONS(MAIN);

DCL (TYPE,MOD,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             */

/*************************/
/* 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  */
                        /* (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     */
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, will be shaded  */
```

Figure 1 (Part 1 of 2). "Sketch" sample 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)  */ /*E*/
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(5.0,2.0,26,'All dimensions are in feet');
                         /* 26 characters at (x=5,y=2)          */
/*************************/
/* SEND PICTURE TO SCREEN */
/*************************/

CALL ASREAD(TYPE,MOD,COUNT); /* Send the picture to the screen */ /*F*/
                             /*  and await a response          */
CALL FSTERM;             /* Terminate GDDM                      */ /*G*/

%INCLUDE ADMUPINA;       /* GDDM Entry declarations             */ /*H*/
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

END SKETCH;
```

Figure 1 (Part 2 of 2). "Sketch" sample graphics program

Housekeeping: You are required or advised to put some housekeeping statements into any GDDM graphics program:

- The FSINIT call /*A*/ initializes GDDM and is compulsory. The FSTERM call /*G*/ at the end is advised, to free all the storage and other resources acquired by GDDM. Its omission may cause subsequent programs (or reruns of the same program) to fail through lack of storage.

- The GSSEG call /*B*/ creates a **graphics segment** and is recommended before any graphics primitives are drawn. A graphics segment is a logical grouping of primitives and of the attributes that determine their appearance. If you do not use GSSEG, for the 3179-G, 3270-PC/G and /GX family, and 5550 family, the primitives will be discarded after any local operation takes place at the device (for example, if the screen is scrolled or if a system-issued message is displayed).

- In PL/I, but not in other languages, each GDDM entry point (that is, each GDDM call-name) used by your program should be declared. The declarations should specify the data types of all parameters. A set of files is supplied with GDDM that contains these entry declarations for you to include in your programs. Each file has a name of the form ADMUPINx for the nonreentrant entry points (or ADMUPIRx for the reentrant entry points), and contains declarations for all the entry points starting with "x". The declarations necessary for the example are included at /*H*/. It is customary to include the files at the end of the program because they affect the line numbers of all subsequent statements.

Under CMS, if you include several files, you may exceed the external names limit. To avoid the problem, you can edit the files to remove names you do not use. Some of the examples in this guide may not work if you leave all the files unedited.

Default Attribute Values: All graphics attributes have default values initially, that is, when a segment is opened. You need to set a particular attribute only if you require a different value, as at `/*C*/`.

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

Graphics Text: GSCHAR at `/*E*/` produces **graphics text**. Such text is created from lines, arcs, areas, and dot images like the rest of the graphics. It should not be confused with alphanumerics (which is described in "Chapter 8. Basic alphanumerics" on page 75).

Output of the Picture: The picture gradually being built by the program is held inside GDDM. It is not transmitted to the screen until you issue a specific "send" command, most commonly a call to ASREAD as at `/*F*/`. The new (or modified) picture then appears on the display, and a screen "read" is issued. The terminal operator can reply to the read by causing an interrupt on the screen, for example, by pressing ENTER or a PF key. The three parameters of ASREAD will then be set by GDDM to indicate the type of response. Control will return to the program at the statement following the ASREAD. In the example, the type of response is not relevant; the program will terminate.

Pages: The picture is built up and stored by GDDM in a logical entity called a **page**. The example uses only one page, which GDDM created by default. A program can explicitly create and use multiple pages, although only one page is current at any one time. Graphics calls always apply to the current page. When the program executes an ASREAD call, the current page is sent to the terminal.

Data Types of GDDM Call Parameters: These are not necessarily apparent from the program. The parameters were mostly constants. Often the parameters will be variables and will have to be declared appropriately. These are the three PL/I data types used in GDDM call statements:

- FLOAT DECIMAL(6). This is used for all graphics calls specifying positioning of any sort. For example, the GSMOVE and GSLINE calls in the program had float-decimal parameters. The COBOL equivalent is COMPUTATIONAL-1; in FORTRAN it is REAL*4.

- FIXED BIN(31). This is used for all integer attribute and parameter settings. For example, the GSCOL and GSLW settings were fixed binary, as were the character string lengths. The COBOL equivalent is PICTURE S9(8); in FORTRAN it is INTEGER*4.

- CHARACTER. The data for text output is obviously in character form. The COBOL equivalent is PICTURE X(n); in FORTRAN, the equivalent is string literals or a numeric data array initialized with string literals.

The *GDDM Base Programming Reference, Volume 1* has a complete description of all the GDDM base calls and their parameters.

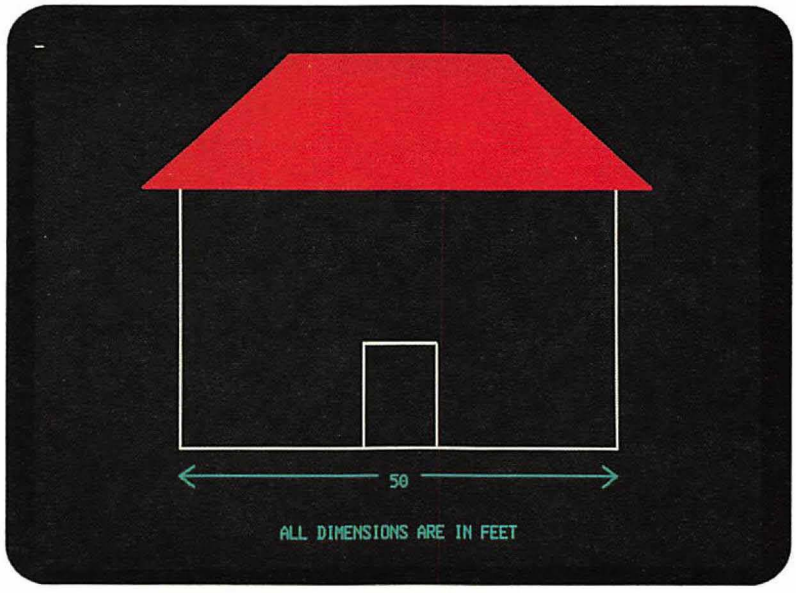


Figure 2. Output from "Sketch" sample graphics program

How to compile and run a GDDM Program under CMS

After creating a GDDM program, you will need to know what steps are required to run it. As an example, these are typical commands required to compile and execute a PL/I GDDM program under CMS:

```
CP LINK SYSTEM 2DD 2DD
ACCESS 2DD B
```

These two commands make the disk holding GDDM (2DD in the example) known to your virtual machine.

```
GLOBAL MACLIB ADMLIB
PLIOPT POST (INCLUDE FLAG(I)
```

The PLIOPT command invokes the PL/I Optimizing Compiler to compile the program. The INCLUDE option is required to pick up ADMUPINA, ADMUPINF, and ADMUPING, the declarations of the GDDM entry points. The macro library (ADMLIB) that contains these has been made known to CMS with a GLOBAL MACLIB command. The FLAG(I) option is not essential, but it ensures that useful messages about dummy variables are not suppressed. These are created when parameter attributes do not match GDDM's requirements.

```
GLOBAL TXTLIB ADMNLIB ADMPLIB ADMGLIB PLILIB
LOAD POST
START *
```

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. The LOAD command loads the program into storage and the START command starts execution. The picture of the house will appear on the screen that you use to invoke the program.

How to compile, link-edit, and run a GDDM program under TSO

Figure 3 shows a CLIST that you can use to compile, link-edit, and run a GDDM program under TSO. The PLI command invokes the PL/I optimizing compiler to compile the program.

```
/*************************************************************/
/* TEST(INCLCARD) CONTAINS THE FOLLOWING RECORD:             */
/*    INCLUDE INCLIB(ADMASNT)                                */
/* FOR USE WITH THE GDDM NON-REENTRANT INTERFACE.            */
/*                                                           */
/* REPLACE ADMASNT                                           */
/*      BY ADMASRT IF USING THE REENTRANT INTERFACE          */
/*   OR BY ADMASPT IF USING THE SYSTEM PROGRAMMER INTERFACE  */
/*   OR BY ADMASRT AND ADMASPT IF USING BOTH THE             */
/*   REENTRANT AND SYSTEM PROGRAMMER INTERFACES              */
/*************************************************************/
PROC 1 NAME
PLI TEST(POST) OBJECT(TEST(POST)) +
    LIB('GDDM.REL210.GDDMSAM') PRINT(*) INC FLAG(I)
ALLOC F(INCLIB) DA('GDDM.REL400.GDDMLOAD') REUSE SHR
LINK (TEST(POST), TEST(INCLCARD)) +
    LOAD(TEST(POST)) LIST PLIBASE PRINT(*)
CALL TEST(POST)
```

Figure 3. Sample TSO CLIST

The *GDDM Base Programming Reference* manual describes the steps required on other subsystems and using other languages.

Error handling

For reasons of clarity, the example 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 will reach the output statement (ASREAD in the example). This may or may not produce recognizable graphics, depending on the errors. The terminal operator will need 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 10. Debugging aids" on page 117.

Chapter 3. Basic input/output functions

This chapter discusses the following topics:

- The two basic output calls (ASREAD and FSFRCE)
- A call to check whether pictures are too complex to be displayed (FSCHEK)
- A **device-dependent** mechanism for saving pictures on auxiliary storage (FSSAVE) and redisplaying them later (FSSHOR).

GDDM maintains a record of the contents of each page. The program may change the contents, and so may the terminal operator, in ways to be described later.

Whenever the program issues an output call (ASREAD or FSFRCE), GDDM updates the screen so that it displays the alphanumeric and graphics contents of the current page. Some devices (the dual screen configuration of the 3270-PC/GX work station, or 5080 Graphics System, for example) have two screens, one for the graphics and the other for the alphanumerics. Whatever the type of screen, GDDM does not necessarily send the whole page – it sends only those parts that have been changed. When the target device is a printer, of course, the whole picture has to be sent.

Send output and await reply using call ASREAD

This is the basic call for sending out the current page. Other input/output calls for particular purposes will be introduced in later chapters. ASREAD requests a write-and-read operation: the current picture is sent to the screen and a response is awaited. The cursor is positioned in the top left-hand corner of the screen, unless otherwise specified by an ASFCUR call or set by the terminal operator in a previous interaction.

In other words, an ASREAD call requests that, after transmitting the data stream, GDDM should wait for the operator to reply before returning control to the program. This is the format of the call:

```
CALL ASREAD(TYPE,VALUE,COUNT);          /* Send output to device */
```

The parameters are set by GDDM to indicate the type of interrupt that was received. In the above example of an ASREAD call, the names of the parameters have been chosen to reflect their function, namely the type of interrupt, a value associated with the type, and the number of modified fields. Figure 4 on page 14 shows their possible values.

For interrupts of types 0-2, the last parameter indicates how many alphanumeric fields have been modified by the operator. For a discussion on how these fields are created and processed, see "Chapter 8. Basic alphanumerics" on page 75. The

handling of light-pen fields is covered in "Chapter 16. Advanced procedural alphanumerics" on page 235.

A returned type of 7 indicates that the read was performed on an output-only device such as a printer. In such circumstances, GDDM changes the write-and-read into a write only.

Interrupt	Type	Value	Count
ENTER key	0		number of modified fields
PF key	1	PF key number	number of modified fields
Light pen	2		number of modified LP-fields
Badge reader	3	0(valid), 1(not)	
PA key	4	PA key number	
CLEAR key	5		
Other	6		
Output – only device	7		
Mouse or puck button	10	Button number	

Figure 4. Parameters returned by ASREAD

In the special case of CICS pseudoconversational mode, the first ASREAD in all subsequent invocations of the pseudoconversation will perform **only** input – the output is suppressed.

See "Pseudoconversational programming under CICS" on page 391 for a description of this mode of programming, and the differences in effect of the various GDDM calls.

Transmitting output using call FSFRCE

When you want to update the screen without waiting for a reply, you must use the FSFRCE call instead of ASREAD. There are no parameters:

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

This causes all changes made to the current page since the last FSFRCE or ASREAD to be sent to the device.

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

Here is an example of how **not** to use FSFRCE:

```
CALL FSINIT;                          /* Initialize GDDM          */
CALL GSSEG(0);                        /* Open segment             */
CALL GSMOVE(25.0,60.0);               /* Start drawing the picture */
   .
   .
   .
/*******************************/
/* PICTURE SENT TO DEVICE..    */
/*******************************/
CALL FSFRCE;                          /* Send out the picture     */
/*******************************/
/* ..BUT DISAPPEARS IMMEDIATELY */
/*******************************/
CALL FSTERM;                          /* Terminate GDDM           */
```

If this program is run on CMS, for example, the graphics will appear on the screen for only a moment. Control will return to the program, the FSTERM will be executed and the program will terminate, returning control to CMS. To hold the picture on the screen, ASREAD must be used instead.

Checking picture complexity using call FSCHEK

Some pictures are too complex to be displayed at the terminal. The limits depend on the type of terminal. On a 3279, for instance, it is set by the availability of programmed symbol stores. On other types, it is set by the size of the buffer in which the terminal stores the vectors that comprise the picture. The size of the data stream may also limit picture complexity. More information is given in Appendix A, "Major types of supported device" on page 507.

Except on a 3179-G, 3270-PC/G or /GX work station, 5550 family multistation, and a 5080 Graphics System, a call to FSCHEK allows the program to determine whether the next output operation (typically an ASREAD or FSFRCE) would exceed any such limits:

```
CALL FSCHEK;          /* Determine whether overflow would occur */
```

This will return an error condition if the picture is too complex. To diagnose the error condition, the program can issue an FSQERR call. This call is described in "Chapter 10. Debugging aids" on page 117. For the moment, here 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 */
  .
  .
  .
END;
```

FSCHEK only checks the picture – it does not perform any output. A further call, such as ASREAD or FSFRCE, must be issued to send out the data stream.

On a 3270-PC/G or /GX, too-complex pictures are degraded by GDDM, as explained in "Retained and non-retained modes" on page 508. Therefore the FSCHEK call, although not invalid, never returns an error condition with these terminals.

Saving current page contents using call FSSAVE

With this call you can save the alphanumerics and graphics contained in the current page, or the alphanumerics and image contained in the current page. The saved picture may subsequently be redisplayed using FSSHOR. The format of the object saved is very similar to that of the eventual data stream. It is **device-dependent**. Other, device-independent, ways of saving graphics are described in "Chapter 11. Graphics segments" on page 127 and "Chapter 13. Picture handling in graphics data format" on page 171. Unless you require to save alphanumeric data (see "Chapter 6. Displaying text" on page 53) with your picture, **you are recommended to use these other methods**. Not only do they have the advantage of device-independence, but they allow you to manipulate, and add to, the saved picture.

Here is an example of FSSAVE:

```
CALL FSSAVE('DEMO8');     /* Save picture on auxiliary storage  */
```

The parameter is the name to be assigned to the picture when written to auxiliary storage. On CMS the full object name would become 'DEMO8 ADMSAVE A1'. On other subsystems, 'DEMO8' would be a member name in a library assigned for saved pictures.

If your picture is complex, you may get a diagnostic message saying that the object is too big to be saved. In that case you must reduce the complexity or the size of your picture, and then retry the FSSAVE.

The FSSAVE and FSSHOR calls are not supported when the picture has been created for a 5080, or a plotter, or a family-4 printer (printer families are explained in "Chapter 22. Using printers" on page 395).

Displaying a saved picture using call FSSHOR

With this call you can show a picture previously saved with an FSSAVE call.

```
CALL FSSHOR('DEMO8',TYPE,VALUE); /* Send saved picture to screen */
```

The saved picture will now appear on the display screen and remain there until the operator causes an interrupt (by pressing ENTER or a PF key, for example). Control will then return to the program, where normal processing of the current page may continue.

GDDM returns codes giving information about the interrupt in the second and third parameters. They have the same meanings as those returned in the first two parameters of ASREAD, as shown in Figure 4 on page 14. Data typed by the operator is not returned by FSSHOR.

The saved picture is not added to the previous graphics on the screen: it uses its own page and replaces the previous display. After the operator acknowledges the saved picture (by causing an interrupt), the program continues execution and the next ASREAD, FSFRCE, or FSSHOR will determine the screen contents to replace the saved picture.

If the picture was saved under Release 1 or 2 of GDDM, then the terminal operator will not be able to enter any data. This is because the Release 1 and 2 version of the FSSAVE call changed all unprotected fields to protected.

There is another call similar to FSSHOR, called FSSHOW. It differs in not returning any information about the interrupt generated by the operator:

```
CALL FSSHOW('DEMO8');              /* Send saved picture to screen */
```

Possible errors when showing saved pictures

- The FSSHOR device is not compatible with the device that was current when the FSSAVE was performed.
- The 3274 controller was configured for compressed data streams when the FSSAVE was performed, but is differently configured for the FSSHOR.
- Reference is made in the saved data stream to PS-stores that are either not present on the target device or have been reserved by the program.
- In the case of 3270-PC/G or /GX work stations, the FSSAVE and FSSHOR devices are compatible, but they have been customized differently – with different screen sizes, for instance.
- The 5080 graphics system does not support FSSAVE and FSSHOR.

Chapter 4. Graphics primitives

This chapter describes the GDDM calls that add graphics primitives (lines and arcs, for example) to your picture.

Coordinate system

You can define the (x,y) coordinate system that is used to address the drawing area. The coordinate system is known as the **graphics window**. When adding a primitive to your picture, you define locations such as the end of a line in terms of graphics window coordinates, which are also known as **world coordinates**. They are defined by this type of call:

```
CALL GSUWIN(0.0,200.0,0.0,100.0); /* Define coordinate system   */
                                  /* where x range is at least  */
                                  /* 0 to 200 (left to right)   */
                                  /* and y range is at least    */
                                  /* 0 to 100 (bottom to top)   */
```

If you are going to specify a graphics window, you must do so before opening a graphics segment or creating any graphics primitives.

If no window is explicitly defined, the default of exactly 0 through 100 in both directions applies. In this case, however, the coordinates may not be uniform: one x unit on the screen may not physically equal one y unit. This can lead to unexpected results, such as circles appearing as ovals and squares as rectangles. The GSUWIN call always creates a uniform set of coordinates. There is a full discussion of graphics windows in "The graphics window" on page 101.

The starting point for all primitives consisting of lines or arcs is the current position. The current position is the end point of the previous primitive, unless a GSMOVE has been issued.

Drawing a straight line using call GSLINE

This call draws a line from the current position to a specified end point, for example:

```
CALL GSLINE(25.0,90.0); /* Draw straight line to (X=25,Y=90)    */
```

The line is to be drawn in the current color, using the current line width and the current line type. The setting of such attributes is addressed in "Chapter 5. Graphics attributes" on page 35.

After this call the current position is (25,90) – the end point of the line.

Changing the current position using GSMOVE

This call is used to move to the required starting point of a new primitive. The format of the call is similar to that of GSLINE:

```
CALL GSMOVE(50.0,0.0); /* Change current position to (X=50,Y=0) */
```

The call is used whenever the end point of the previous primitive is not the required starting point of the next primitive.

Drawing a sequence of lines using GSPLNE

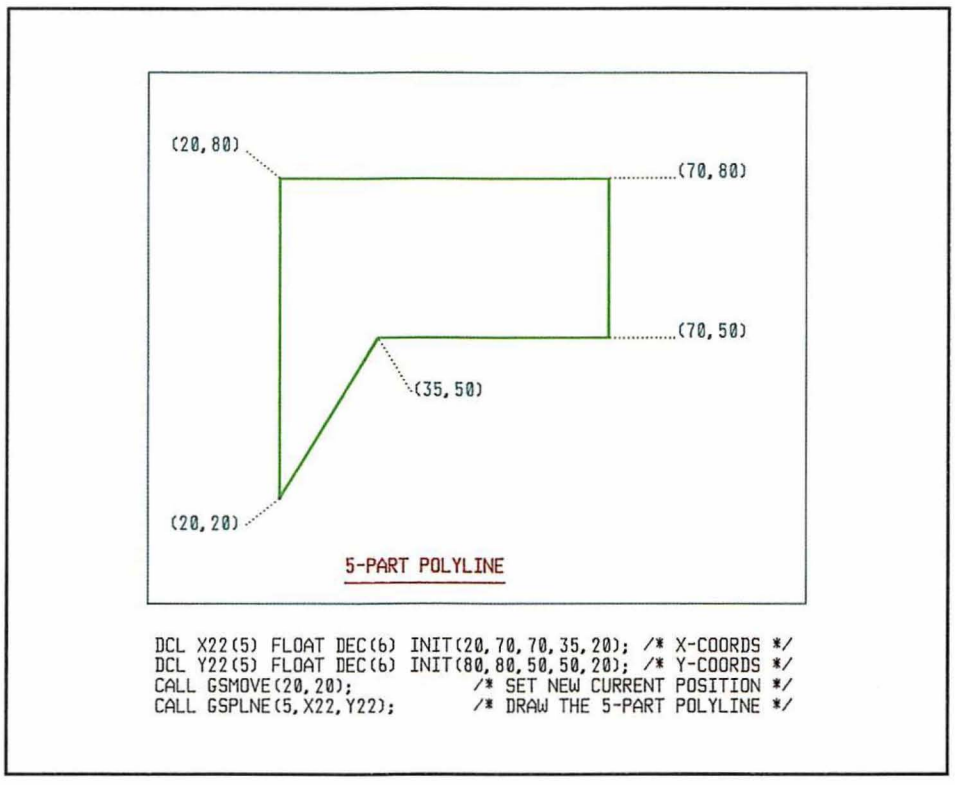
Rather than issue a series of GSLINE calls, the programmer can place the line-end coordinates in an array and issue a single call to draw the sequence of lines called a **polyline**. This is the format of the call:

```
DCL X22(5) FLOAT DEC(6) INIT(20.0,70.0,70.0,35.0,20.0);
                         /* x line-end coordinates              */
DCL Y22(5) FLOAT DEC(6) INIT(80.0,80.0,50.0,50.0,20.0);
                         /* y line-end coordinates              */

CALL GSMOVE(20.0,20.0);  /* Set current position.               */
CALL GSPLNE(5,X22,Y22);  /* Draw 5-part polyline.                  */
                         /* The first line will run from the       */
                         /* current position to (20,80), the second*/
                         /* from (20,80) to (70,80), and so on     */
```

As with most primitives, the first line of the polyline will start at the current position. The current position after the polyline has been drawn is, as you would expect, the end of the last line.

Figure 5 on page 21 shows the effect of the above GSPLNE call.




```
DCL X22(5) FLOAT DEC(6) INIT(20,70,70,35,20); /* X-COORDS */
DCL Y22(5) FLOAT DEC(6) INIT(80,80,50,50,20); /* Y-COORDS */
CALL GSMOVE(20,20);           /* SET NEW CURRENT POSITION */
CALL GSPLNE(5,X22,Y22);       /* DRAW THE 5-PART POLYLINE */
```

Figure 5. Drawing a polyline

Drawing a circular arc using call GSARC

This is one of several statements to draw arcs. The arcs will not appear circular on the screen unless you ensure that the window has uniform coordinates, as described in "Coordinate system" on page 19. This is the format of the call:

```
CALL GSARC(25.0,60.0,90.0); /* Draw 90 degree arc with          */
                            /* center at (25,60)                */
```

The arc's starting point is the current position. The first two parameters specify the center of the arc and the third parameter gives its sweep in degrees. A positive angle of sweep denotes a **counterclockwise** arc. A negative angle of sweep will give a clockwise arc.

Note that the radius is not specified explicitly. It will be determined by the distance between the arc's center and the current position before drawing.

Figure 6 shows the effect of two GSARC calls, one clockwise and the other counterclockwise.

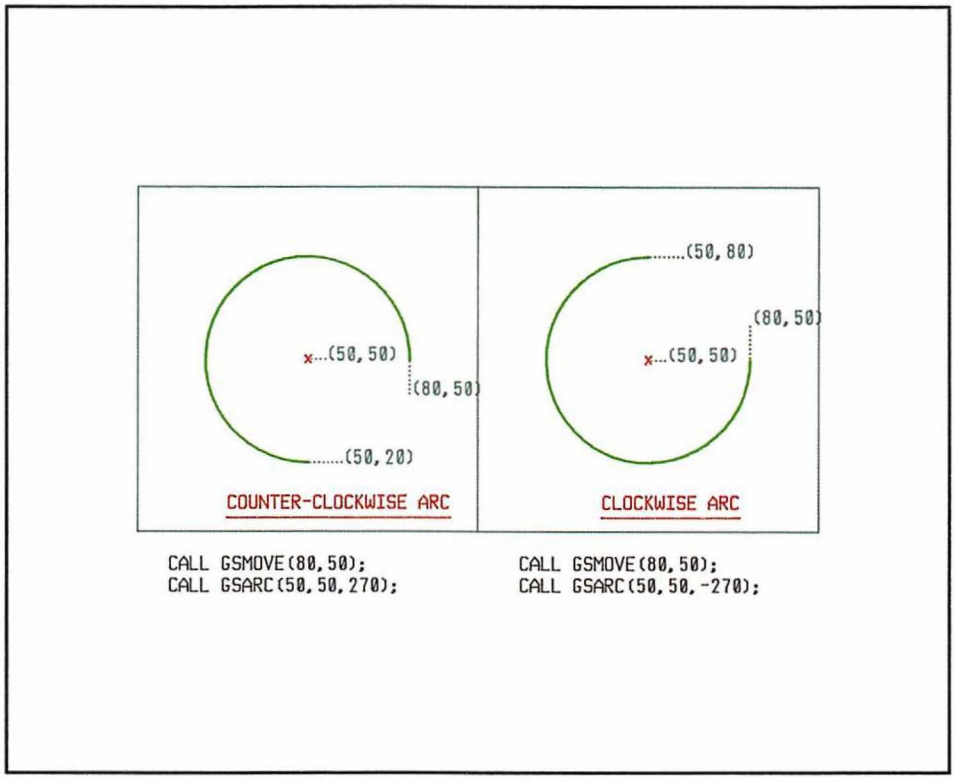


Figure 6. Drawing a circular arc

It is easy enough to write GSARC calls for arcs of 90, 180, or 360 degrees. It is very difficult, though, to determine which GSARC call will go from a known current position to a required end position. Trial and error is hardly a satisfactory method. You should either resort to graph paper, compass, and protractor to determine the GSARC parameters, or use GSPFLT (polyfillet), a simpler call which is described in "Drawing a curved polyfillet using call GSPFLT" on page 24.

Drawing an elliptic arc using call GSELPS

This call draws an elliptic arc that starts at the current position and follows an elliptic curve until it reaches the prescribed end point. This is a typical call:

```
CALL GSMOVE(60.0,70.0);    /* Set starting point for curve       */
CALL GSELPS(20.0,10.0,45.0,30.0,60.0);
                           /* Draw an elliptic arc that has axes */
                           /* of 20 & 10, that is tilted at 45   */
                           /* degrees to the horizontal and that */
                           /* runs from the current position to  */
                           /* an end point of (X=30,Y=60)        */
```

This call is best understood by looking at Figure 7, which shows the various elements of the ellipse.

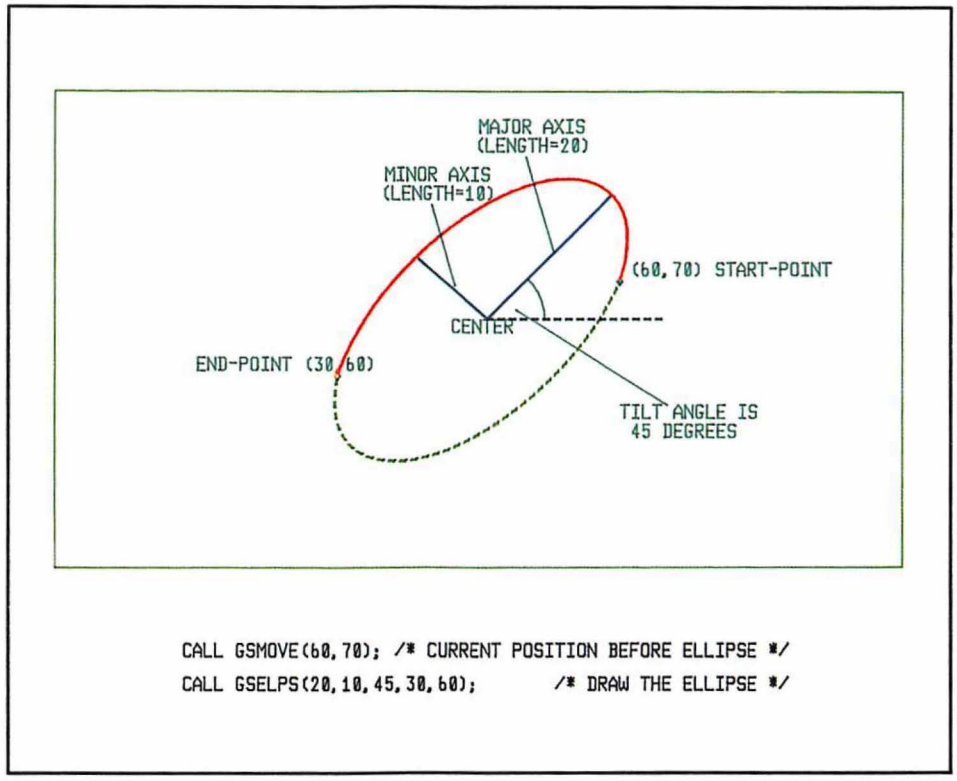


Figure 7. Drawing an elliptic arc

In general there are four elliptic arcs that satisfy the five specified parameters. GDDM will never draw an elliptic arc that is longer than half an ellipse. Of the remaining two arcs, one is clockwise and the other is counterclockwise. If the two axis parameters have the same sign (as in the example), GDDM will draw the counterclockwise arc; otherwise it will draw the clockwise one.

Drawing an elliptic arc using call GSELPS

This call draws an elliptic arc that starts at the current position and follows an elliptic curve until it reaches the prescribed end point. This is a typical call:

```
CALL GSMOVE(60.0,70.0);   /* Set starting point for curve       */
CALL GSELPS(20.0,10.0,45.0,30.0,60.0);
                          /* Draw an elliptic arc that has axes */
                          /* of 20 & 10, that is tilted at 45   */
                          /* degrees to the horizontal and that */
                          /* runs from the current position to  */
                          /* an end point of (X=30,Y=60)        */
```

This call is best understood by looking at Figure 7, which shows the various elements of the ellipse.

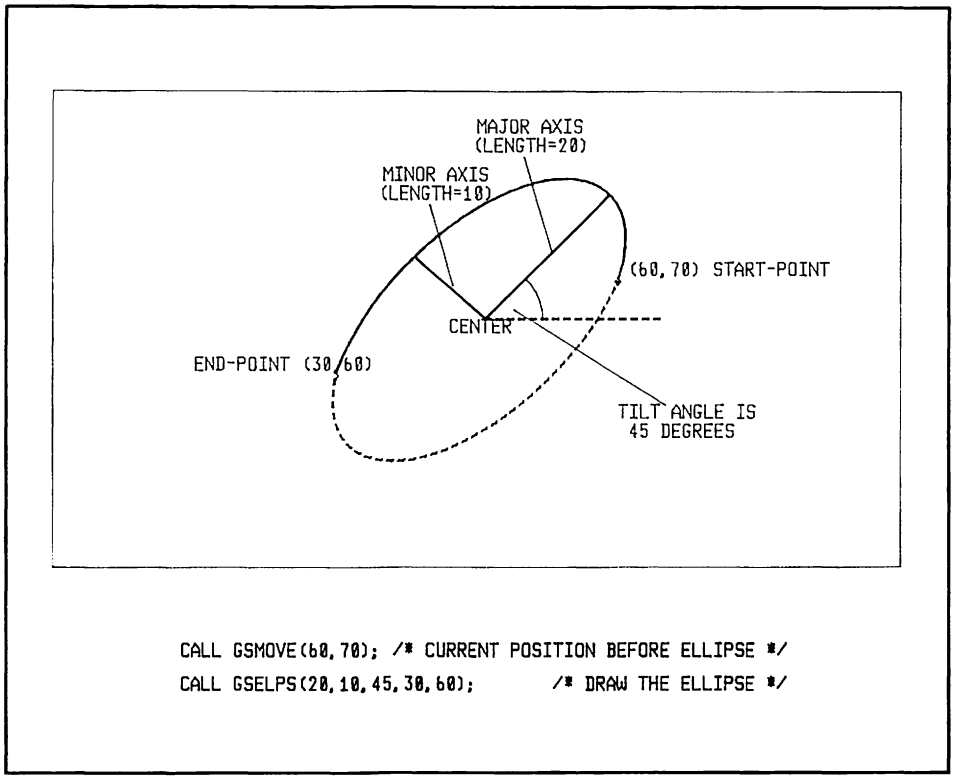


Figure 7. Drawing an elliptic arc

In general there are four elliptic arcs that satisfy the five specified parameters. GDDM will never draw an elliptic arc that is longer than half an ellipse. Of the remaining two arcs, one is clockwise and the other is counterclockwise. If the two axis parameters have the same sign (as in the example), GDDM will draw the counterclockwise arc; otherwise it will draw the clockwise one.

Drawing a graphics marker symbol using call GSMARK

This call draws a single **graphics marker** at a specified position. A graphics marker is a symbol used to indicate a point on the screen. The symbol used depends on the current setting of the marker attribute (see "Setting the current marker symbol, using call GSMS" on page 37). The default is a cross. This is the format of the call:

```
CALL GSMARK(50.0,43.0);  /* Draw graphics marker at (X=50,Y=43) */
```

The marker is positioned so that its center lies at the specified position. The current position is updated to that of the marker.

Drawing several graphics marker symbols using call GSMRKS

This call is a quick way to draw more than one marker. As with the GSPLNE call seen earlier, the coordinates of the points at which the markers should appear are stored in two arrays. Here is an example:

```
DCL X09(7) FLOAT DEC(6) INIT(40.0,50.0,75.0,75.0,80.0,45.0,45.0);
                                                  /* x coordinates */
DCL Y09(7) FLOAT DEC(6) INIT(20.0,20.0,35.0,55.0,20.0,20.0,50.0);
                                                  /* y coordinates */
CALL GSMRKS(7,X09,Y09);  /* Draw 7 graphics markers,             */
                         /* the first at (40,20),                */
                         /* the second at (50,20) and so on      */
```

After drawing the markers, the current position will be set to that of the last marker in the series.

Scaling a marker symbol using call GSMSC

You can control the size of marker symbols, if they are from a vector symbol set (see "Setting the current marker symbol, using call GSMS" on page 37 for more information about markers and symbol sets). For instance:

```
CALL GSMSC(2.0);
```

makes subsequently drawn vector symbol markers twice their default size. The default size of markers is such that their width is equal to the width of the default character box (see "Chapter 7. Basic graphics text" on page 55).

The GSMSC call has no effect on image symbol markers. They are always displayed at the size defined by the symbol itself.

Drawing a curved polyfillet using call GSPFLT

This call is similar in format to GSPLNE. A series of points is passed to GDDM in two arrays. The difference is that whereas GSPLNE results in a sequence of straight lines, GSPFLT results in a smooth curve.

This is the format of a typical call:

```
DCL X09(5) FLOAT DEC(6) INIT(20.0,70.0,70.0,35.0,20.0);
                                              /* x coordinates */
DCL Y09(5) FLOAT DEC(6) INIT(80.0,80.0,50.0,50.0,20.0);
                                              /* y coordinates */

CALL GSMOVE(20.0,20.0);      /* Set current position            */
CALL GSPFLT(5,X09,Y09);      /* Draw 5-part polyfillet.         */
                             /* The first "construction line"   */
                             /* will extend from the current    */
                             /* position to (20,80). The second */
                             /* construction line will run from */
                             /* (20,80) to (70,80), and so on   */
```

The easiest way to visualize the resultant curve is to consider the polyline that would be drawn from the current position through the specified points. These line segments may then be thought of as construction lines for the polyfillet. The polyfillet will start at the current position and finish at the end of the last construction line. On the way it will touch tangentially the midpoints of all the intermediate construction lines. Figure 8, and Figure 9 on page 26 clarify the algorithm. The curves are tangential to the end points of the first and last construction lines, and tangential to the midpoints of all the others.

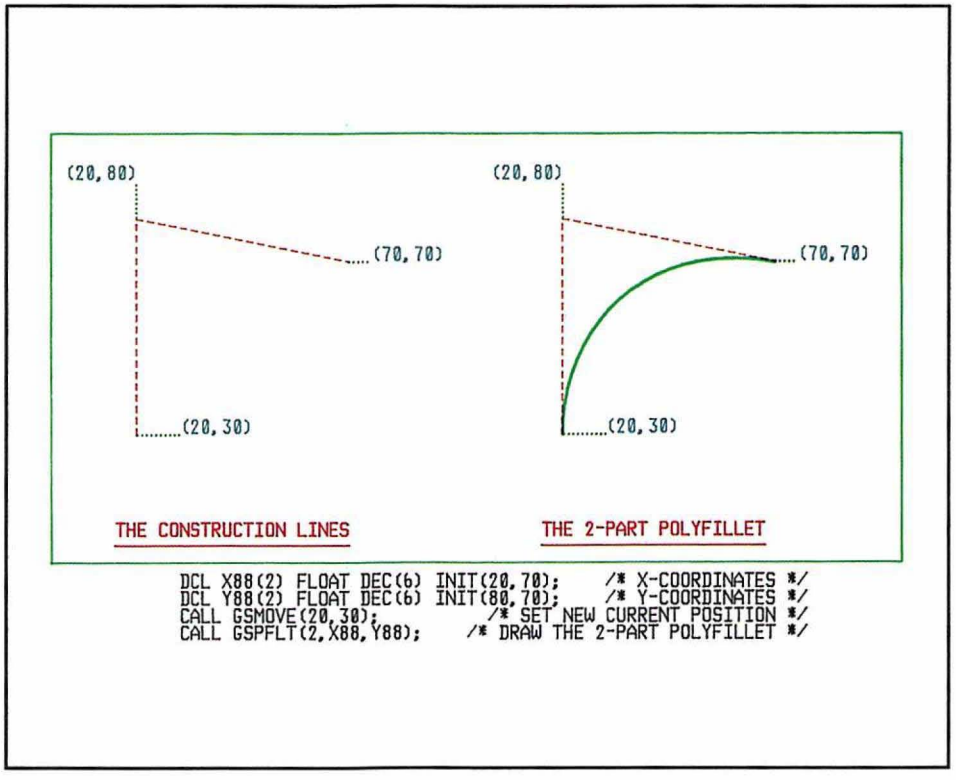


Figure 8. Drawing a 2-part polyfillet

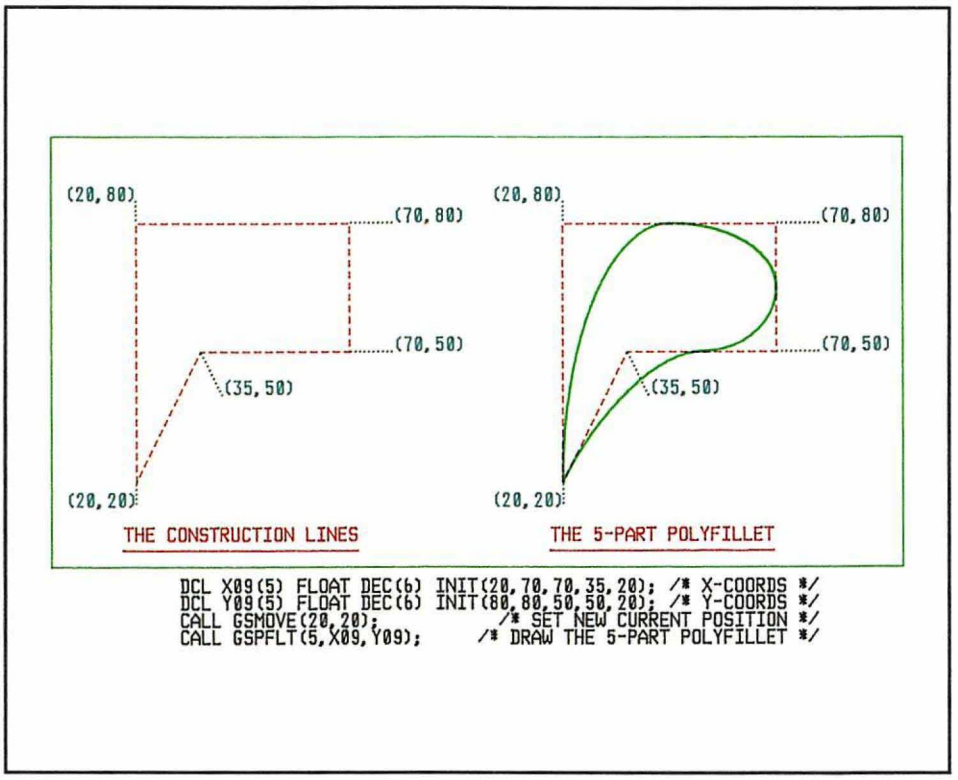


Figure 9. Drawing a 5-part polyfillet

Drawing a graphics area using call GSAREA

A graphics area is a shaded shape. It is defined by specifying its outline and then requesting that it be shaded in. The outline may be constructed using any of the primitives just described (except markers).

Here is an example of a graphics area specification:

```
CALL GSMOVE(70.0,10.0);     /* Move current position to the      */
                            /* start of the area's outline.      */
CALL GSAREA(1);             /* Tell GDDM we are starting an area.*/
                            /* Parameter of 1 = draw boundary    */
                            /* Parameter of 0 = suppress boundary*/
CALL GSLINE(60.0,70.0);     /* The area's outline                */
                            /* begins with a line,               */
CALL GSARC(50.0,80.0,270.0); /* continues with a circular arc,   */
CALL GSLINE(30.0,10.0);     /* and two more lines                */
CALL GSLINE(70.0,10.0);     /* complete the outline.             */

CALL GSENDA;                /* Tell GDDM that the outline is     */
                            /* complete and should now be shaded.*/
                            /* The current color and shading     */
                            /* pattern will be used.             */
```

The resultant shape will be that of a keyhole, as shown in Figure 10 on page 27.

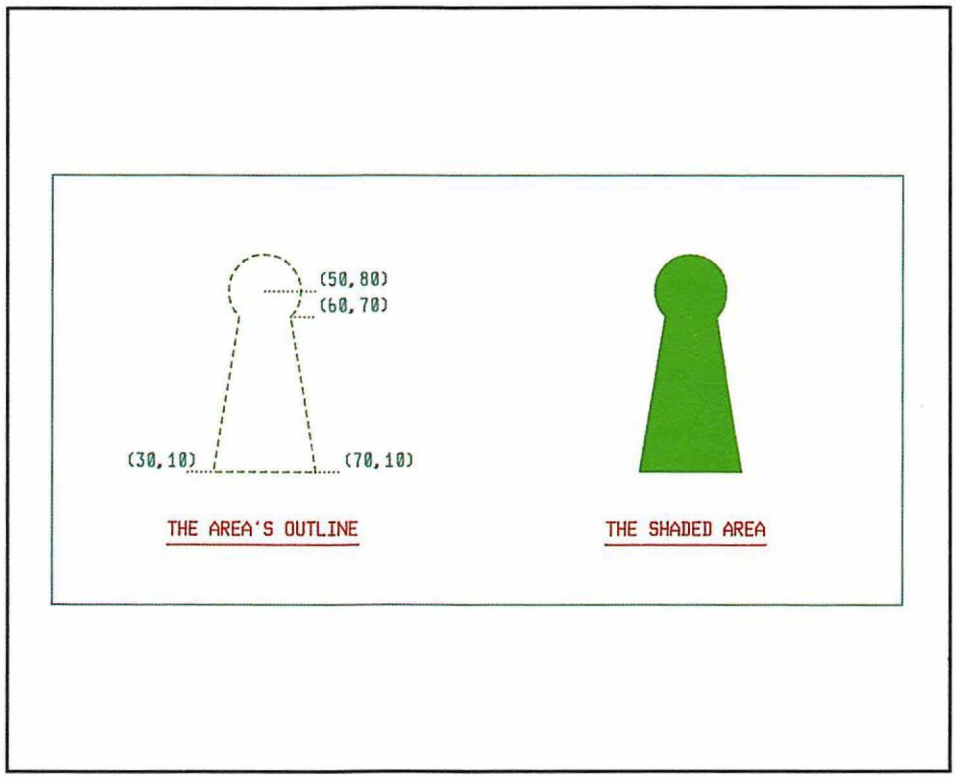


Figure 10. A typical graphics area

Closure of area's outline

The area's outline must be **closed**. If the end of the last primitive in the area is not the same as the current position at the start of the area, GDDM will add a closure line. For example, if your area has only two lines in it (forming a "V"), GDDM will add a third line to make it into a triangle. The current position after the GSENDA call will be at the end of the **added closure line**.

Changing attributes

There are restrictions on changing attributes while drawing an area. They are described in "Changing attributes inside an area" on page 46.

The shading algorithm

A region will be shaded if you need to cross an odd number of lines to move from that region to outside the picture. If you need to cross an even number of lines to leave the picture, the region will not be shaded. Figure 11 on page 28 illustrates this algorithm.

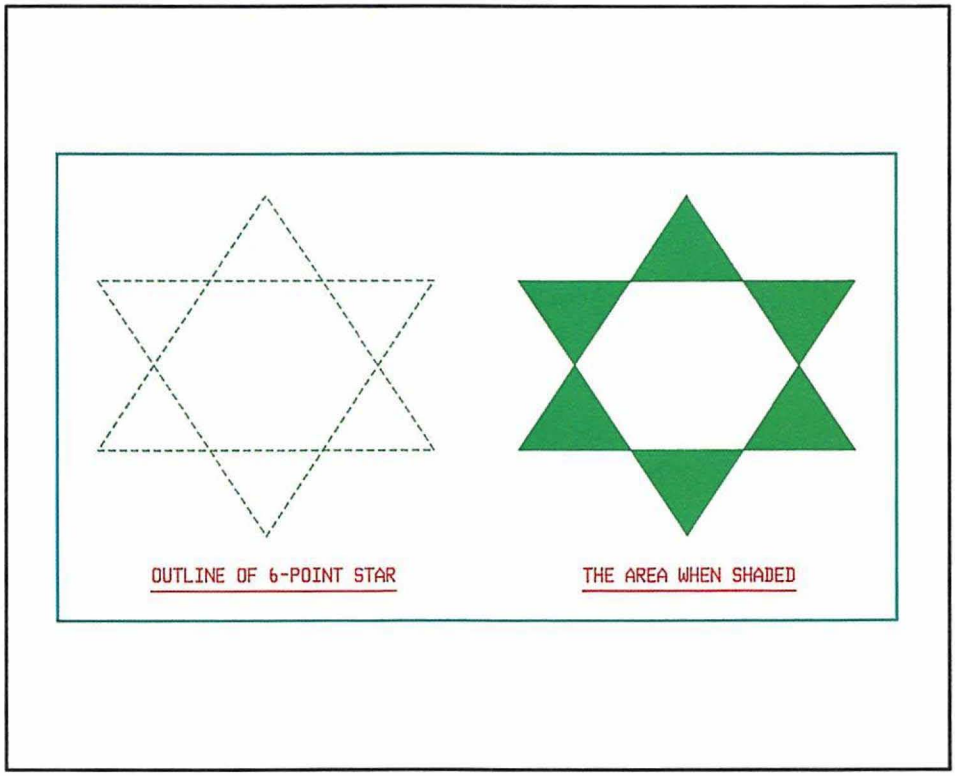


Figure 11. Illustration of GDDM's shading algorithm

If you look at the left-hand part of the figure you will see that all the points in the central area of the star are two line-crossings from infinity. In whichever direction you move, you will cross two lines before leaving the figure. The central part of the star will therefore not be shaded. The six outermost parts of the star are all either one or three line-crossings from infinity, depending on which direction you take. These parts of the shape are therefore shaded. The right-hand part of the figure shows the shaded area.

GSMOVE inside an area

It is permitted to include GSMOVEs inside an area specification. In that case the outline drawn before the GSMOVE must be closed. If not, GDDM will add a closure line before performing the move. The following example illustrates this. The two-part area produced by these GDDM calls is shown in Figure 12 on page 29.

```
CALL GSPAT(3);           /* Area to be shaded                    */
                         /* with system pattern 3.               */
CALL GSMOVE(20.0,20.0); /* Move to area's start position.       */

CALL GSAREA(0);          /* Start area - do not show boundary.   */
CALL GSLINE(40.0,40.0); /* Draw first line of boundary.         */
CALL GSLINE(60.0,20.0); /* Draw second line of boundary.        */

CALL GSMOVE(70.0,60.0); /* Previous part of outline was not     */
                         /* closed, GDDM will add closure line   */
                         /* from (60,20) to the area's           */
                         /* start position at (20,20).           */
CALL GSLINE(80.0,80.0); /* Draw second part of area's outline.  */
CALL GSLINE(30.0,90.0);
CALL GSENDA;             /* Second part of outline was not       */
                         /* closed, GDDM will add closure line   */
                         /* from (30,90) to the start of the     */
                         /* second part at (70,60).              */

CALL ASREAD(TYPE,MOD,COUNT); /* Send the 2 shaded triangles      */
                             /* to the screen.                   */
```

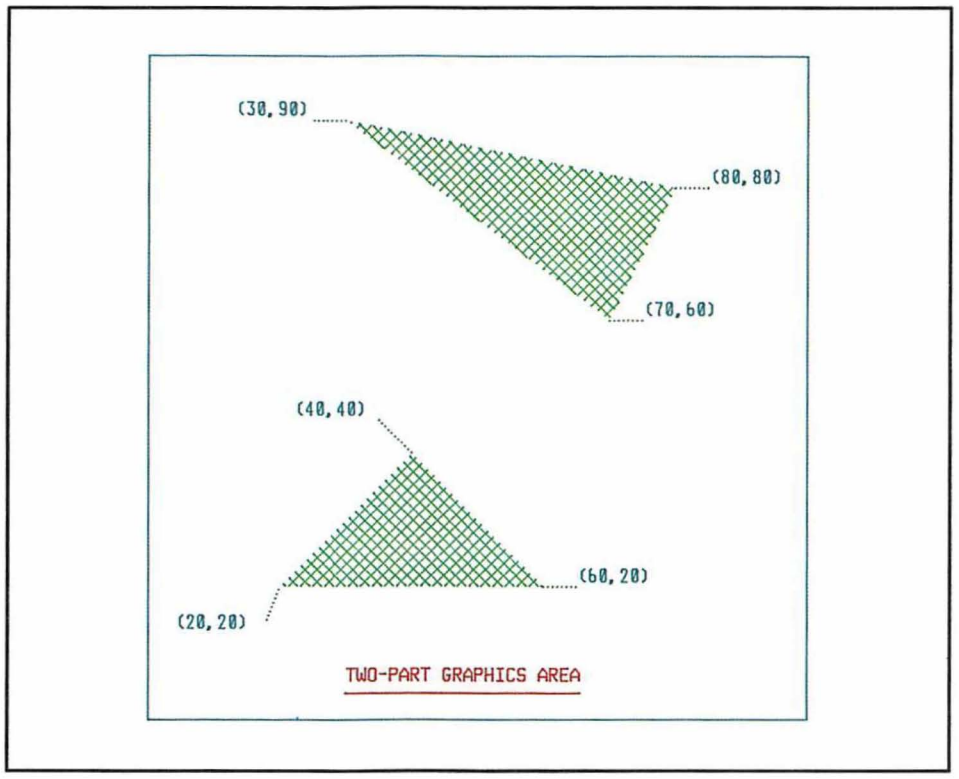


Figure 12. Two-part graphics area with the boundary not drawn

Drawing graphics images using calls GSIMG and GSIMGS

There are two graphics primitives remaining; they operate in a different way from all the others, because they deal with graphics images. Full image processing functions are dealt with in "Chapter 19. Image basics" on page 305 and "Chapter 20. Advanced image functions" on page 331 of this book. See those chapters for GDDM support of image processing on image devices.

The GSIMG call allows you to declare a pattern of dots within the program and then add the pattern to the current page's graphics. Each dot will be displayed as one **pixel** (also occasionally referred to as a pel).

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 from one device to another. For instance, the pixel aspect ratios of the 3279 and 3270-PC/G are different.

This is an example of its use:

```
DCL SPIDER CHAR(198);
UNSPEC(SPIDER)=
        '00000000000000000000000000000000000000000000000000'B||
        '00000000000000000000000010000000000000000000000000'B||
        '00000000000010000000000010000000000000000000000000'B||
        '00000000000010000000000010000000000000000000000000'B||
        '00000000000010000000010000000000000000000000000000'B||
        '00000000000010000000100000000000000000000000000000'B||
        '00000000000010000001000000000000000000000000000000'B||
        '00000000000010000010000000000000000000000000000000'B||
        '00000000000010000010000000000000000000000000000000'B||
        '00000000000010000010000001100000000000000000000000'B||
        '00000000000010000010000111110000000000011000000000'B||
        '00000000000010000100001111110000000000100000000000'B||
        '00000000000010000100001111110000000001100000000000'B||
        '10000000000000010001111111111000000001000000000000'B||
        '01111111110000010111111111000000000011000000000000'B||
        '00000000011000011111111111000000010000000000000000'B||
        '00000000000110111111111111111111000000000000000000'B||
        '00000000000001111111111111000000000000000000000000'B||
        '00000000000001111111111111000000000000000000000000'B||
        '00000000000001111111111111000000000000000000000000'B||
        '00000011111111111111111101110000000000000000000000'B||
        '00000110000000111111110000001110000000000000000000'B||
        '00001100000000001111101000000001111111111000000000'B||
        '01110000000000000100001100000000000000000100000000'B||
        '00000000000000000010000010000000000000000000000000'B||
        '00000000000000000010000001000000000000000000000000'B||
        '00000000000000000010000001000000000000000000000000'B||
        '00000000000000000010000001000000000000000000000000'B||
        '00000000000000000010000001000000000000000000000000'B||
        '00000000000000000010000001000000000000000000000000'B||
        '00000000000000000010000000100000000000000000000000'B||
        '00000000000000000010000000010000000000000000000000'B||
        '00000000000000000010000000001000000000000000000000'B||
        '00000000000000000010000000000100000000000000000000'B;

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

Its parameters are as follows:

- The first parameter must always be set to zero.

- The second parameter, 43, gives the width of the image in display points. This may be any number less than 2040, but the image data must have each row padded to a multiple of 8 bits.

  In the GSIMG example above, the width is 43 and the data has been padded to 48 display points per row.

- The third parameter, 33, gives the depth of the image in display points.

- The fourth parameter, 198, gives the length of the image data in bytes, including padding.

- The last parameter, SPIDER, gives the name of the character variable in which the dot pattern has been stored.

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

Figure 13 shows output from several GSIMG calls, similar to the one above. Note that "black" images will show, when placed on a shaded background.

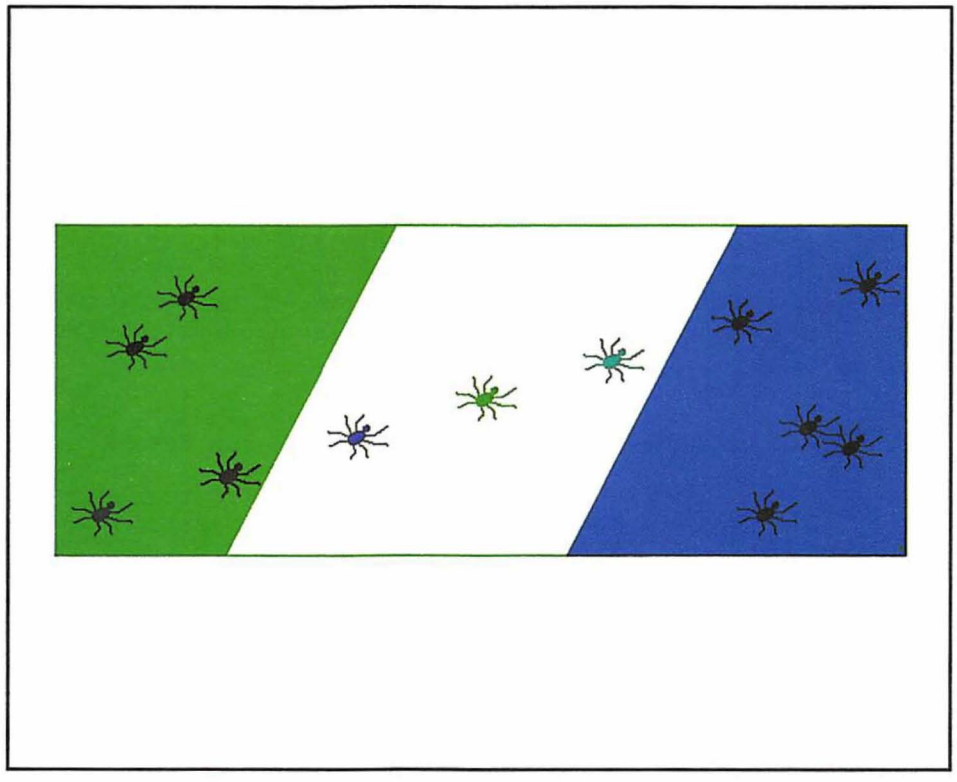


Figure 13. Output from GSIMG statements

The GSIMGS call is similar to GSIMG, except that it allows the image to be scaled. For example:

```
CALL GSIMGS(0,43,33,198,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 will fit 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 will not overflow the image window. The top left-hand corner of the image window will be at the current position.

Because the array need not be a square, the horizontal and vertical dimensions are scaled separately. The mechanism allows only integral scaling, and does not allow scaling down. If a scale factor of less than one would be required to fit the image window, the image is displayed using a factor of one, and is allowed to overflow the image window.

Another method of presenting images (using an image symbol set) is described in "Chapter 7. Basic graphics text" on page 55.

Querying the current position using call GSQCP

At any stage in a graphics program you may query the current position. It will be returned in world coordinates. This is the call:

```
DCL (X,Y) FLOAT DEC(6);/* Parameters for query current position */

CALL GSQCP(X,Y);       /* Query the current position.           */
```

On return from this call, GDDM will have set the current position into variables X and Y. Here is an example of using this function to underline a graphics text string:

```
CALL GSCHAR(20.0,34.0,26,'Figure 8. The Eye of a Fly');
                         /* Write text.                          */
CALL GSCOL(2);           /* Change color to red.                 */
CALL GSQCP(X,Y);         /* Determine position of right-hand end */
                         /* of text string.                      */
CALL GSMOVE(X,33.5);     /* Move down by 0.5 y window units.     */
CALL GSLINE(20.0,33.5); /* Underline the text in red.            */
```

Here, the y position was known (34). GSQCP was used to determine the x position of the end of the string.

Querying the cursor position using call GSQCUR

To query the cursor position in terms of your world-coordinate system, you issue this call at some stage after executing an ASREAD:

```
DCL INWIN FIXED BIN(31); /* Declare fullword parameter.          */
DCL (X,Y) FLOAT DEC(6);  /* Declare 2 float parameters.          */

CALL GSQCUR(INWIN,X,Y);  /* Query cursor position.               */
```

INWIN will be set to 1 if the cursor position was inside your graphics window, and to 0 otherwise.

Parameters X and Y will be set to the x and y coordinates of the center of the cell containing the cursor.

Another way of determining the cursor position is described in "Chapter 14. Interactive graphics" on page 177.

Device variations

IBM 5080 Graphics System

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

You cannot produce multicolored images by overlaying graphics images created by GSIMG. The whole of each successive image will blank out any underlying graphics.

Chapter 5. Graphics attributes

There are several attributes that affect the appearance of graphics primitives such as lines and arcs. Each of these attributes has a default setting initially. For example, on a 3179-G or 3279 terminal the default color is green and the line type solid.

At any stage the program can change a particular attribute. All primitives drawn subsequently will assume the new attribute value. In a program that uses segments, the effect of the calls that change attributes is limited to the segments in which they are 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, however, that you can change the default attribute settings in your program to defaults of your own choosing. See "Changing default attribute values" on page 47 for details.

Setting a new current color, using call GSCOL

The current color affects the appearance of all graphics output – lines, arcs, areas, graphics text, graphics images, and markers. It is set by this call:

```
CALL GSCOL(2);                    /* Set the current color to red */
```

The parameter may take these values:

-2 White
-1 Black
0 Default (initially green on color displays, black on printers)
1 Blue
2 Red
3 Pink (magenta)
4 Green
5 Turquoise (cyan)
6 Yellow
7 Neutral (white on display, black on printers)
8 Background (black on displays, white on printers).

Information about what happens if the device does not support the chosen color is given in the *GDDM Base Programming Reference* manual. Information about the 16-color version of the 3270-PC/GX is given in "IBM 3270-PC/G and /GX work stations" on page 49.

The same codes (except −2 and −1) are used in other calls for specifying colors. A suggested mnemonic for the codes for blue through neutral is:

Boys Reading Politics Go To Yale Now

Setting a new line type, using call GSLT

There are a number of different line types (or styles).

This call sets a new current line type:

```
CALL GSLT(1);                /* Set the current line type to dotted */
```

The parameter may take the value 0 through 8. The effect of these line types on a 3179-G terminal can be seen in Figure 14.

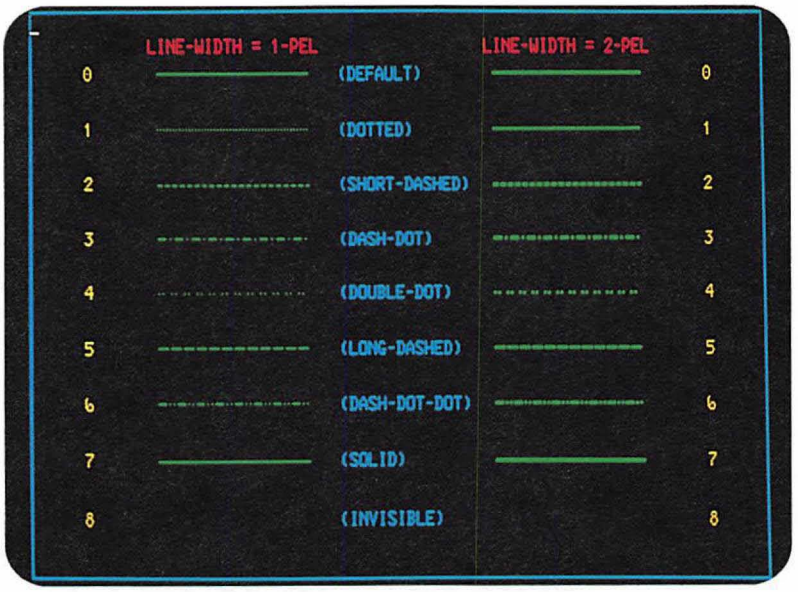


Figure 14. GDDM line types and line widths

Setting this attribute affects the appearance of all subsequently drawn primitives such as lines, arcs, and ellipses. It also affects the boundaries of graphics areas.

Setting a new line width, using calls GSFLW or GSLW

You can vary the line width used for graphics with this call:

```
CALL GSFLW(0.66);/* Set current line width to two-thirds standard */
```

The parameter specifies a factor by which the standard line width for the current device is to be multiplied.

Omitting the call or specifying a value of 1 gives a line of the standard width for the device. On display devices the standard width is one pixel, and the only other available thickness is two pixels. These two widths are shown in Figure 14.

On all devices, a value of zero gives the current drawing default. This is initially the standard width for the device.

On high-resolution devices, such as the 4250 printer, a line one or two pixels wide would be nearly invisible. Line widths of up to 600 pixels are allowed. The standard width on a 4250 is six pixels. More information about standard and maximum line widths is given in the *GDDM Base Programming Reference* manual.

There is another call that is similar in effect to GSFLW, but requires a fixed-point, and therefore integral, parameter:

```
CALL GSLW(2);   /* Set the current line width to twice standard */
```

Setting the current marker symbol, using call GSMS

As described in "Drawing a graphics marker symbol using call GSMARK" on page 24, the primitive GSMARK puts out a graphics marker at a specified location. There are several styles of marker and the GSMS call is used to change to a new style.

```
CALL GSMS(3);                          /* Set marker type to diamond */
```

The parameter may take these values:

0	Default (initially a cross)
1	Cross
2	Plus-sign
3	Diamond
4	Square
5	Six-point star
6	Eight-point star
7	Shaded diamond
8	Shaded square
9	Dot
10	Small circle
65 to 254	User-defined markers.

The markers numbered 1 through 10 are called **system markers**. They are symbols contained in GDDM-supplied symbol sets. The markers are illustrated in Figure 15 on page 38.

It is also possible for the users to create their own markers, using the GDDM Image Symbol Editor or Vector Symbol Editor. These markers may be of any size. They will still be positioned by GDDM such that the center of the marker symbol lies at the specified position.

Information about the size of vector symbol markers is given in "Scaling a marker symbol using call GSMSC" on page 24.

The following code will load a user marker set, and then display one of its markers. The GSLSS call `/*A*/` is described in "Symbol sets for graphics text" on page 222. The first parameter is set to 4 to indicate that the symbol set being loaded is a marker set. The second parameter is the name of the symbol set. The third parameter must be set to 0 in this instance. The GSMS call `/*B*/` identifies the symbol by its position in the set. A marker symbol set may have markers at any or all of the positions 65 through 254. If you specify a position where no marker has been created (in other words, an empty position in the symbol set), no marker will be drawn.

You are allowed only one user marker set at a time. You can then choose either a marker from the user set, or one of the ten system markers. If you load a second user marker set, it will replace the previously loaded one.

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

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

CALL GSMARK(50,50);         /* Draw user marker 72 at X=50,Y=50 */
```

If the marker set is multicolored, you must set the current color to 7 (neutral) before using any markers that need to be multicolored in the display.

Figure 15. The 10 GDDM system markers

Setting the current pattern, using call GSPAT

The scheme for shading patterns is similar to that for markers. There are 16 system patterns, and the user may also create his own patterns with the Image Symbol Editor (but not the Vector Symbol Editor), and subsequently specify their use. This is the call to select a new current shading pattern:

```
CALL GSPAT(11);     /* Set current pattern to system-pattern 11 */
```

All subsequently drawn areas will be shaded in this pattern until a new shading pattern is specified. The parameter may take these values:

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

The available system patterns for displays and 3287s are shown in Figure 16 on page 39.

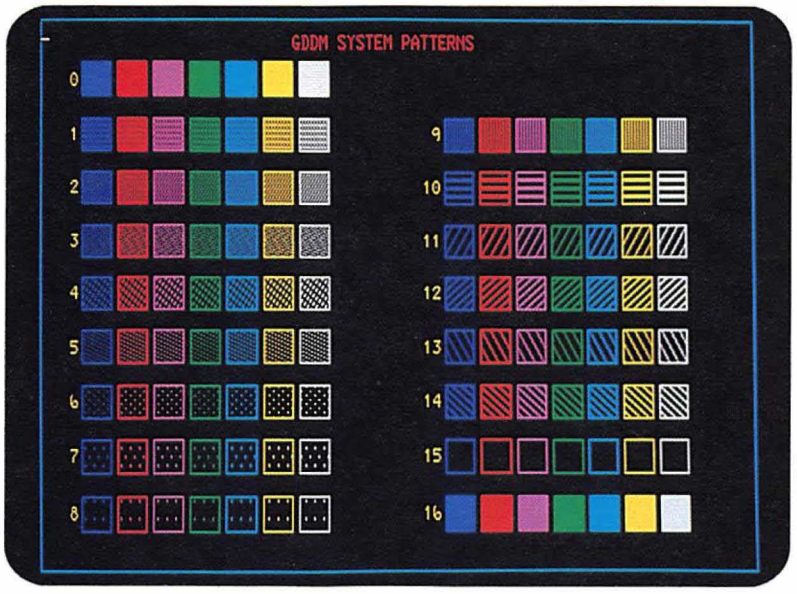


Figure 16. The 16 GDDM system shading patterns

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.

The following code will load a user pattern set, and then use one of its patterns. The first parameter of the GSLSS call /*A*/ is set to 3 to indicate that the symbol set being loaded is a pattern set. The GSPAT call /*B*/ identifies the pattern by its position within the set. A pattern symbol set may have patterns at any or all of the positions 65 through 254. If you specify a position at which no pattern has been created (in other words, an empty position in the symbol set), subsequent areas will be unshaded.

```
CALL GSLSS(3,'PRETTY',0); /* Load user pattern set             */ /*A*/
                          /* called PRETTY.                    */

CALL GSPAT(97);           /* Set pattern to symbol 97 (X'61')  */ /*B*/
                          /* in the currently loaded user      */
                          /* pattern set.                      */

CALL GSMOVE(80.0,22.4);   /* Move to start point of            */
                          /* graphics area.                    */
CALL GSAREA(1);           /* Start a graphics area.            */
CALL GSLINE(90.0,30.0);   /* Draw first line of the outline,   */
   .                      /* and so on...                      */
   .
CALL GSENDA;              /* End the area, and shade it with   */
                          /* pattern 97 from the user pattern  */
                          /* set called PRETTY.                */
```

You are allowed to load only one user pattern set at a time. You may then use either a pattern from the loaded set or one of the 16 system patterns. You must set the current color to 7 (neutral) before using a pattern from a multicolored set if you want the pattern to be multicolored in the display.

Several sample user pattern sets are supplied with the GDDM package. One of them, the geometric pattern set, is shown in Figure 17 on page 40.

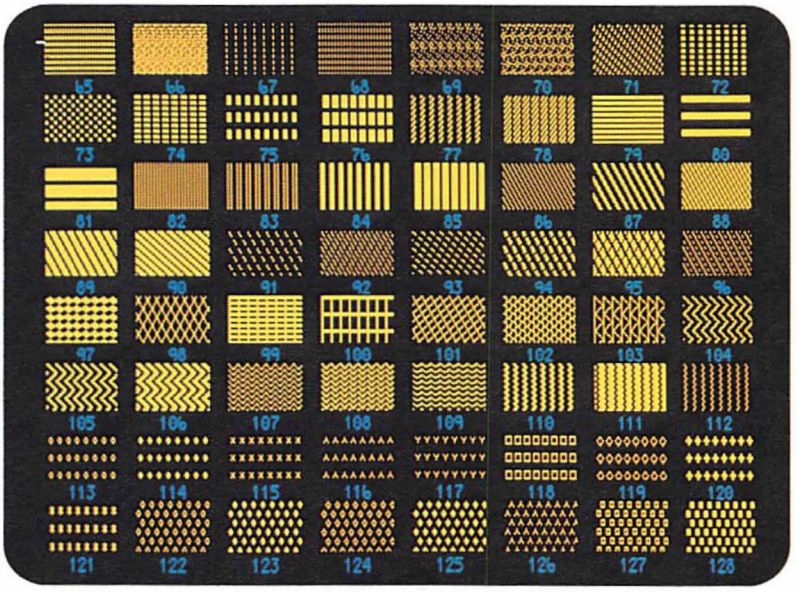


Figure 17. GDDM geometric pattern set - ADMPATTC

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

The GDDM 64-color pattern set

The GDDM-supplied symbol sets ADMCOLSD, ADMCOLSN, and ADMCOLSR allow you to shade your areas in any of 64 different colors. These colors are shown in Figure 18 on page 41. 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.*/
```

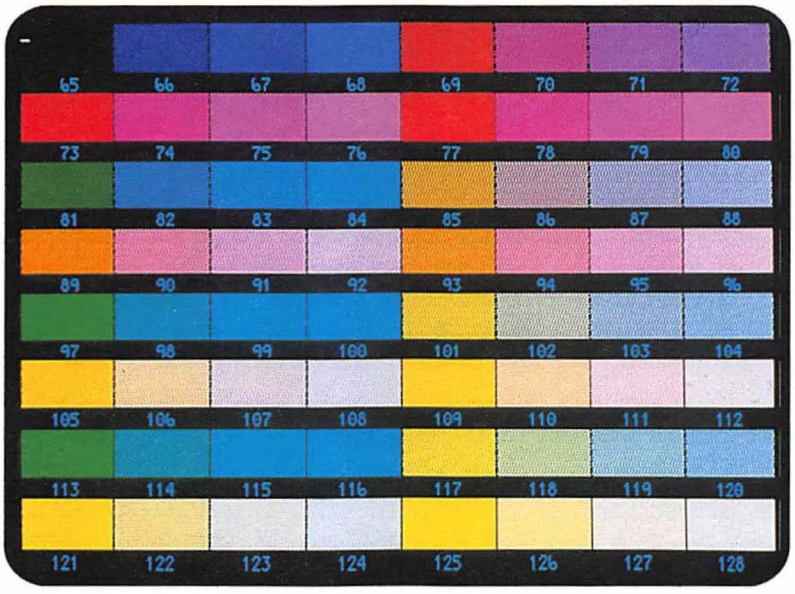


Figure 18. GDDM 64-color pattern set - ADMCOLSD

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 will be white (or black on a printer) unless you reset the color after opening the area. Here is how to draw a red outline around a multicolored area:

```
CALL GSLSS(3,'ADMCOLSD',0);     /* Load 64-color pattern set.     */
CALL GSPAT(83);                 /* Select blue pattern.           */
CALL GSCOL(7);                  /* Set color to neutral           */
                                /* for area fill.                 */
CALL GSAREA(1);                 /* Open area.                     */
CALL GSCOL(2);                  /* Set color to red for outline.  */
        .
        .
/* Draw the area */
        .
        .
```

Or, instead, you can specify GSAREA(0) to suppress the drawing of the boundary.

Mixing foreground colors, using call GSMIX

By default, graphics primitives are drawn on top of the primitives drawn previously. If you draw a blue line and then a green line that crosses it, the crossing point will be shown in green. This form of foreground color mixing is called **overpaint mode**. The other foreground modes that can be set are **mix mode**, **underpaint mode**, and **transparent mode**.

All the displayable colors are made up of one or more of the three primary colors, blue, red, and green. If you set mix mode, and then draw a blue line crossed by a green one, the point where they cross will be a mixture of blue and green, that is turquoise. Using all combinations of the three primary colors, seven colors can be created, as shown in Figure 19.

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

Figure 19. The seven displayable colors

Mixing two colors results in combining their primaries. For example, red mixed with pink (blue and red) will give blue and red, that is, pink. Turquoise (blue and green) mixed with yellow (red and green) will give blue, red, and green, which is white.

A color representation of the possible mixes is given in Figure 20 on page 43.

The third form of color mixing is underpaint mode. Wherever two primitives cross, the displayed color will be that of the first-drawn primitive. If you draw a blue line, then a green line crossing it, the crossing point will be shown in blue. Not all devices support underpaint mode (see "Device variations" on page 49).

The fourth form of color mixing is called **transparent mode**. Primitives drawn in this mode will be transparent and will therefore not appear. Not all devices support transparent mode (see "Device variations" on page 49).

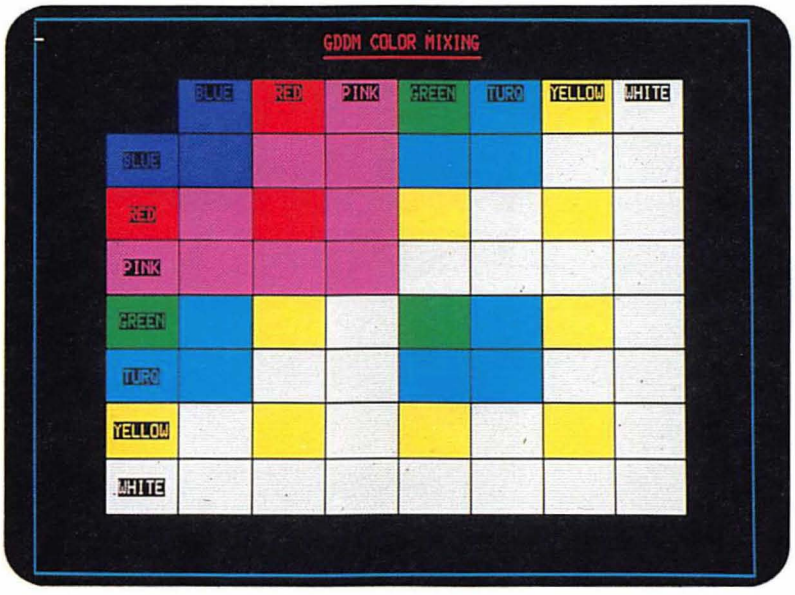


Figure 20. Color-mixing table

The call that defines the mixing mode is a simple one:

```
CALL GSMIX(1);      /* Set current color-mixing rule to mix mode.*/
```

The possible values of the parameter are as follows:

0 Current default
1 Mix mode
2 Overpaint mode (the initial default)
3 Underpaint mode
4 Overpaint mode
5 Transparent mode.

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

Special treatment of the background color, using call GSMIX

One of the colors allowed on the GSCOL call is color 8, the background. This color shows as black on a display and white on a printer or plotter. When it is mixed with another color, it has the following special effects:

- To erase graphics from a part of the screen, you can simply paint over the graphics with a background area, using the initial default color-mixing mode, overpaint. This technique may be used on, for instance, a 3279 terminal, to produce a cartoon effect. To show an owl blinking his eye, 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;
```

- Underpaint mode does not apply when the underlying color is background. The reason is that there is no such thing as a background color to take precedence. Background primitives are represented by switching off all the primary colors.

- Background primitives make no impact on the previously drawn graphics if mix mode is in effect. Remember that the effect of mix mode is to add the primary components of the two colors together. Because "background" means having no primaries, there is nothing to be added – the original color stands.

- The effect of reverse-video can be achieved 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 7. Basic graphics text" on page 55.) Except on a 3270-PC/G or /GX, this technique will not work with mode-1, because the characters occupy whole cells to the exclusion of the graphics. Background mode-1 text would be invisible.

Mixing background colors, using call GSBMIX

We have seen how GDDM gives you control over the mixing of the foreground color of overlapping primitives. For certain primitives, you can also control how the **background** of the current primitive combines with any previously drawn primitives that 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 will be black for a display, and white for a printer or plotter.

The format of the call to set background mix mode is as follows:

```
CALL GSBMIX(2);      /* Set background mix mode to opaque.*/
```

The possible values of the parameter are as follows:

0 Current default
2 Opaque mode
5 Transparent mode (the initial default)

The graphics primitives (and their backgrounds) for which you can set this attribute are:

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

Image markers The background is every pixel that is not set within the marker definition.

Vector markers The background is the complete marker box.

Areas The background is every pixel within the area that is not set by the shading pattern. For example, an area containing a shading pattern that is a grid of horizontal and vertical lines is drawn over some existing primitives. If the background mix mode is set to transparent, the underlying primitives will be seen through the square " holes" contained by the horizontal and vertical lines. If the background mix mode is set to opaque, the underlying primitives will be covered up by the holes, which will contain background color only.

Graphics text The effect of background mix 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" on page 49. For Mode-3 text, the background is the complete character box. For more information, see "Chapter 7. Basic graphics text" on page 55.

GSBMIX has no effect on lines. Background mix mode is valid for all devices when the foreground mix mode is overpaint. For details of which devices support which combinations of foreground and background mix modes, see "Device variations" on page 49.

Transforming primitives, using call GSSCT

You can set a current transform that will be applied to all the primitives that follow using the GSSCT call. Primitives can be transformed in four ways:

Displaced Moved to another x,y location

Scaled Made larger or smaller in the x,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 a primitive attribute, 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 therefore covered more fully in "Transforming primitives within a segment" on page 143 in "Chapter 11. Graphics segments."

Changing attributes inside an area

It is not permitted to change, say, the shading pattern in the middle of defining an area. Only four attributes may be changed: the line type (CALL GSLT), the line width (CALL GSFLW or CALL GSLW), the color (CALL GSCOL), and the mixing mode (CALL GSMIX). Changes to these attributes will affect the drawing of subsequent parts of the area boundary, but not the area fill. The attributes of the fill are fixed when the GSAREA is executed.

Querying graphics attributes

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 permit a subroutine to maintain the environment at the time of its calling. For example, a subroutine that draws a thick red square at an x,y position passed to it might look like this:

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

RSQUARE: 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-1.0,Y-1.0);   /* Move to start of red square.      */
CALL GSLINE(X+1.0,Y-1.0);   /* Draw first line of square.        */
CALL GSLINE(X+1.0,Y+1.0);
CALL GSLINE(X-1.0,Y+1.0);
CALL GSLINE(X-1.0,Y-1.0);

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

END RSQUARE;
```

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.

Changing default attribute values

When a primitive is processed, any attributes that relate to the primitive and that have not been explicitly set assume drawing default values supplied by GDDM. At any time in your program you can change the drawing defaults from the values supplied by GDDM to default values of your own choice – affecting color, line width, line type, shading patterns, graphics text, symbol sets, and many other attributes.

You can achieve this by containing attribute calls within two calls, GSDEFS and GSDEFE, that 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) will result in a square marker symbol.

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 in 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 will be reset to the GDDM default value.

For both merge and override, the existing defaults can be either GDDM 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 will be redrawn using the new one. For example, you could draw and display a segment 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 will be redrawn in the new color. Primitives outside segments will be discarded when the redraw occurs.

See "Chapter 12. Storing graphics" on page 157 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, refer to the *GDDM Base Programming Reference* manual.

Pushing and popping graphics attributes, using calls 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 value of the parameter is:

0 Preserve the attributes (the default)
1 Do not preserve the attributes

You can save all the primitive attributes introduced in this chapter (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 coverage of GSAM in the *GDDM Base Programming Reference* manual.

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 152.

Device variations

The preceding sections of this chapter refer primarily to the 3179-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 3270 family of terminals

This covers members of the the 3270 family that use programmed symbols for graphics, such as the 3279.

GSBMIX call: Background mix is only supported when the foreground mix mode is overpaint.

IBM 3270-PC/G and /GX work stations

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

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

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

The results for mix mode with the above colors are as indicated in the table in Figure 21 on page 50.

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

Figure 21. GSMIX table for mix mode on the 3270-PC/GX

Pattern sets: There must be sufficient symbol set storage available in the work station for any specified pattern set, otherwise the default pattern will be used for shading.

IBM 5080 Graphics System

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 one 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.

Color-separation masters on printers

GSCOL call: If color separation is required on a family-4 device (see "Composed-page printer as a family-4 primary device" on page 399), the value of the GSCOL parameter can range from 0 to the number of entries in the selected color table.

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 434.

Pattern sets: You cannot specify user pattern sets for plotters.

GSBMIX call: Background mix is only supported when the foreground mix mode is overpaint.

4224 IPDS printers

GSMIX call: Only overpaint is supported.

Pattern sets: Only the 16 GDDM-supplied system shading patterns are available. Any other specified pattern results in pattern 16 (solid).

Chapter 6. Displaying text

GDDM provides three different sets of functions for displaying characters and other symbols: graphics text, procedural alphanumerics, and mapped alphanumerics. This chapter 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 8 is in graphics text. It was created simply by executing a GSCHAR call for each line.

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

The location of the text is specified in world coordinates, and it can be positioned to pel 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 3270-PC/G and /GX work stations, and on 5080 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, are primarily intended for use in a graphics context - for instance, to allow the terminal user to enter parameters concerning a picture currently on display. On other types of terminal, graphics text is output only.

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

For more information about writing graphics text, see "Chapter 7. Basic graphics text" on page 55. For input, see "Chapter 14. Interactive graphics" on page 177.

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 or the 4250 printer.

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 fields' size, position, and other characteristics, then to put data into

them. After an ASREAD, alphanumeric data entered by the terminal operator can be read from the fields.

Alphanumeric fields do not generally mix well 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 8. Basic alphanumerics" on page 75 and "Chapter 16. Advanced procedural alphanumerics" on page 235.

Mapped alphanumerics

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

The definition is done interactively, using the GDDM Interactive Map Definition product (GDDM-IMD). 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 as 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.
- 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 17. Mapped alphanumerics" on page 251 and "Chapter 18. Variations on a map" on page 273.

Chapter 7. Basic graphics text

This chapter describes the output of graphics text. Input on 3270-PC/G and /GX work stations, and on 5080 Graphics Systems, is described in "String input" on page 184.

To add graphics text to a display, there are two possible calls. One is GSCHAR:

```
CALL GSCHAR(30.0,90.0,11,'TOTAL SALES');
                                /* Put 11 characters of graphic  */
                                /* text in position (30,90)      */
```

As with all graphics calls, the position is given in world coordinates rather than the rows and columns scheme used for alphanumerics. The text itself may be a character constant (as here) or a character variable.

The second call, GSCHAP, is similar, but the string is located at the current position, instead of a specified position:

```
CALL GSCHAP(11,'TOTAL SALES'); /* Send 11 characters of graphics*/
                               /* text to the current position. */
```

GSCHAR and GSCHAP leave the current position set to the end of the created text string. GSCHAP is most frequently used when concatenating text, for example:

```
DCL PPP PIC'$$$$$9';      /* PL/I picture variable to edit data.*/
DCL PROFIT FIXED BIN(31); /* Variable holds the year's profit.  */
PPP=PROFIT;           /* Convert from numeric to character form */

CALL GSCHAR(30.0,45.0,25,'THE PROFIT THIS YEAR WAS ');

CALL GSCHAP(6,PPP);                /* Concatenate actual profit.*/
CALL GSCHAP(13,' (BEFORE TAX)');   /* Concatenate further text. */
```

If the profit was, say, $45300, the output from these calls would be:

THE PROFIT THIS YEAR WAS $45300 (BEFORE TAX)

Breaking lines of graphics text

To request a line break, you must include the special character code X'15' in your text string. Because PL/I does not support hexadecimal constants, this is the code required:

```
DCL CHAR1 CHAR(1);                  /* Declare temporary variable.*/
UNSPEC(CHAR1)='00010101'B;          /* Assign X'15' into variable.*/
/*****************************/
/* Put out 2-line text string */
/*****************************/
CALL GSCHAR(20.0,20.0,22,'FIRST LINE'||CHAR1||'SECOND LINE');
```

The output will appear as:

FIRST LINE
SECOND LINE

The three modes of graphics text

When creating graphics text, you can specify many attributes that will affect its appearance. The most important of these is the **mode** of the text, which can have the value 1, 2, or 3. You can specify the mode with the GSCM call:

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

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

The character mode determines which type of **symbol set** is used. 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 the main description of symbol sets, see "Chapter 15. Symbol sets" on page 219. 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 "Differences on the IBM 3270-PC/G and /GX work stations" on page 70.

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

Mode-1 and Mode-2 are highly device-dependent. This chapter describes their use primarily on the ordinary terminals in the 3270 family, such as the IBM 3279. Differences on other types of device are described at the end of the chapter.

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 73.

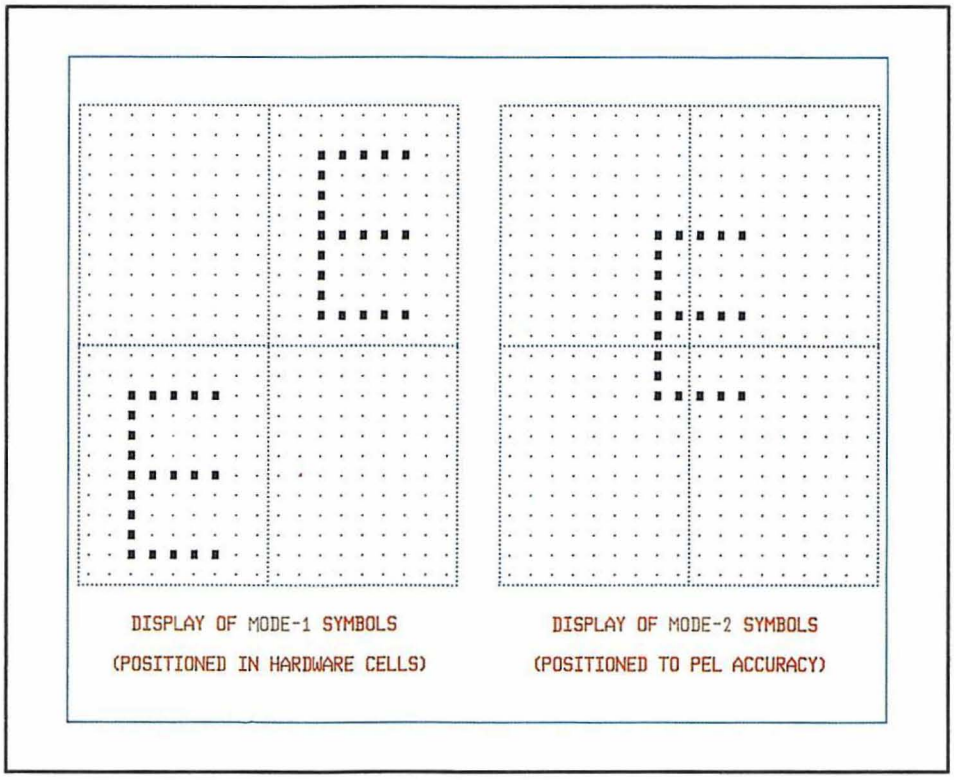


Figure 22. Mode-1 and mode-2 graphics text

Mode-1 graphics text

Mode-1 is basically the same as GDDM alphanumerics output (see "Chapter 16. Advanced procedural alphanumerics" on page 235). The symbols occupy one hardware cell each. By default, the device's own hardware symbol set will be used, but the application program can load its own image symbol set (see "Symbol sets for graphics text" on page 222). Only image symbols that match the hardware cell size may be used.

A mode-1 symbol occupies its cell completely. Any graphics in the cell is obliterated. This can aid the readability of the text.

A general name for mode-1, applicable to all types of terminal, is string-positioning mode, indicating that the application program can control the position of the start of the string only.

Mode-2 graphics text

Mode-2 text is similar to mode-1 in many respects. Like mode-1, it is composed of image symbols. GDDM will load a default image symbol set, or the application may load one explicitly (see "Symbol sets for graphics text" on page 222). The symbols may be of any size, and they are positioned to pixel accuracy.

Figure 22 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 "Mixing foreground colors, using call GSMIX" on page 42 and "Mixing background colors, using call GSBMIX" on page 45).

The general name for mode-2 is character-positioning mode, indicating that the application program dictates the position of each character (or symbol) within the string.

Mode-3 graphics text

Mode-3 text is composed of vector symbols. GDDM will load a default vector symbol set, or the application may load one explicitly (see "Symbol sets for graphics text" on page 222).

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 ordinary 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 "Mixing foreground colors, using call GSMIX" on page 42 and "Mixing background colors, using call GSBMIX" on page 45).

The general name for mode-3 is stroke-positioning mode, because the application program can control the drawing of every stroke of every symbol.

Affecting the appearance of graphics text, using attributes

There are several attributes that affect the appearance of graphics text. How much effect a particular attribute has on the character string depends on the mode of the text. The general situation is that all the attributes apply fully to mode-3 (stroke-positioned) text. Some of them apply to mode-2 (character-positioned) text but hardly any affect mode-1 (string-positioned) text.

Each of the attributes will be described, together with its effect on each of the three modes.

Setting the character box attribute, using call GSCB

This affects the size and spacing of the characters within a text string. The call has two parameters: the width of the character box (expressed in x world coordinates) and the height of the character box (expressed in y world coordinates). This is a typical call:

```
CALL GSCB(2.5,2.0);   /* Set character box of size x=2.5, y=2.0 */
```

This would have the following effect on the three modes of text:

Mode-1 Hardware characters are placed in successive cells. The character-box attribute is therefore completely ignored.

Mode-2 Image symbols are used, and these are of fixed size - they cannot be expanded or contracted to fit the character box. The character-box setting therefore affects their **spacing**. Successive characters will be spaced 2.5 x units apart and the lines 2 y units apart. If you use a symbol set that is larger than this, and do not adjust the character box, then your symbols will overlap. If you use a symbol set that is smaller, there will be extra space around each symbol.

Mode-3 Each character would be scaled to fill the character box of 2.5 by 2 in world coordinates, separate scale factors being used for the width and depth to fill the box in both directions. The space allocated to each character would be 2.5 x units wide (unless the symbol set is proportionally spaced - see "Using proportionally spaced characters" on page 60). Should a new-line character occur, the second line would be placed 2 y units below the first.

The default character box is the hardware character cell.

Figure 23 shows the effect of a GSCB call on text of the three different modes displayed on a color graphics display. If you want to set the character box to four times its normal size, you must first query the attribute's default value in window coordinates:

```
CALL GSQCB(WIDTH,HEIGHT);     /* Query character box.              */
                              /* (When this query is made          */
                              /* before any GSCB call,             */
                              /* default value will be returned) */
CALL GSCB(WIDTH*4.0,HEIGHT*4.0);
                              /* Set character box to              */
                              /* 4.0 times default size.           */
```

This pair of statements was used in the GDDM program that produced Figure 23.

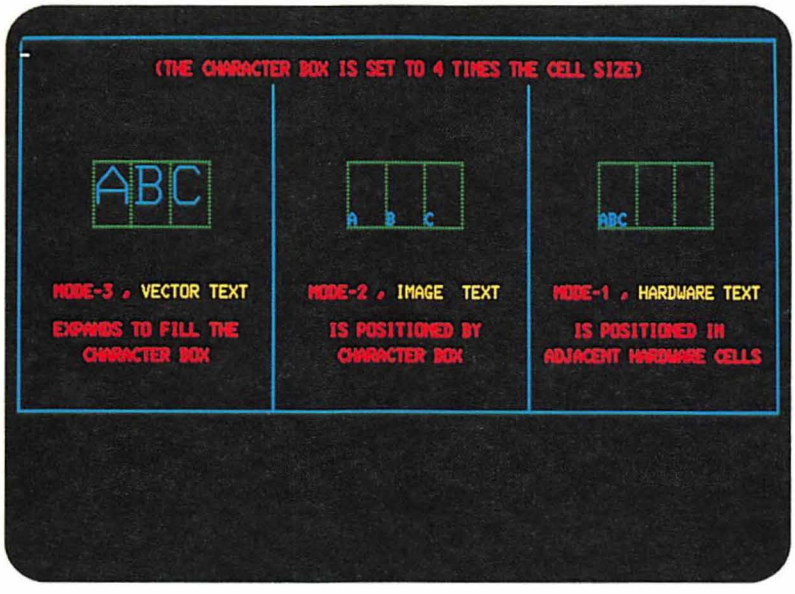


Figure 23. Effect of character-box attribute on the three text modes

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, (see the *GDDM Base Programming Reference, Volume 2* for details), and you can create your own using the Vector Symbol Editor. In the latter case, you assign a width to each character, and the editor records, as part of the character's definition, the ratio between its assigned width and the maximum. Altering the width of a character **does not** alter the size of the character box.

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 24.

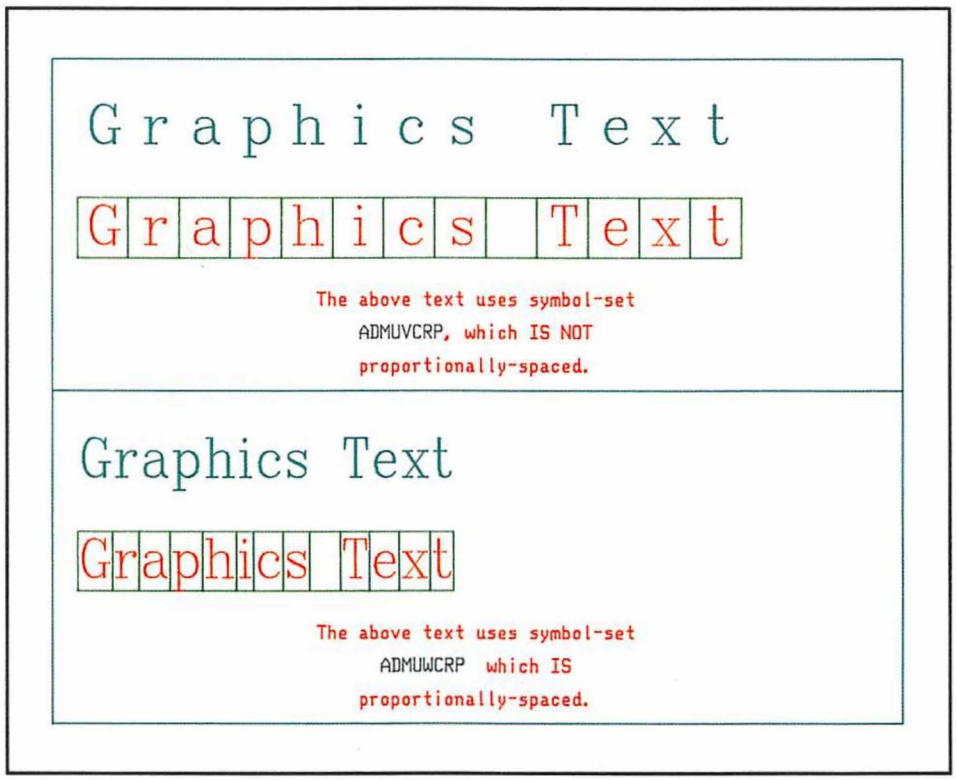


Figure 24. Effects of proportional spacing

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 "The text box" on page 65).

For mode-2 and mode-3 characters, you can also control the amount of space between character boxes, using the character box spacing attribute. See "Setting the character-box spacing attribute, using call GSCBS" on page 65.

Setting the character angle attribute, using call GSCA

This specifies the angle of an imaginary base line along which the characters will be written. The angle is specified as a ratio between the required x and y increments, dx and dy. This is a typical call:

```
CALL GSCA(2.0,1.0);    /* Set character angle of dx=2.0, dy=1.0 */
```

The angle will be that obtained by moving 2 x units (measured in world coordinates) in the x direction and 1 y unit (again measured in world coordinates) in the y direction.

When the graphics window has been chosen so that 1 x unit is physically equal to 1 y unit (see discussion in "The graphics window" on page 101), the angle of the base line will be given by arctan(dy/dx). Or, to get an angle A, you should set the parameter dx = cos(A) and dy = sin(A). For some angles one or both of the parameters will be negative.

Setting a character angle has a different effect on each of the three modes:

Mode-1 The attribute is ignored.

Mode-2 The character boxes are placed side by side along the base-line, but the characters themselves are not rotated. As with character box, the attribute affects the **positioning** of mode-2 text but not its appearance. The lower left-hand corner of each mode-2 character will be placed at the lower left-hand corner of each (tilted) character box.

Mode-3 Character boxes of the specified (or defaulted size) will be placed side by side along the base-line. The vector symbols will fill these (tilted) character boxes. In other words, each character will be rotated so that its base lies on the baseline.

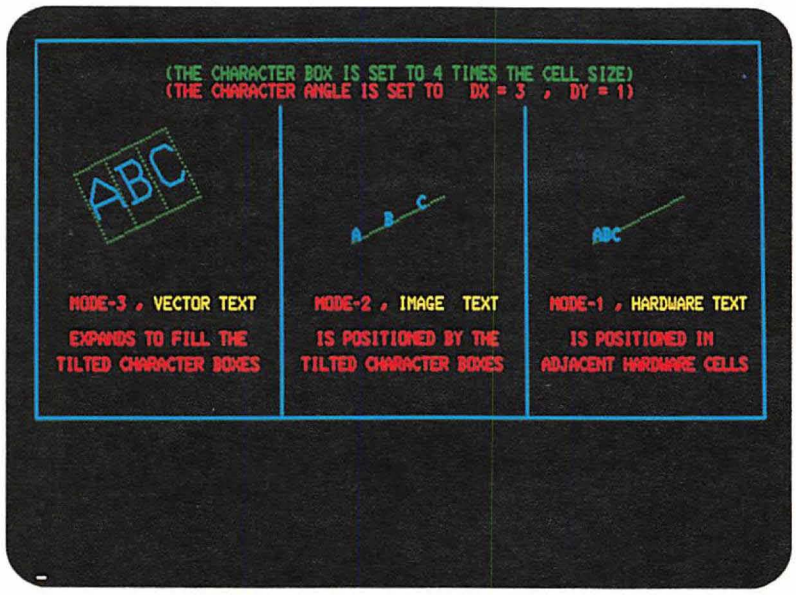


Figure 25. Effect of character-angle attribute on the three text modes

Figure 25 shows the effect of the above GSCA call on text of the three different modes.

Changing the character direction attribute, using call GSCD

This attribute provides support for languages that are not written in the European left-to-right fashion. This is a typical call:

```
CALL GSCD(2);          /* Set character direction to downward.   */
```

After this call, a GSCHAR of the string ABC would appear as:

A
B
C

This is the standard direction for Chinese text. It might also be used to annotate the y axis of a business chart.

There are four possible values for the single parameter:

1 Normal direction (left to right)
2 Downward
3 Right to left
4 Upward.

A new line is placed below the previous one for directions 1 and 3, and to the left for directions 2 and 4.

Of course, this attribute does not act independently. It interacts with other attributes such as character box and angle. This is the effect of setting a downward direction for the three different modes:

Mode-1 The attribute is supported by using successive character positions running in the appropriate direction. This means that successive cells running in the appropriate direction are used. Note that the character angle is always ignored for mode-1. A GSCD downward request has the same effect whether the character angle is set to 0, 90, or 180 degrees, or some sloping angle.

Mode-2 The character boxes are placed as for mode-3. The image symbols are positioned at the bottom left of the character boxes, as always.

Mode-3 The first character box is placed on the (possibly tilted) base line. The next character box is placed **underneath** it, with the top of the character box on the base line. Further character boxes are placed similarly.

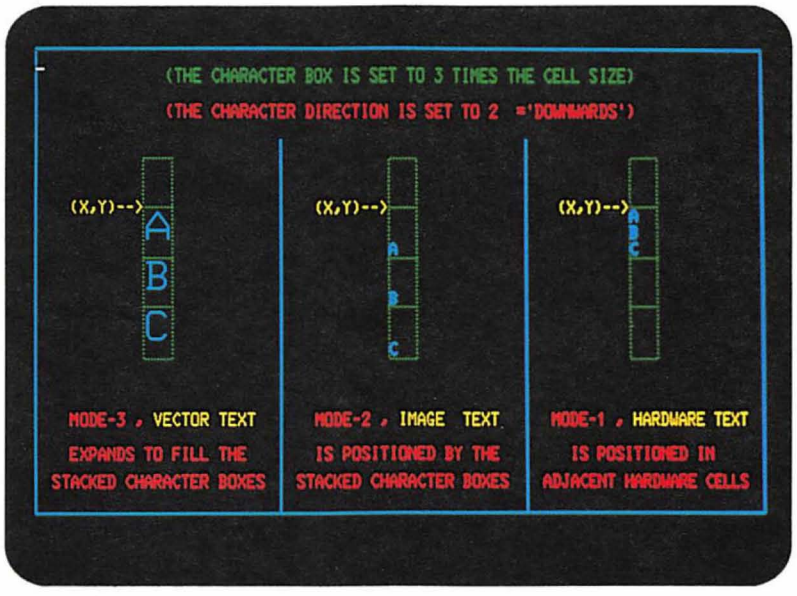


Figure 26. Effect of character-direction attribute on the three text modes

Figure 26 shows the effect of the above GSCD call on text of the three different modes displayed on a color graphics display.

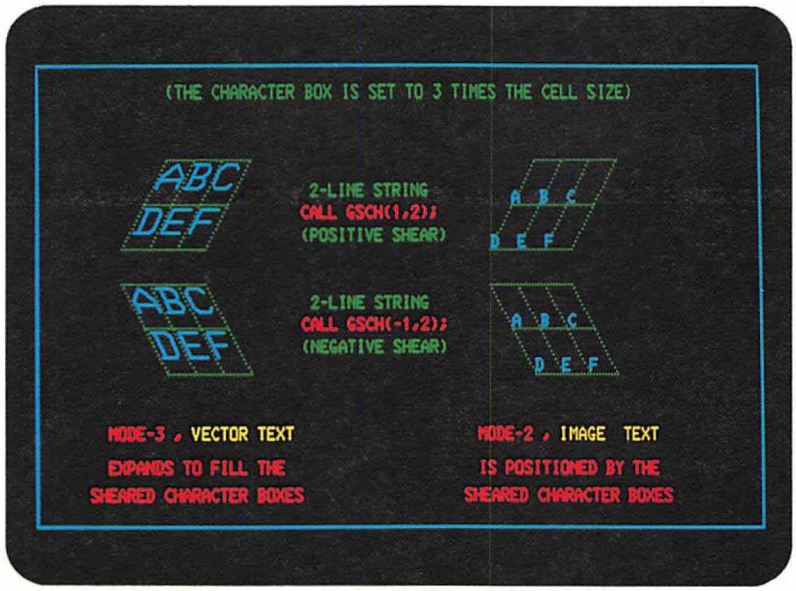


Figure 27. Effect of character-shear attribute on image and vector text

Shearing characters attribute, using call GSCH

This attribute gives an italicizing effect on mode-3 symbols by shearing the top of each character box to the right or the left. The amount of shear is given in the same way as the character angle was specified – by stating a dx and a dy. If dx and dy are positive, the characters will slope forward. If dx is negative, they will slope backward. This is a typical call:

```
CALL GSCH(1.0,3.0); /*Shear the characters right, dx=1.0, dy=3.0*/
```

As with GSCA, the parameters express a ratio. They are in world coordinates (not absolute units).

This will be the effect of the call on the three different modes:

Mode-1 The attribute is ignored.

Mode-2 The attribute has no effect on the appearance of individual characters nor on the positioning of characters in a single line of text. If image symbols are used, the characters will be placed in the bottom left of the character boxes, as usual.

The attribute does have an effect on positioning when there is more than one line of text. The boxes of the second and subsequent rows will be placed so that their tops coincide with the bottoms of those in the previous row. A block of several equal-length lines of text will itself then form a parallelogram.

Mode-3 The first line of character boxes is placed along the base-line specified by the character angle (if any). The tops of each box are now sheared (parallel to the base line) to form parallelograms. The mode-3 symbols are now transformed to fit accurately into these character boxes. If there are two or more lines of text, then, as explained for mode-2 text,

each line of character boxes will be offset from the previous one because of the alignment of the parallelogram character boxes.

Figure 27 on page 64 shows the effect of character shear, both on the positioning of the character boxes and on the drawing of each character.

Setting the character-box spacing attribute, using call GSCBS

This attribute gives you control over the spacing between character boxes in a text string. Once it has been set, it applies to all mode-2 and mode-3 text. For mode-1 text, it is ignored. This is a typical call:

```
CALL GSCBS(0.9,3.0);                    /* Set character box spacing.*/
```

The parameters are the width multiplier and the height multiplier. Both parameters are multipliers of the dimensions of the character box. A positive multiplier will put extra space between character boxes. A negative multiplier can be used to overlap character boxes. A value of zero in a multiplier gives standard spacing (the default). For any individual symbol set, whether proportionally or non-proportionally spaced, the dimensions of the character box are constant.

The width multiplier is specified as a fraction of the width of the current character box, and affects the horizontal space between character boxes.

The height multiplier is specified as a fraction of the height of the current character box, and affects the vertical space between character boxes.

The effect of the multipliers depends on the direction of the text. See the *GDDM Base Programming Reference* manual for details.

Characters in proportionally spaced vector symbol sets will still have their individual widths, but will be separated by the specified or defaulted character-box space.

The text box

The set of character boxes in which the text string specified in a GSCHAR or GSCHAP call is drawn are conceptually enclosed within a rectangle or parallelogram called a **text box**.

If you allow the character-box space to default, the set of character boxes will be contiguous.

The dimensions of the text box for left-to-right text will therefore be:

For a string containing no new-line characters, the height of the text box will be the same as the character-box height, and the width will be equal to an exact number of character-box widths.

If there are new-line characters, the box will be equal in depth to the character-box height multiplied by the number of lines, and as wide as the longest line.

If you use a non-default character-box space, or proportionally spaced vector symbols, the width of the text box will not be a simple multiple of the character box width. For example, with non-default character-box spacing, the dimensions of the text box have to take account of the appropriate number of character box spaces.

You can use the GSQTB call to find out the positions of the corners of the box, and the current position after the characters have been drawn. You will be aware of a particular need for it if you use character-box spacing or proportionally spaced vector symbols.

Here is an example:

```
DCL XCOORDS(5) FLOAT DEC(6),
    YCOORDS(5) FLOAT DEC(6);
     /* Length   String   Count   Returned coordinates */
CALL GSQTB(3,     'ABC',    5,      XCOORDS,YCOORDS);
```

The first parameter is the length of the string, and the second, its contents. The last two parameters are arrays in which GDDM returns information about the text box. The third parameter specifies the number of elements in these arrays.

The arrays can have up to five elements each. In the first four, GDDM returns the positions of the corners of the text box **as offsets from the starting point of the string**. Their order is: top left, bottom left, top right, bottom right. Precise definitions of these terms are given in *GDDM Base Programming Reference* manual. 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.

You should note that all the offsets are always returned as if the starting point of the string is at 0,0. This means that you have to add the actual coordinates of the starting point of the string to the returned offsets to get the actual positions of the corners of the text box. For example, the following section of sample code adds the first four offsets to the actual position of the starting point, to draw a line around the string:

```
DCL XC(5) FLOAT DEC(6),          /* Declare arrays for           */
    YC(5) FLOAT DEC(6);          /* GSQTB call                   */

DCL NEWLINE CHAR(1);             /* Declare new line character   */
UNSPEC(NEWLINE) = '00010101'B;   /* and initialize it.           */

CALL GSCM(3);                    /* Specify mode-3 (vector text) */

/* Now write the string of characters and query their text box  */
CALL GSCHAR  (X,Y,19,'CURRENT'||NEWLINE||'EXPENDITURE');
CALL GSQTB       (19,'CURRENT'||NEWLINE||'EXPENDITURE',5,XC,YC);

CALL GSMOVE(X+XC(1),Y+YC(1)); /* Move to bottom left of text box*/
CALL GSLINE(X+XC(3),Y+YC(3)); /* Draw around ...                */
CALL GSLINE(X+XC(4),Y+YC(4)); /*              the ...           */
CALL GSLINE(X+XC(2),Y+YC(2)); /*                       text box */
CALL GSLINE(X+XC(1),Y+YC(1)); /*                                */

CALL GSMOVE(X+XC(5)+10,Y+YC(5));/* Move to 10 x units along from*/
               .                /* what was current position    */
               .                /* after text was written.      */
               .
```

The primary application of the GSQTB calls is with proportionally spaced mode-3 vector symbols. It can be used with mode-1 and mode-2 text.

In the case of mode-2 text, it is particularly important to remember that the call returns the coordinates of a box that encloses the character boxes within the string, not the symbols. Image symbols do not necessarily fill the character boxes, and can also extend outside them, as can be seen from Figure 27 on page 64. And

if the boxes are angled, their edges will be staircased. In all cases, the text box runs through the extremities of the character boxes.

Setting the text alignment attribute, using call GSTA

If you allow the text alignment attribute to default, text is aligned such that a point on the text box corresponds with either the position specified in the x and y coordinates in the GSCHAR parameters, 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. For example, if you have a normal graphics window, the GDDM default character angle, direction, and shear, and the width and height of the character box are both positive values, the alignment point will be the bottom-left corner of the leftmost character box in the first row of text. Default alignment points for other character directions are given below.

You can use the text alignment attribute call to alter the alignment point of a text box. This is a typical call:

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

The call is valid for all three modes of graphics text. The first parameter horizontally aligns the text box. It has the following possible values:

-1 Alignment according to character direction:

Direction	Alignment
Left to right	Left edge of first character
Downward	Left edge of first character
Right to left	Right edge of first character
Upward	Left edge of first character

0 The default (initially the same as -1).

1 Alignment according to current character direction:

Direction	Alignment
Left to right	Left edge of text box
Downward	Left edge of text box
Right to left	Right edge of text box
Upward	Left edge of text box

2 Left edge of text box

3 Center (arithmetic mean of left and right edges of text box)

4 Right edge of text box

The second parameter vertically aligns the text box. It has the following possible values:

-1 Alignment according to current character direction:

Direction	Alignment
Left to right	Bottom edge of first character
Downward	Top edge of first character
Right to left	Bottom edge of first character
Upward	Bottom edge of first character

0 The default (initially the same as -1)

1 Alignment according to current character direction:

Direction	Alignment
Left to right	Bottom edge of text box
Downward	Top edge of text box
Right to left	Bottom edge of text box
Upward	Bottom edge of text box

2 Top edge of text box

3 Cap of character furthest towards top of text box (see note).

4 Center (arithmetic mean of top and bottom edges of text box)

5 Base of character furthest towards bottom of text box (see note).

6 Bottom edge of text box

Note: Vertical parameter values of 3 and 5 apply only to symbol sets where the positions of bases and caps are defined. GDDM-supplied symbol sets do not define these positions; parameter value 3 will therefore have the same effect as value 2, and value 5 will have the same effect as value 6.

If you have a normal graphics window, the GDDM default character angle, direction, and shear, and the width and height of the character box are both positive values, the meanings of terms like "top-left" and "bottom-right" are obvious. The meanings are not so obvious when text is rotated or sheared. For example, the term "top-left" actually refers to the corner of the text box that is top-left when **no** rotation or shearing is applied. There is an illustration of this in the coverage of GSQTB in the *GDDM Base Programming Reference* manual.

Also, if you change the direction of the graphics window so that, for example, low x values lie on the right-hand side of the display, the term "left" will apply to the side of the display corresponding to low x values. The same principle applies to changing the direction of the graphics window in the y direction.

Example using graphics text attributes

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, and character symbol-set. Whenever some graphics text is written (with a GSCHAR or GSCHAP call), the current values of these eight attributes will apply, whether they have been explicitly set or defaulted. Here is an example using the first five attributes. The symbol set attribute is discussed in "Chapter 15. Symbol sets" on page 219.

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

CALL GSCHAR(4.0,8.0,3,'ABC');/* Mode-1, color green, default    */
                                /* direction.                       */

CALL GSCA(1.0,1.0);             /* Set character angle to dx=1.0,   */
                                /* dy=1.0 (45 degrees above         */
                                /* horizontal, if                   */
                                /* 1 x unit = 1 y unit)             */

CALL GSCB(8.0,6.0);             /* Set character box to             */
                                /* 8 x units by 6 y units.          */

CALL GSCHAR(24.0,30.0,3,'GHI');
                                /* Mode-1, color green, default     */
                                /* direction  (still).              */

CALL GSCM(3);                   /* Set mode to 3 - vector text.     */
CALL GSCHAR(60.0,45.0,5,'PQRST');
                                /* Green vector characters.         */
                                /* The string and each character    */
                                /* tilted at 45 degrees, each       */
                                /* character of size 8 by 6         */
                                /* in world coordinates.            */

CALL GSCH(-1.0,5.0);            /* Request backward shear.          */
CALL GSCHAR(10.0,15.0,2,'YZ');
                                /* Same as previous string except   */
                                /* that the top of each character   */
                                /* is sheared to the left.          */

CALL GSCM(2);                   /* Set mode-2 - image characters.   */
CALL GSCOL(6);                  /* Change color to yellow.          */
CALL GSCHAR(50.0,50.0,4,'JKLM');
                                /* Yellow image characters.         */
                                /* The string slopes at an          */
                                /* angle of 45 degrees but the      */
                                /* individual characters are        */
                                /* not rotated or sheared.          */

CALL GSCD(2);                   /* Set downward character direction*/
CALL GSCM(3);                   /* Revert to vector characters.     */
CALL GSCHAR(20.0,90.0,2,'OP');
                                /* Yellow sheared vector characters*/
                                /* - each character is rotated 45   */
                                /* degrees and placed beneath the   */
                                /* previous one. The text string    */
                                /* is therefore at an angle of      */
                                /* minus 45 degrees to horizontal.  */

CALL ASREAD(TYPE,MOD,COUNT); /* Send out all the graphics text.  */
```

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 attributes (such as character mode). If such a default is subsequently changed (from mode-3 to mode-2, for example) the appearance at ASREAD will be affected.

Device variations

The preceding sections of this chapter refer primarily to members of the 3270 family that use programmed symbols for graphics, such as the 3279. However, most 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.

Differences on the IBM 3179-G Color Display Station

Mode-1 text Graphics and text are presented on an equal footing: where they coincide, both are displayed. Mode-1 does not have the advantage that the text is always the sole occupant of the text box.

Differences on the IBM 3270-PC/G and /GX work stations

Mode-1 text Graphics and text are presented on an equal footing: where they coincide, both are displayed. Mode-1 does not have the advantage that the text is always the sole occupant of the text box.

The symbols are not located in hardware-defined cells. They can be of any size. The start of the string is positioned to pixel accuracy.

Mode-2 and -3 text The work station has a hardware image and vector symbol set. These 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 (see "Default symbol sets for graphics text" on page 389).

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. Graphics cells have a predefined size, but not predefined locations.

Differences on the 5080 Graphics System

Mode-1 text As for 3270-PC/G and /GX, above.

Mode-2 text Other data in the cell is obscured by the text.

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

Differences on 5550-family multistations

Mode-1 and -2 text The same as for 3270-PC/G and /GX

Mode-3 text The same as for 3270-PC/G and /GX, if Japanese 3270-PC/G software from Version 6 and onward is used.

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

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

Differences on composed-page printers

This section describes how text on composed-page printers, such as the IBM 4250 and the 3800 Models 3 and 8, differs from text on the ordinary members of the IBM 3270 family, like the 3279:

Mode-1 text and graphics
Graphics and text are presented on an equal footing: where they coincide, both are displayed. Mode-1 does not have the advantage that the text is always the sole occupant of the text box.

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 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.

Differences on plotters

Some special considerations for plotters are described in "Symbol sets" on page 439.

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 426.

If no image symbol set is loaded by the program, the default vector symbol set ADMDVSS 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 426.

Advantages and disadvantages of each character mode

Each of the three character modes has its own advantages that will prove the best choice in particular situations. These are the main features of each mode:

Mode-1: String positioning

Advantages: This is the cheapest mode to use as very little processing is required by GDDM. Multicolored symbols are permitted, except on the IBM 5080 Graphics System. On devices in the IBM 3270 family (except the 3270-PC/G and /GX), the fact that mode-1 text is the sole occupant of its cells aids its readability where text and graphics coincide. Other modes will 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 will therefore vary 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.

- Composed-page 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 positioning

Advantages: The limitations on character size and positioning mentioned for mode-1 can be avoided. You can use image symbols. Multicolored symbols are again permitted, except on the IBM 5080 Graphics System. 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 positioning

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 these devices.

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

Chapter 8. Basic alphanumerics

This chapter introduces the facilities that GDDM provides for output and input of alphanumeric data.

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, as the terms are synonymous). GDDM allows 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 mapping. An introduction to mapping, and guidance on when to use it in preference to procedural alphanumerics, are given in "Chapter 17. Mapped alphanumerics" on page 251.

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 that they combine depends on the device. On the 3179-G, 3270-PC/G and /GX family, and 5550 family, you can control the precedence of alphanumerics over graphics. See "Device variations" on page 86. On a 3279, the alphanumerics take precedence; no graphics will 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 and the 5080 Graphics System), the graphics are displayed on one screen and the alphanumerics on another. See "IBM 5080 graphics system" on page 87 for details of alphanumerics on the 5080.

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);
```

The six parameters have these meanings:

3 The **field identifier**. Any later call that refers to the new field will use this identifier (in other words, it is the name of the field). If a field with identifier 3 already exists, the new field replaces the old one.

14 The row in which the data of the alphanumeric field will start. The rows are numbered from the top.

5 The column of the first data position in the field.

1 States that the field will have only 1 row.

21 Gives the width. It will be 21 columns across.

2 Specifies the type of the field - how it should be handled by the terminal. These are the possible settings:

- 0 Unprotected alphanumeric. "Unprotected" means that the operator may type data into the field.
- 1 Alphanumeric output, numeric input. Also unprotected - but the field will accept numeric input only. If the terminal does not support this feature, this setting is equivalent to 0.
- 2 Protected alphanumeric. The keyboard will lock if the operator tries to type into the field.
- 3-6 Various types of light-pen field. The field will be sensitive to the light-pen if the terminal has this feature.

Note that 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/column coordinates.

Sending and Receiving alphanumeric data

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 will be transmitted to the program when the terminal operator presses ENTER (or causes any other interrupt). The program may 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 will wrap the field around the screen and continue 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 will have its top left-hand corner character in row 4, column 21, and will 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 will be 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

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 will cease to exist.

To delete all the alphanumeric fields in the page (and the graphics too), you must call FSPCLR (see "The page and page window" on page 93).

Positioning and querying the alphanumeric cursor

You can set the position of the cursor with a call to ASFCUR. If the operator is expected to type some information, it will probably be 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 */
```

As you would expect, 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 other two 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  */
```

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 will set 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 will be returned in field coordinates. F_IDENT will be set to the alphanumeric field identifier and ROW and COLUMN will 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 will be set to 0 and **page** coordinates will be 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 "IBM 5550 multistation" on page 245.

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 will result but the last byte of the field on the left will lose its data and appear blank.

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

The representation in the buffer will include trailing attribute bytes to end each field. The default setting for this trailing attribute is auto-skip, meaning that the cursor will automatically jump to the next unprotected field when the current field is 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. You need therefore allow only 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.

Field attributes

These attributes affect the way the terminal handles the fields, and also their appearance. There are a number of different attributes that you may set:

- **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 75). 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 */
  ```

  These are the possible settings of the second parameter:

  -1 Leave type as it is
  0 Unprotected alphanumeric
  1 Alphanumeric output, numeric input
  2 Protected alphanumeric
  3 Light-pen attention field
  4 Light-pen selection field
  5 Light-pen enter field
  6 General light-pen field.

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

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

  The second parameter may take these values:

  -1 Leave intensity as it is
  0 Invisible
  1 Normal (the default)
  2 Bright.

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

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

  These are the possible settings:

  -1 Leave color as it is
  0 Default
  1 Blue
  2 Red
  3 Pink
  4 Green
  5 Turquoise
  6 Yellow
  7 Neutral (white on displays, black on printers).

Codes 0 to 7 are used in other calls that refer to these colors. A suggested mnemonic for the codes for blue through neutral is:

```
Boys Reading Politics Go To Yale Now
```

If the field's symbol set is multicolored, the color must be set to 7 (neutral) (see "Multicolored symbols" on page 228 for more details).

- **Symbol set.** The symbol set to be used may be specified by a call such as CALL ASFPSS(6,196). Field 6 would now be displayed using the symbols of symbol-set 196 (see "Chapter 15. Symbol sets" on page 219 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 will be underscored */
```

The possible settings are:

-1 Leave highlight as it is
0 Normal
1 Blink
2 Reverse-video (that is, neutral characters on a colored background)
4 Underscore.

- **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 allow 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 tables themselves are established by calling ASDTRN, as described in the *GDDM Base Programming Reference* manual.

- **Transparency**. You can allow graphics on the screen to extend into the cells of alphanumeric characters using an ASFTRA call (see "Device variations" on page 86).

- **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 "Double-byte character set alphanumerics" on page 245.

- **Field outlining**. An outline can be drawn around a field with an ASFBDY call (see "Field outlining on the IBM 5550 multistation" on page 249).

The first three fields in Figure 28 illustrate the use of field attributes.

Figure 28. Using alphanumeric field and character attributes

Character attributes

For some of the attributes, such as color, it may be desirable to have different values within an alphanumeric field. GDDM permits the setting of character attributes for color, symbol set, and highlighting. The last three fields in Figure 28 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 will 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. They may 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  */
```

The possible attribute settings are similar to those for ASFHLT:

" " (Blank) states that the attribute should be inherited from the field highlight attribute.
1 Blink
2 Reverse-video
4 Underscore.

Note that a setting of 0 is not permitted when using character highlight attributes.

The third of the character attribute statements (ASCSS) is described in "Chapter 15. Symbol sets" on page 219.

The terminal operator may set character attributes, if the device has the requisite keyboard. If the "red" button is pressed on the keyboard, all characters that are entered will have 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 will be ignored.

The program can discover what character attributes were set by the terminal operator 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.    */
                   (See ]Chapter 15.  Symbol sets≠ on page 219).
```

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

Sample alphanumerics program

Figure 29 shows a simple alphanumeric program that gives details of a bank customer and his account balances in response to a typed-in account number. Sample output from this program is shown in Figure 30 on page 85.

If the account number is invalid, an error message is issued. If an account is overdrawn, the balance is displayed in red.

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 will sound on the next screen output.

```
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:');
```

Figure 29 (Part 1 of 2). "Bank Account" sample alphanumerics program

```
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.  */
     /*****************************************/
     /* 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 29 (Part 2 of 2). "Bank Account" sample alphanumerics program

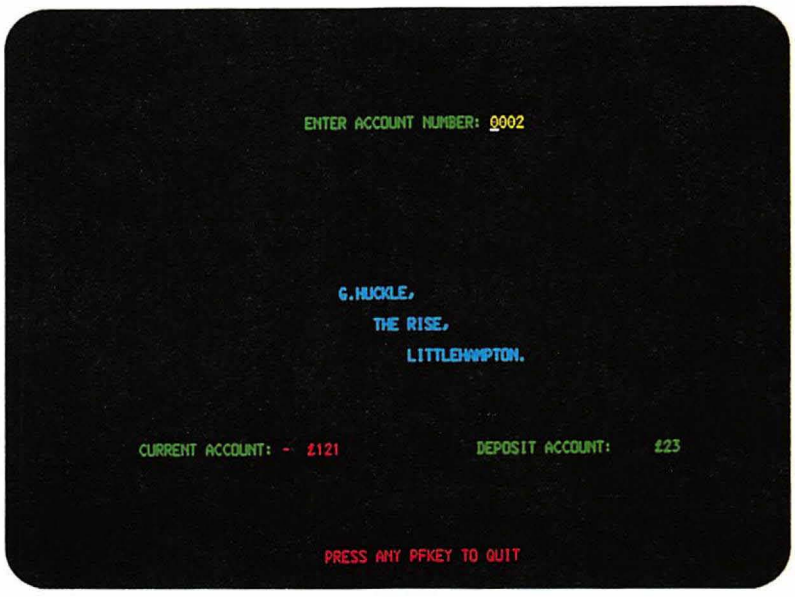


Figure 30. Output from "Bank Account" sample alphanumerics program

Mixing graphics and alphanumerics

You may freely mix GDDM calls that refer to alphanumerics with those that refer to graphics. The alphanumeric and the graphical data will be added to the current page, although GDDM will hold 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 will send first the graphics, then the alphanumerics. Those hardware cells that are part of alphanumeric fields will contain no graphics at all - only the alphanumerics will appear (except in the case described in "Device variations" on page 86).

The program shown in Figure 31 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 */
```

Figure 31 (Part 1 of 2). Part number sample alphanumerics program

```
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.         */
/*************************************/
/* 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 31 (Part 2 of 2). Part number sample alphanumerics program

Device variations

3179-G, 3270-PC/G and /GX family, and the 4224 printer

As explained in "Mixing graphics and alphanumerics" on page 85, hardware cells used for alphanumerics contain, by default, no graphics. On the 3270-PC/G and /GX work stations, 3179-G color display stations, and 4224 printers, you can allow the cell background to become transparent. The alphanumeric characters will 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 alphanumeric fields will also be honored.

IBM 5080 graphics system

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 work station. Alphanumeric output goes only to the 3270 screen. For more information about the 5080, see "Processing option for the 5080 graphics system" on page 390 and "Interactive graphics on the IBM 5080 graphics system" on page 214.

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 will be transparent.

Chapter 9. Hierarchy of GDDM concepts

You have already seen that you can code simple graphics programs with only a little knowledge of graphics concepts. This chapter looks at some of the concepts that you can use in more advanced programs, considering first just the physical subdivision of the screen (or the printer page). There is a hierarchy of objects, each fitting inside the previous one.

The GDDM hierarchy

These are the objects in the hierarchy:

1. The device
2. The partition set and partition
3. The page
4. The graphics field
5. The picture space
6. The viewport.

The above hierarchy (except for the partition set and partition) is shown in Figure 37 on page 101. These hierarchical objects are present, even in the simplest of GDDM programs. Often, though, you do not need to specify many of them in your program. You can leave them to take the default values.

Those objects that are specified explicitly must be defined in the appropriate order (that is, moving down the hierarchy). For example, you can define objects 2 and 4, or objects 1, 2, and 6. You cannot define object 6, then object 4.

Independently of these objects you may define:

7. The graphics window. This is the coordinate system to be used when specifying the graphics.

When these seven have been defined or defaulted, you can open a:

8. Graphics segment.

   This is a means of grouping together logically connected primitives and their attribute settings. It cannot be defaulted, but is not mandatory: primitives can be drawn outside segments.

The above basic hierarchy essentially relates to graphics objects. There are a number of additional objects that you can use in GDDM programs. They are virtual devices and operator windows, alphanumeric fields, alphanumeric maps, and image fields.

Virtual devices and operator windows are introduced under the following section "The device" on page 90 and that is the level at which they fit into the hierarchy.

You can consider alphanumeric fields, alphanumeric maps, and image fields as all being at the same level as the graphics field, within a page.

To understand which of all the objects in the entire hierarchy you need to define explicitly for a given program, it is necessary to explain the nature and purpose of each one.

The device

The device is the highest level in the hierarchy. Your program can select a device to be used as either the current primary device, or as the current alternate device. All commands will refer to the communication with that device until a different current primary or alternate device is selected. The call that opens a device is DSOPEN. The call that makes a device the current primary or alternate device is DSUSE. There is generally no need to issue these calls when you want the output to appear on the invoking device only. This is because the default primary device is the invoking terminal, so GDDM issues its own internal DSOPEN and DSUSE. However, if you want 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, then you can issue your own DSOPEN with different parameters. Another way is to use **nicknames** to modify the internal DSOPEN. Nicknames can also be used to modify your own DSOPEN calls.

If you use the (WINDOW,YES) processing 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. You can therefore 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 DSOPENS for the same device will 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.

In addition, the **terminal user** can control the size, position, and viewing priority of operator windows at the screen, without interacting with any application program, by using the user control function.

Operator windows can also be used within a single application program, to share out a real screen between several virtual devices, and to provide the various functions of the single application in separate and independent areas of the screen. Therefore, 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.

For more information about operator windows, see "Operator windows" on page 467.

The principal calls are:

DSOPEN Open a device

DSUSE Specify device usage

DSDROP Discontinue device usage

DSCLS Close a device.

These calls and the use of nicknames are described more fully in "Chapter 21. Device support" on page 367.

The partition set and partition

For a real or virtual device, you can create several **alternative** logical screens in an application. Each logical screen is called a partition set, and only one per virtual device can be shown to the terminal user at any time.

A partition set can be divided into one or more independent rectangular partitions, which may overlap. It is possible for the terminal user to view and interact with an application through more than one partition within a partition set, at the same time.

The partition set is created by the PTSCRT call. Its main purpose is to define a grid of rows and columns to be used for specifying the sizes and positions of all the partitions in the partition set. A partition is created by the PTNCRT call. The default partition size is equal to the partition set grid. For details of the default partition-set grid size, see the description of the FSQURY call in the *GDDM Base Programming Reference* manual. An example 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 change the attributes (size, position, order of viewing priority, and visibility) of partitions.

If no PTSCRT call has been issued, no PTNCRT call can be issued: a default partition covering the complete screen is created.

Figure 32 on page 92 illustrates a partitioned screen based on the default partition set grid.

Partitioning is supported on all family-1 display devices including the 3179-G, the 3270-PC/G and /GX family, and 3193 Display Station. The IBM 3290 Information Panel, 8775 Display Terminal, and 3193 Display Station each have a hardware-partitioning facility. On ordinary terminals in the 3270 family, such as the 3278 and 3279, and on 3179-G, 3270-PC/G and /GX work stations, and 5550-family

multistations, GDDM emulates hardware partitioning. For all display devices, partitions are emulated when operator windows are used, or when partition overlap is specified, or when user control has been made available to the terminal user.

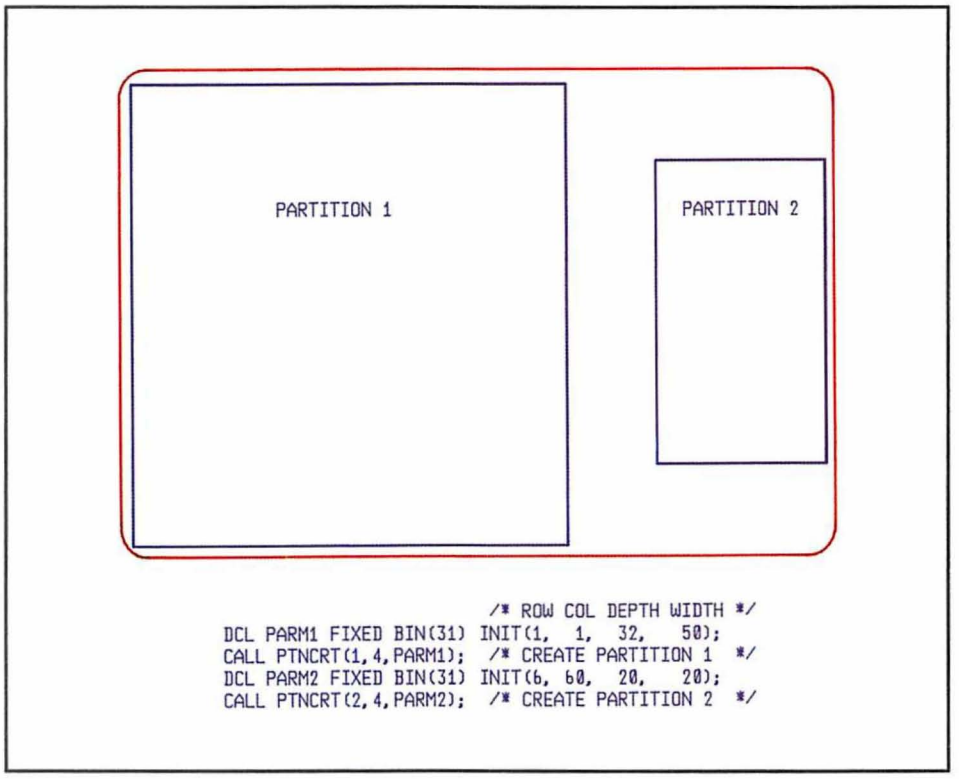


Figure 32. PTNCRT – create a partition

Creating a partition set makes it current, and associates it with the primary device. Each partition set can have one or more partitions associated with it. A partition belongs to the partition set current at the time of creation. Each partition can have a set of pages associated with it. A page belongs to the partition current at the time of creation.

Calls that operate on partitions and partition sets

The following are the principal calls that operate on partitions and partition sets.

PTSCRT Create a partition set

PTSSEL Select a partition set

PTNCRT Create a partition

PTNSEL Select a partition

PTNMOD Modify a partition.

The above calls and other partition and partition set calls are described in "Partitions" on page 441.

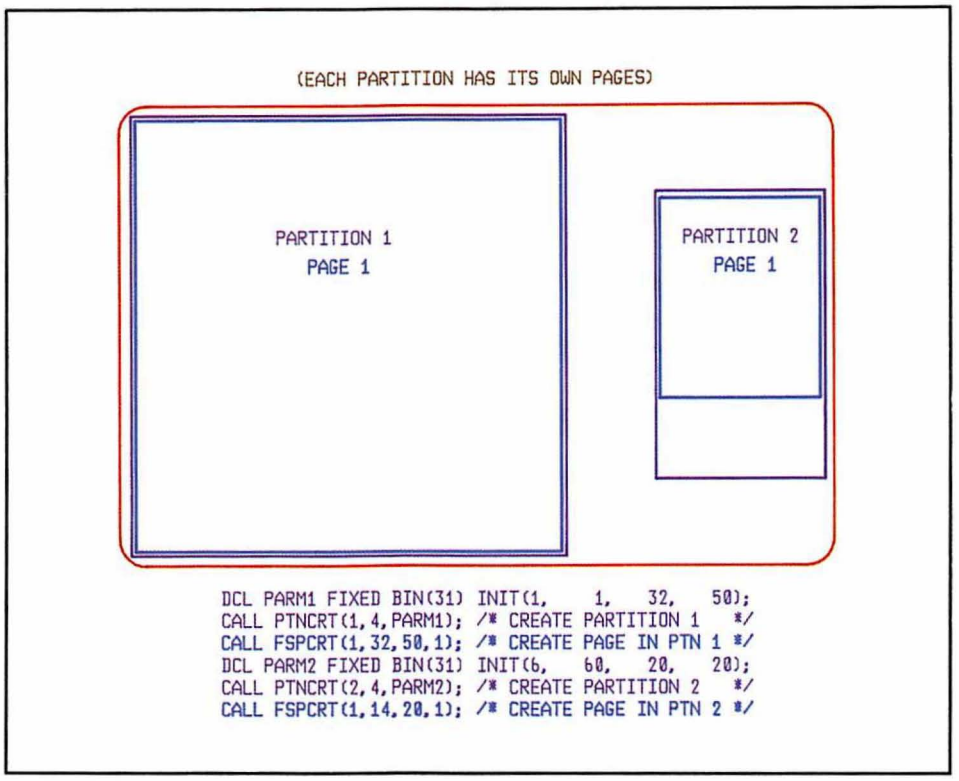


Figure 33. FSPCRT – defining a page

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. An ASREAD, GSREAD, MSREAD, or FSFRCE call sends the current page of each partition in the current partition set to the primary device, and (except in the case of FSFRCE) awaits the terminal user's input. For hardware partitions, the input can come from any one partition. For emulated partitions, 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 will be displayed.

Usually the page size matches the partition (or the printer page), but when you create the page, you can request a size that is smaller than the partition (or printer page) or, on display devices, larger than the partition. You can control how much of the page is shown on a display device by setting the **page window**. At page-creation time, the page window is set to either the page or partition size, whichever is the smaller. A page window always lies within page and partition boundaries: the page window is positioned where you specify it on the page, and the top-left-hand corner of the page window coincides with the top-left-hand corner of the partition. For 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.

This is a typical page-create call:

```
CALL FSPCRT(2,27,80,0);                          /* Create page 2 */
```

The parameters are as follows:

2	The page identifier - its name. Any positive number may be chosen. Zero is reserved for the default page.
27 and 80	Give the number of rows and columns in the page. If either (or both) of these parameters is zero, the whole partition (or printer page) width or depth (or both) will be used.
0	The last parameter is not used by the current release of GDDM, but must be present and have a value of 0 through 3.

A page may be subdivided into any or all of these:

- a graphics field
- an image field
- procedural alphanumeric fields
- mapped 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 356 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 "Dialog with the terminal operator" on page 256. For example:

```
CALL MSPCRT(3,27,80,'MAP001D6');                 /* Create page 3 */
```

The parameters are the same as for FSPCRT, except that the fourth one is the name of a mapgroup.

Figure 33 on page 93 shows a page in each of two partitions on the screen (which is assumed to be 32 by 80 for the example).

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 will use the default page. This page is of the default size, namely the whole screen (or partition, where supported, or printer page). It has a page identifier of zero.

Calls that operate on pages

These are the calls that operate on pages:

FSPCLR	Delete the graphics field and any image, alphanumeric, or mapped fields within the current page. The page will then be empty, just as it was immediately after its creation (by FSPCRT). In particular, no picture space, viewport, or graphics window will remain. It also resets

any previously created pan and zoom transform. A new graphics field, image field, and some new alphanumeric fields can then be created, if required.

FSPDEL Delete a specified page. All associated storage is freed and the page identifier becomes unknown to GDDM. You may subsequently create another page using the same identifier as before. It is not possible to delete the default page. If the current page is deleted, the default page becomes the current page.

Note the difference between clear and delete. When you clear a page (or a graphics field), it is still there and you may refill it. When you delete a page, it ceases to exist.

FSPQRY Returns information about the page whose identifier was specified in the first parameter.

FSPSEL Select a page. The named page becomes the current one. Any later alphanumeric or graphics statements will refer to the new current page. An output statement will send the new current page to the device.

The previously selected page is left as it was. When it is reselected, processing may continue exactly as it would if you had not broken off to service another page.

FSQCPG Returns the page identifier of the current page

FSQUPG Returns a valid, currently unused page identifier. It allows a program to create a new page whose identifier will not conflict with an existing page, without the need to keep track of the page identifiers currently in use.

A general discussion and example of using two pages is given in "Creating two pages of graphics" on page 107.

The relationship between pages and page windows is described in "Large and small pages" on page 459.

The graphics 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 will exclusively occupy the remainder. This is a typical call:

```
CALL GSFLD(6,21,22,60);                    /* Define graphics field */
```

The parameters are in rows and columns:

6 and 21 — The row and column of the top left-hand corner of the required graphics field. If either is zero, this is taken as a request to delete the graphics field.

22 and 60 — The depth and width of the proposed graphics field, expressed in rows and columns. Again a zero value for either parameter requests deletion of the existing graphics field.

Figure 34 shows the defined graphics field 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.

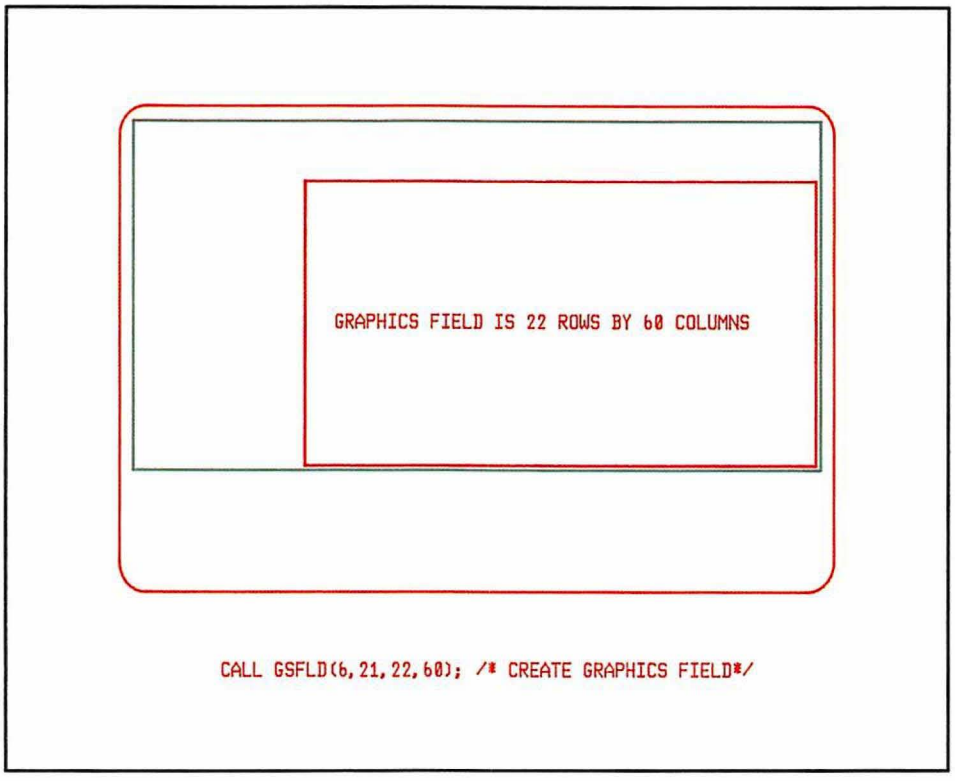


Figure 34. GSFLD – defining a graphics field

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 will apply. This covers the whole of the current page.

Only one graphics field is permitted per page. Definition of a second causes the deletion of the first one and all its contents.

On a dual-screen configuration of the 3270-PC/GX work station, a conceptual grid is imposed on the graphics screen for the purpose of interpreting the parameters of a GSFLD call. The grid has the same number of rows and columns as the alphanumeric screen – the numbers explicitly specified in a page-create call, or the default of 24 rows by 80 columns. The result is that the graphics field is of the same size and is in the same position as it would be in a single-screen configuration.

Note that on ordinary terminals in the 3270 family, such as the 3278 and 3279, (but not on the 3179-G, 3270-PC/G and /GX family, 5550 family, or on printers), the bottom right-hand hardware cell usually 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, there will be a permanent one-cell blank on this edge.

On the 5080 graphics system, GSFLD parameters are handled in the same way as on a dual-screen 3270-PC/GX with graphics capabilities.

You can query the current graphics field like this:

```
DCL (ROW,COL,DEPTH,WIDTH) FIXED BINARY(31);
CALL GSQFLD(ROW,COL,DEPTH,WIDTH);
```

The first two parameters return the row and column position of the top left-hand corner of the graphics field, and the last two parameters its size in rows and columns.

Calls that operate on the graphics field

In addition to GSFLD and GSQFLD, already described, there is:

GSCLR Clears all the graphics from the current graphics field.

The picture space

This call is used when you require a particular aspect ratio for your drawing area. If, for example, you are making a diagram of a factory floor that is 50 m by 30 m, you will need a drawing area of the same ratio.

The graphics field is specified in terms of physical rows and columns. The aspect ratio of the graphics field (that is, the ratio of the physical width and depth) is therefore device-dependent. If you want to ensure a particular ratio for your drawing area, you must specify a picture space:

```
CALL GSPS(1.0,0.5);/*Define picture space where width = depth*2 */
```

The parameters specify the ratio of the width of the picture space to its depth. One parameter must be exactly 1, the other between 0 and 1. Subject to the requested ratio, GDDM will create as large a picture space as possible within the graphics field. Either the horizontal or the vertical boundaries will coincide with those of the graphics field.

Figure 35 on page 98 shows the effect of three different picture-space definitions, each of which lies inside the graphics field that was defined in the previous section.

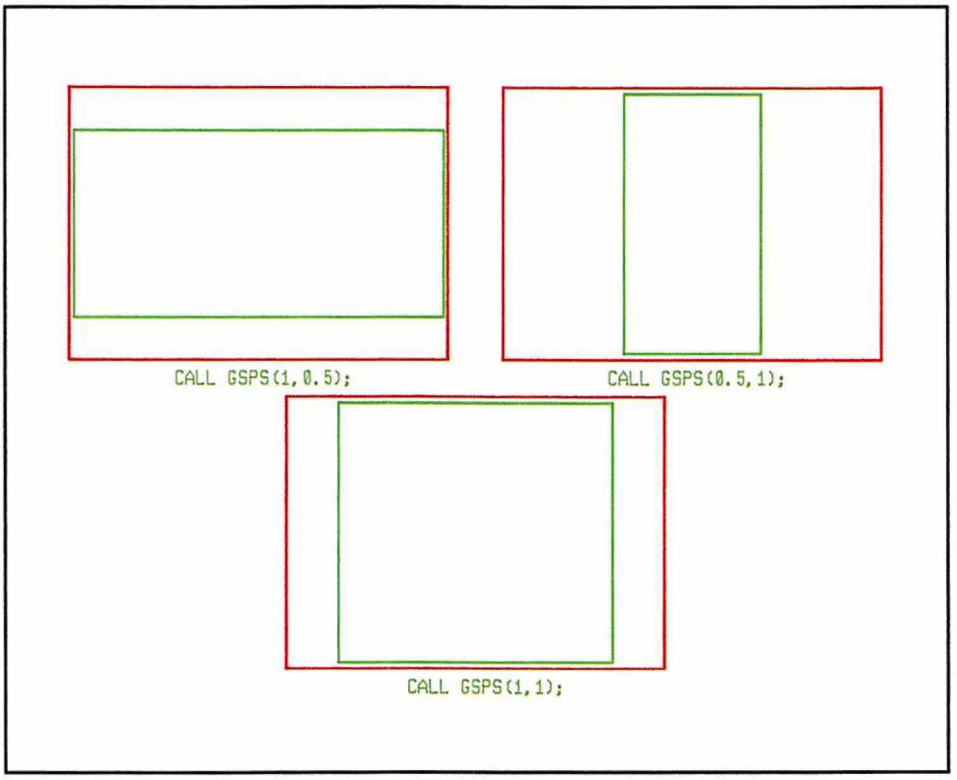


Figure 35. GSPS – defining a picture space

The picture space will be defaulted to the whole graphics field if reference is made to any object lower in the hierarchy.

The viewport

Viewports are not required in a graphics program unless more than one picture (using the term in its everyday sense) is to be sent to the same graphics field. In that case, you may prefer to address separately the drawing area for each picture.

When speaking about subdividing the physical screen area, the viewport lies at the bottom of the hierarchy. It is part of the picture space (or, by default, the whole of the picture space). It is the area of the screen to which the current graphics are to be sent.

The call takes this form:

```
CALL GSVIEW(0.0,0.7,0.25,0.5);          /* Define the viewport */
```

All four parameters are expressed in picture space units. They cannot exceed the parameters specified (or defaulted) for the picture space:

- The first two specify the left and right viewport boundaries. If the picture space was, say, of ratio 1:0.5 (width = 1,depth = 0.5) then this viewport would lie in the leftmost seven-tenths of the picture space.
- The last two parameters specify the lower and upper viewport boundaries. With the same picture space as was used before, the viewport would lie in the top half of the picture space.

Figure 36 shows the positioning of the viewport within the (1:0.5) picture space.

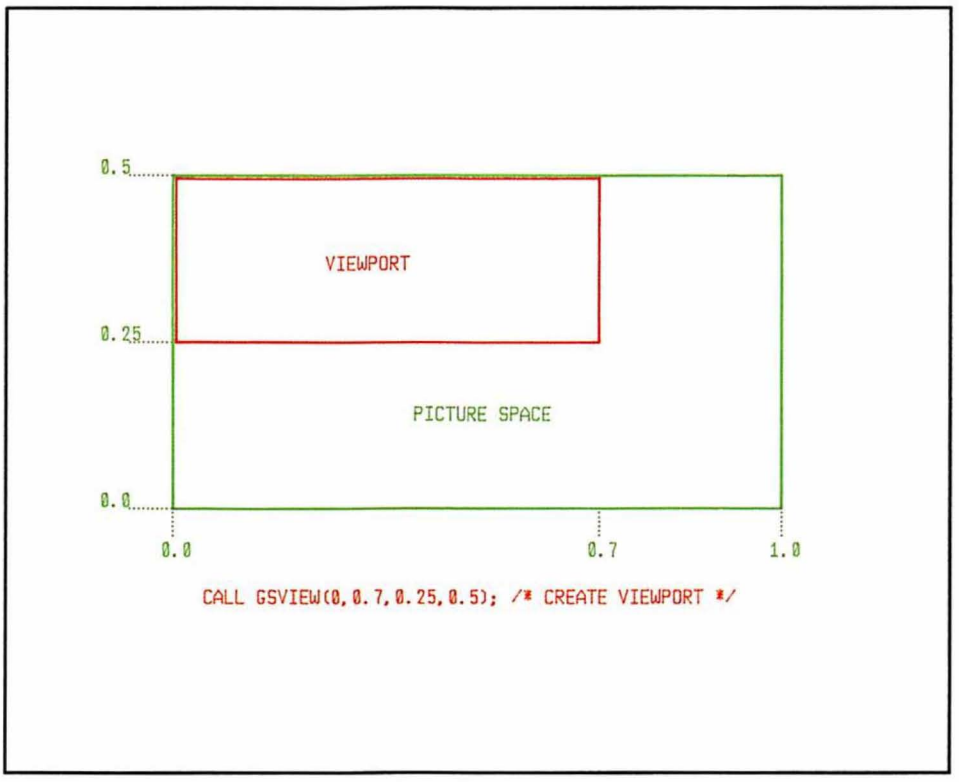


Figure 36. GSVIEW – defining a viewport

There are several points to note about viewports.

- **Querying the picture space**. If the picture space was not explicitly specified, you will not know the picture space coordinates. They are device-dependent. To create a viewport it will be necessary to query the picture space coordinates at execution time. The viewport can then be expressed in terms of the parameters returned from the query. This is a common 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.        */
```

- **Graphics window fits over the viewport**. Graphics primitives are drawn in a coordinate system (called the graphics window) that defaults to 100 by 100, or may be chosen explicitly.

  This coordinate system fits over the viewport. The x range extends over the width of the viewport and the y range over the depth.

  In the very basic example programs shown earlier, it seemed that the coordinate system addressed the whole screen. That was only because the hierarchy of graphics objects had all been allowed to take the default value.

This is often the case – the partition, the page, the graphics field, the picture space, and the viewport all then occupy the whole screen.

- **Clipping at graphics window**. The graphics primitives may be clipped at the edges of the graphics window, as explained in "Graphics clipping" on page 110.

- **Using viewports to get multiple pictures**. Use of viewports permits several different pictures to be combined in one graphics field. Only one viewport exists (per page) at a given time; but several can be created successively, each filled with graphics. Assume, for example, that STARS_AND_STRIPES is a subroutine of graphics calls that fills the viewport with a representation of the American flag. Then these instructions would fill the screen with four flags, one in each quarter of the screen. Note that the subroutine has to be reexecuted for each new viewport.

```
CALL GSQPS(WIDTH,HEIGHT);
                /* Ask GDDM what ratio the picture space has */
CALL GSVIEW(0.0,WIDTH*0.5,0.0,HEIGHT*0.5);
                                        /*Viewport bottom left. */
CALL STARS_AND_STRIPES;                 /* Draw flag.           */
CALL GSVIEW(0.0,WIDTH*0.5,HEIGHT*0.5,HEIGHT);
                                        /* Viewport top left.   */
CALL STARS_AND_STRIPES;                 /* Draw flag.           */
CALL GSVIEW(WIDTH*0.5,WIDTH,0.0,HEIGHT*0.5);
                                        /*Viewport bottom right.*/
CALL STARS_AND_STRIPES;                 /* Draw flag.           */
CALL GSVIEW(WIDTH*0.5,WIDTH,HEIGHT*0.5,HEIGHT);
                                        /* Viewport top right.  */
CALL STARS_AND_STRIPES;                 /* Draw flag.           */
CALL ASREAD(TYPE,MOD,COUNT);            /*Send picture to device*/
  /*** Subroutine to draw American flag ***/
  STARS_AND_STRIPES: PROC;
  CALL GSSEG(0);                /* Open unnamed segment.      */
  CALL GSCOL(2);                /* Set current color to red. */
  and so on.
  CALL GSSCLS;                  /* Close the current segment.*/
  END STARS_AND_STRIPES;
```

- **Viewports can overlap**. The following example uses two square viewports of height and depth equal to the depth of the screen. One is aligned against the left-hand edge of the screen, and the other against the right-hand edge. Because the screen width is less than twice the screen depth, the viewports overlap in the middle.

```
CALL GSQPS(WIDTH,HEIGHT);               /* Ask GDDM what ratio  */
                                        /* the picture space has*/
CALL GSVIEW(0.0,HEIGHT,0.0,HEIGHT); /* Create l-h viewport  */
    /*              .                 */
    /*              .                 */
    /* Draw in left-hand viewport     */
    /*              .                 */
    /*              .                 */
CALL GSVIEW(WIDTH-HEIGHT,WIDTH,0.0,HEIGHT); /* Create right-*/
    /*              .                 */       /* hand viewport*/
    /*              .                 */
    /* Draw in right-hand viewport*/
    /*              .                 */
    /*              .                 */
```

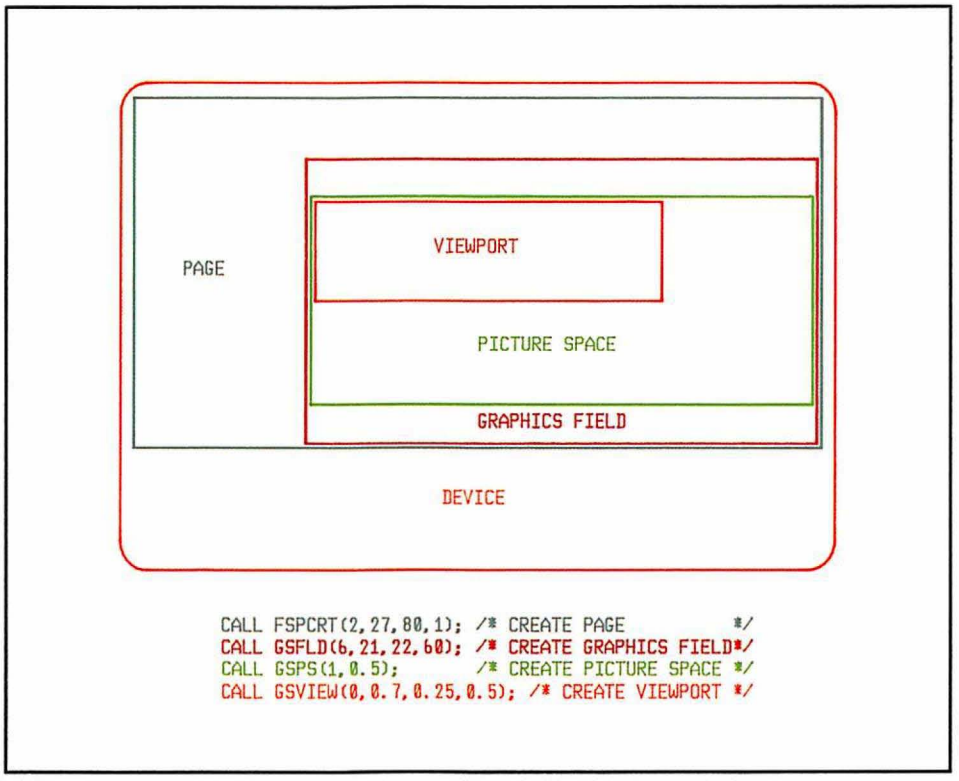


Figure 37. Defining a complete graphics hierarchy (without partitioning)

The graphics window

The objects in the hierarchy considered above were all concerned with a physical subdivision of the display device's screen (or the printer page). The graphics window is completely different. It is the name given to the system of coordinates used to draw graphics on the viewport. It has a different meaning here from the "page window" used in a scrolling context. (The scrolling page window is explained in "Large and small pages" on page 459.)

You may specify a coordinate system with this type of call:

```
CALL GSWIN(0.0,80.0,-30.0,70.0); /*Define user coordinate system*/
```

- The first two parameters state that the x range of the horizontal viewport boundary will run from 0 to 80.
- The second two parameters specify a y range of -30 to 70 for the vertical viewport boundary.

This is the coordinate system that will be used to draw the lines, arcs, and text strings for the graphics output. For example, to draw a line along the bottom of the viewport (or of the screen, if all the graphics objects have been defaulted), you would issue these two calls:

```
CALL GSMOVE(0.0,-30.0);          /* Move to bottom-left corner,  */
CALL GSLINE(80.0,-30.0);         /* draw a line along the bottom.*/
```

If a graphics segment is opened before a graphics window has been specified, GDDM will use the default window. This is equivalent to the statement:

```
CALL GSWIN(0.0,100.0,0.0,100.0);
```

Uniform world coordinates

You might expect these statements always to give 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 in general, they will give a rectangle with one side 10 x-units long and the other 10 y-units long. A square will result only if one x-unit on the screen is physically equal to one y-unit. If this is the case, your program is said to be using **uniform world coordinates**. The commonest symptom of nonuniform coordinates is circles appearing oval.

The simplest way to ensure uniform coordinates is to issue a GSUWIN call:

```
CALL GSUWIN(0.0,80.0,-30.0,70.0);
```

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 will mean that one axis contains slack: if the x range fits the width, the y range will be less than the height, or if the y range fits the height, the x range will be 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);
```

An alternative to GSUWIN is to define a viewport with a width-to-height ratio (that is, an aspect ratio) equal to the ratio of the x range to the y range. If, as is usual, the viewport is allowed to default to 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 );
```

Putting origin of uniform coordinates at bottom left-hand corner

These calls will put the (0,0) position in the bottom left-hand corner of the picture space, which is at the bottom left-hand corner of the screen by default, and give uniform coordinates:

```
DECLARE (HEIGHT,WIDTH) FLOAT DECIMAL(6);

CALL GSQPS(WIDTH,HEIGHT);        /* Query default picture space */
CALL GSWIN(0.0,100*WIDTH,0.0,100*HEIGHT);
```

The longer dimension will have a coordinate range of 0 through 100, and the shorter, 0 through a value less than 100 (or 0 through 100 if the picture space is square). On most displays, the x range will be from 0 through 100, and the y range, 0 through less than 100.

Inverting the graphics window

It is permitted to define a coordinate system in this way:

```
CALL GSWIN(0.0,100.0,100.0,0.0)
```

where the top y value is less than the bottom y value. This call would turn any later graphics upside-down (compared with its appearance under a default graphics window). Graphics text will be inverted only if it is mode-3.

A common error when first using GDDM is to define a graphics window by using:

```
CALL GSWIN(0.0,80.0,32.0,0.0)
```

in an attempt to match the row and column coordinates. (Remember that rows are numbered from the top, whereas y-window coordinates are numbered from bottom to top). The programmer is then rather surprised that his mode-3 graphics appears upside-down! If you want to use mode-3 text with such a graphics window, you must set the y range to −32 to 0, and make all the y coordinates negative.

Similarly, you can interchange the two "x" GSWIN values. This program 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,MODE,COUNT); /* Send picture to screen and wait*/
                              /* for acknowledgement.             */

CALL GSCLR;           /* Clear all graphics.                      */
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 will now be on the right,   */
                      /* aiming crossbow at apple on the left.    */
CALL ASREAD(TYPE,MODE,COUNT);

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 will still be on the left,*/
                      /* but he will be upside-down, aiming       */
                      /* at an upside-down apple!                 */
CALL ASREAD(TYPE,MODE,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 will be upside-down on      */
                      /* the right-hand side of the picture.      */
CALL ASREAD(TYPE,MODE,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;
```

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 your complete picture into one segment, or you can divide it into several if that is more convenient. And you can draw primitives outside segments if you do not require GDDM to retain a record of them after they have been sent to the screen.

Segments have attributes, such as whether they can be moved, and whether they are visible. These can be explicitly set with a GSSATI call, or allowed to default. 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 causes the defaulting of any of the items in the physical hierarchy that have not yet been defined, as does specifying one or more segment attributes with a GSSATI call. These actions also cause the default graphics window to be chosen if an explicit graphics window has not been specified.

Segments have a chapter to themselves: "Chapter 11. Graphics segments" on page 127. 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, it is permitted to redefine the viewport or the graphics window (still under the same higher objects in the hierarchy). This will define 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 specified in a previous environment.

Picture space and graphics field

It is possible to redefine a picture space if it contains no graphics. To clear existing graphics, you can execute a GSCLR call. It is not possible to define a second graphics field without destroying the first graphics field and all its contents.

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 GDDM hierarchy

As an illustration of the GDDM hierarchy at work, here is an example that creates a hierarchy and then redefines the viewport and graphics window:

```
CALL FSPCRT(1,1,20,80); /* Create page (level2), thereby        */
                /* causing the device (level1) to be defaulted  */
                /* to the device invoking the program           */

CALL GSPS(1.0,1.0);
                /* Define picture space (level4), thereby       */
                /* causing the graphics field (level3) to be    */
                /* defaulted to the whole page.                 */

CALL GSSEG(0);  /* Open unnamed segment, thereby causing the    */
                /* viewport (level5) and the window to be       */
                /* defaulted (to 'whole p-space' & 100 by 100)  */
CALL GSMOVE(25.0,70.0);              /* Move to (X=25,Y=70)     */
and so on.      /* Draw first part of picture                   */
CALL GSSCLS;    /* Close first segment                          */


CALL GSWIN(0.0,1000.0,0.0,2000.0);
                           /* Redefine the window. All other    */
                /* entities remain as before. All further       */
                /* graphics will be expressed in terms of the   */
                /* new window coordinates                       */
CALL GSSEG(0);  /* Open another segment                         */
CALL GSMOVE(900.0,1810.0);           /* Move to (X=900,Y=1810) */
and so on.      /* Draw second part of picture                  */
CALL GSSCLS;    /* Close second segment                         */


CALL GSVIEW(0.0,0.5,0.0,1.0);/*Redefine viewport to cover left- */
                /* hand half of picture space. The window of    */
                /* 0-1000, 0-2000 remains in operation          */
CALL GSSEG(0);  /* Open another segment                         */
CALL GSMOVE(730.0,1500.0);           /* Move to (X=730,Y=1500) */
and so on.      /* Draw third part of picture                   */
CALL GSSCLS;    /* Close third segment                          */
CALL ASREAD(TYPE,MOD,COUNT);         /* Send out all three     */
                /* parts of picture that has been constructed   */
```

Creating two pages of graphics

This example shows how you can create two pages of graphics (and alphanumerics, too) and move backward and forward between the two pages:

```
TWOPAGE: PROC OPTIONS(MAIN);

CALL FSINIT;                                    /* Initialize GDDM */
CALL GSSEG(0);          /* Open segment, using default device    */
                        /* and default page (page 0).            */
                        /* The graphics field, the picture-space */
                        /* and the viewport all default to       */
                        /* the whole screen.                     */
                        /* A 100 by 100 window will be used      */
CALL GSLINE(60.0,60.0);             /* Draw one line on page 0 */

CALL ASREAD(TYPE,MODE,COUNT);       /* Send output to page 0   */


CALL FSPCRT(2,0,0,1);  /* Create a second page (page 2). This   */
                       /* action causes the new page to become  */
                       /* automatically selected. All further   */
                       /* graphics and alphanumerics CALLS will */
                       /* 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,MODE,COUNT);  /* Send output from second page  */

CALL FSPSEL(0);        /* Reselect the first page. It is still  */
                       /* 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 GSLINE(80.0,90.0);/* This line is drawn from (60,60) - the */
                       /* current position when the page was    */
                       /* last selected.                        */

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 will be drawn from (0,0).   */
                       /* When a new segment is opened,         */
                       /* the current position                  */
                       /* is always set to (0,0).               */

CALL ASREAD(TYPE,MODE,COUNT);  /* Send output from second page. */
                       /* This output will consist of the two   */
                       /* segments belonging to the page        */
                       /* (the first containing one text string,*/
                       /* the second containing one line).      */
```

```
CALL FSPSEL(0);          /* Reselect first page                    */

CALL ASREAD(TYPE,MODE,COUNT);  /* Send output from first page.  */
                         /* This output will consist of one       */
                         /* segment containing two (joined) lines */
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;
END TWOPAGE;
```

So, these are the points to remember:

- When you select another page for processing (or create another page, thereby causing it to be selected), the first page is left as it was.
- You may leave the current segment open on the first page, ready for further graphics when you return.
- When you reselect the first page, you may add further graphics immediately **if you left the segment open.** To open a new segment would be an error. You cannot open a new segment without previously closing any existing open segment.
- If you want to start a new picture on a page that already has some old 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 typical two-device graphics hierarchy

An example of the hierarchy that a program might address is given in Figure 38 on page 109. The figure shows that the program is communicating with two devices - device 12 and device 27. The first device has two pages, one with a graphics field and two alpha fields, the other with just a graphics field. The second device has only one page, containing a graphics field and three alpha fields. Neither device is partitioned.

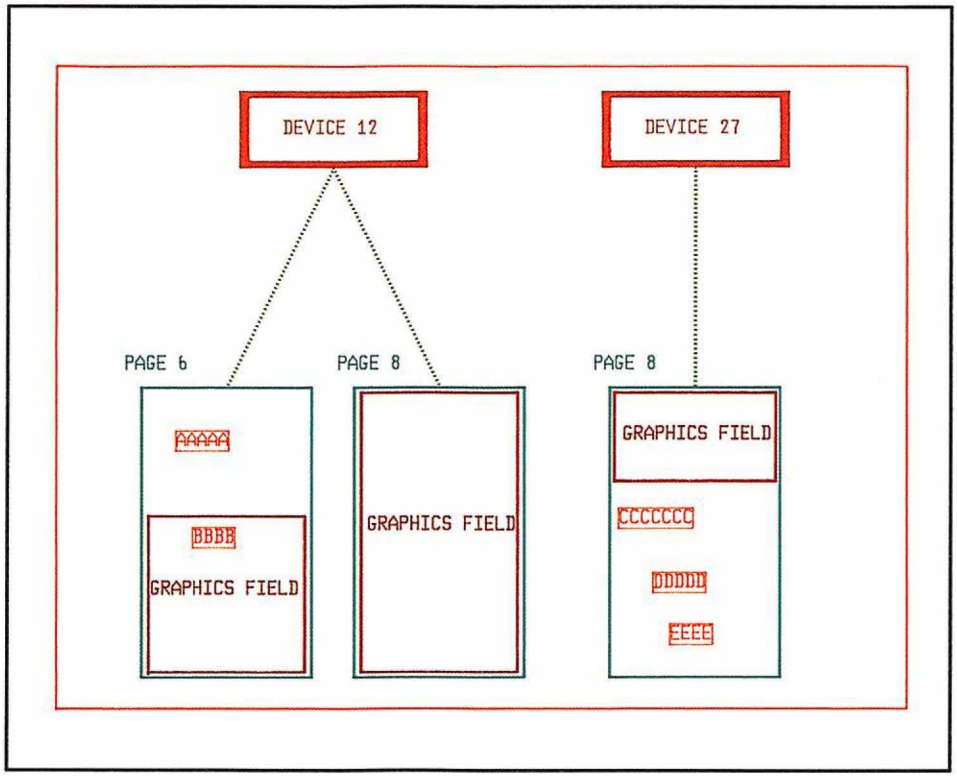


Figure 38. Example of 2-device graphics hierarchy

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, say, the Italian Gothic symbol set (ADMUVGIP) on both the target devices, you have to load it twice - once when device 12 is current and once when device 27 is current.

- Alphanumeric fields and sets of default alphanumeric field attributes apply **per page**. You can use the same identifiers for alpha fields on one page as you do for those on another page.

- One graphics field is allowed **per page**. That graphics field may then contain several graphics segments. Again the same identifiers can be used for the segments on one page as for those on another page.

- 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 14. Interactive graphics" on page 177.) 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, say, the default 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 answer is that the call is valid, and the result depends on whether 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 primitive data to be retained by GDDM. You can use it to restrict the amount of data sent to a device. The data boundary cannot be set while a graphics segment is open.

The format of the call is:

```
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 to the data boundary is controlled by the GSCLP call:

```
CALL GSCLP(1);  /* Enable precise clipping for the current page */
```

Note that clipping is enabled or disabled **per page**. The possible settings of the parameter, and its effects, are as follows:

0 Disables clipping (the default). The data boundary will have no effect. Clipping is initially disabled (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.

1 Enables precise clipping to the data boundary. All primitives inside the boundary will be retained. If a primitive lies across the boundary, only the part of the primitive inside the boundary will be retained. Any primitive that lies completely outside the boundary will not be retained.

2 Enables rough clipping to the data boundary. All primitives inside the boundary will be retained. If a primitive lies across the boundary, the whole primitive, including the part outside the boundary, will usually be retained. In general, any whole primitives that lie completely outside the boundary will not be retained.

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 will be maintained when the segments are manipulated in and around the graphics window. See Figure 39 on page 111 for an illustration of this.

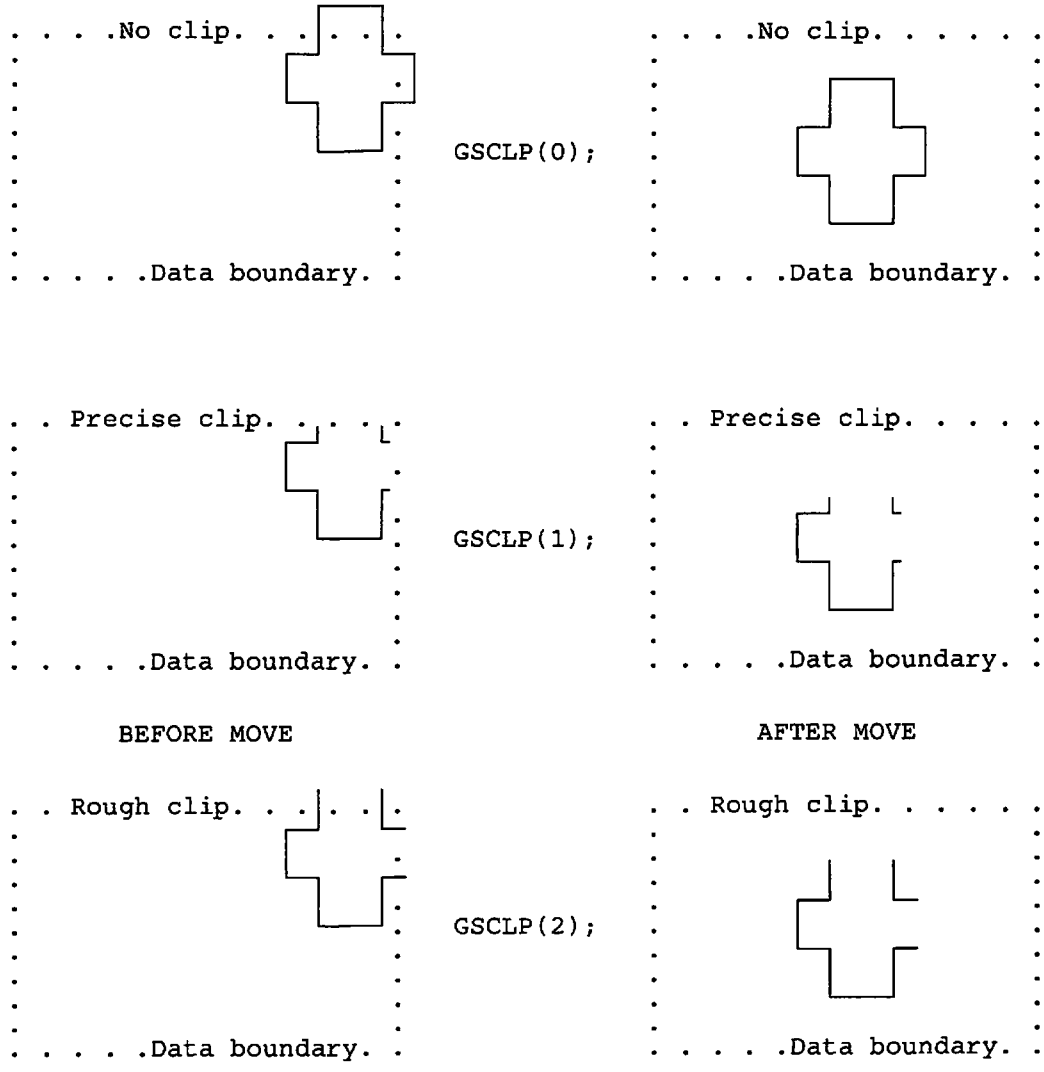


Figure 39. The difference between a precise clip and a rough clip

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.

For details of how graphics text is clipped, see the *GDDM Base Programming Reference* manual.

Segment viewing limits are also specified in world coordinates, but only apply to how much of the current segment, and any segments it calls, is seen at the terminal.

The format of the call is:

```
CALL GSSVL(0.0,30.0,0.0,50.0); /* Set segment viewing limits */
```

The GDDM default segment viewing limits are the graphics field.

Whether clipping to the data boundary is enabled or disabled, primitives are **always** clipped precisely to the specified or defaulted segment viewing limits.

If segment viewing limits have been set and an object is moved around the screen, the object will only be partly or wholly visible when part or all of it lies within the limits. Figure 40 illustrates this effect.

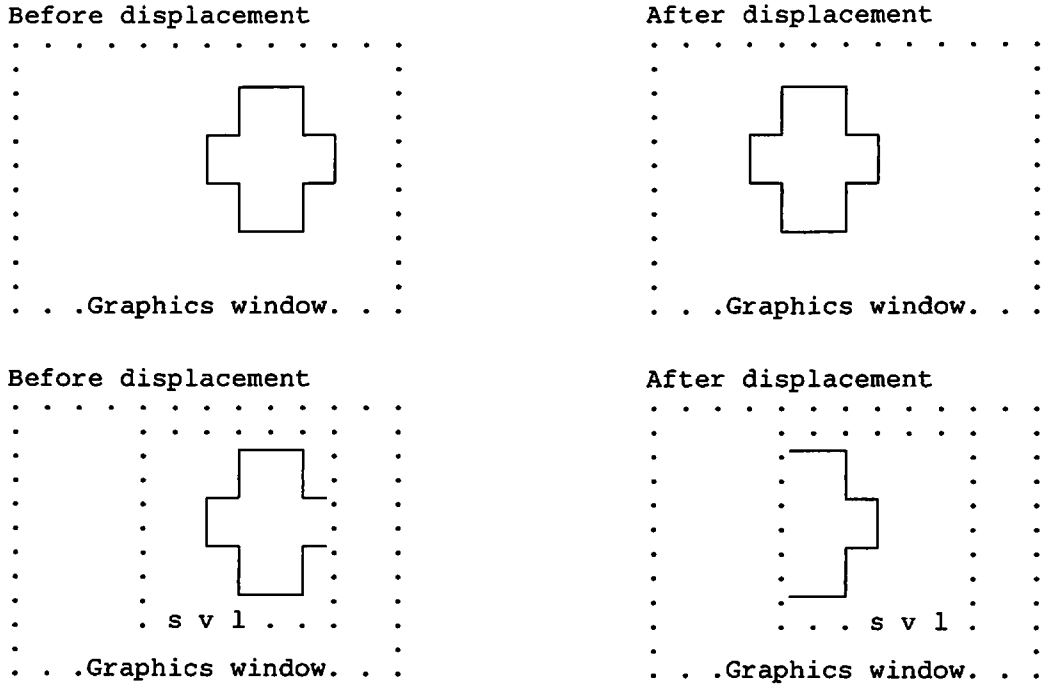


Figure 40. The effect of segment viewing limits on displacement

Sample pan and zoom program using clipping

If you have some GDDM graphics calls that draw something (say a machine-room layout), then it is possible to execute those calls under the influence of any chosen GSWIN call. If clipping has been enabled, just a part of the machine room layout is displayed in the graphics window. In other words, it is possible to zoom in to part of the picture, by setting particular values for the graphics window before executing the GDDM calls that draw the complete picture. The next example program uses this technique.

Other methods of panning and zooming are described in "Panning and zooming" on page 164 and in "Panning and zooming" on page 388.

Figure 41. First output from "Great Britain map" sample program

The program contains a subroutine that draws a map of Great Britain. The terminal user is prompted to specify which part of the map he would like to view.

Output of the complete map is shown in Figure 41. A zoom in to part of the south coast is shown in Figure 42 on page 115.

```
ZOOMMAP: PROC OPTIONS(MAIN);
DCL (TYPE,MOD,CNT) BIN(31); /* Parameters for ASREAD            */
DCL (W1,W2,H1,H2) CHAR(4);  /* Parameters to receive the zoom   */
                            /* area requested by the operator   */
DCL WW1 FLOAT DEC(6) INIT(0);    /* L-hand graphics window param*/
DCL WW2 FLOAT DEC(6) INIT(1000); /* R-hand graphics window param*/
DCL HH1 FLOAT DEC(6) INIT(0);    /* Bottom graphics window param*/
DCL HH2 FLOAT DEC(6) INIT(2000); /* Top graphics window param   */

CALL FSINIT;                /* Initialize GDDM                  */
CALL FSPCRT(93,0,0,1);      /* Whole screen. 32 by 80 on a 3279 */
CALL ASDFLD(1,1,3,1,31,2);  /* Define protected field that will */
                            /* prompt operator for input        */
CALL ASCPUT(1,31,'ENTER THE REQUIRED WIDTH RANGE:');

CALL ASDFLD(2,1,35,1,4,1);  /* Field to receive 1st width parm  */
CALL ASFCOL(2,6);           /* Field color attribute = yellow   */
CALL ASDFLD(3,1,40,1,2,2);
CALL ASCPUT(3,2,'TO');
```

```
CALL ASDFLD(4,1,43,1,4,1);  /* Field to receive 2nd width parm  */
CALL ASFCOL(4,6);           /* Field color attribute = yellow   */
CALL ASDFLD(5,1,48,1,17,2); /* Define protected prompt field    */
CALL ASCPUT(5,17,'AND HEIGHT RANGE:');
CALL ASDFLD(6,1,66,1,4,1);  /* Field to receive 1st height parm */
CALL ASFCOL(6,6);           /* Field color attribute = yellow   */
CALL ASDFLD(7,1,71,1,2,2);
CALL ASCPUT(7,2,'TO');
CALL ASDFLD(8,1,75,1,4,1);  /* Field to receive 2nd height parm */
CALL ASFCOL(8,6);           /* Field color attribute = yellow   */
CALL ASDFLD(9,32,10,1,58,2);/* Protected information field      */
CALL ASCPUT(9,58,
   'FULL WIDTH IS 0 - 1000 MILES; FULL DEPTH IS 0 - 2000 MILES');

/********************/
/* SET CLIPPING ON  */
/********************/
CALL GSCLP(1); /* Set clipping on, as coordinates may be outside*/
               /* graphics window when a partial map is drawn   */

LOOP:;
/************************************************/
/*  NOW REDEFINE THE GRAPHICS FIELD (TO PERMIT  */
/*  THE SETTING OF A NEW PICTURE SPACE). THIS   */
/*  HAS THE DESIRABLE SIDE EFFECT OF DELETING   */
/*  THE PREVIOUSLY DISPLAYED MAP SECTION.       */
/************************************************/
CALL GSFLD( 2,1,30,80);
                  /* Leave out top & bottom rows of the screen.*/

/************************************************/
/*  NOW SET THE PICTURE SPACE (AND THEREFORE THE */
/*  VIEWPORT) TO MATCH THE ASPECT RATIO OF THE  */
/*  PART OF THE MAP THAT IS TO BE DISPLAYED.    */
/************************************************/
IF (WW2-WW1)>(HH2-HH1) THEN
CALL GSPS(1.0,(HH2-HH1)/(WW2-WW1)); /* Define the picture space */
ELSE CALL GSPS((WW2-WW1)/(HH2-HH1),1.0);

/************************************************/
/*  NOW SET THE WINDOW TO THE REQUESTED X RANGE  */
/*  AND Y RANGE. AS THE WINDOW OVERLAYS THE     */
/*  VIEWPORT, AND CLIPPING TAKES PLACE AT THE   */
/*  VIEWPORT BOUNDARY, ONLY REQUESTED PART OF   */
/*  THE MAP WILL BE DISPLAYED ON THE SCREEN.    */
/************************************************/
CALL GSWIN(WW1,WW2,HH1,HH2);

/************************************************/
/*  NOW OPEN A GRAPHICS SEGMENT AND DRAW A      */
/*  THICK RED LINE ROUND THE VIEWPORT BOUNDARY  */
/************************************************/
CALL GSSEG(0);              /* Open a graphics segment          */
CALL GSLW(2);               /* Set line width to thick          */
CALL GSCOL(2);              /* Set current color to red         */
CALL GSMOVE(WW1,HH1);       /* Move to bottom left of viewport  */
CALL GSLINE(WW1,HH2);       /* Draw line to top left of viewport */
CALL GSLINE(WW2,HH2);       /* And so on                        */
CALL GSLINE(WW2,HH1);
CALL GSLINE(WW1,HH1);
```

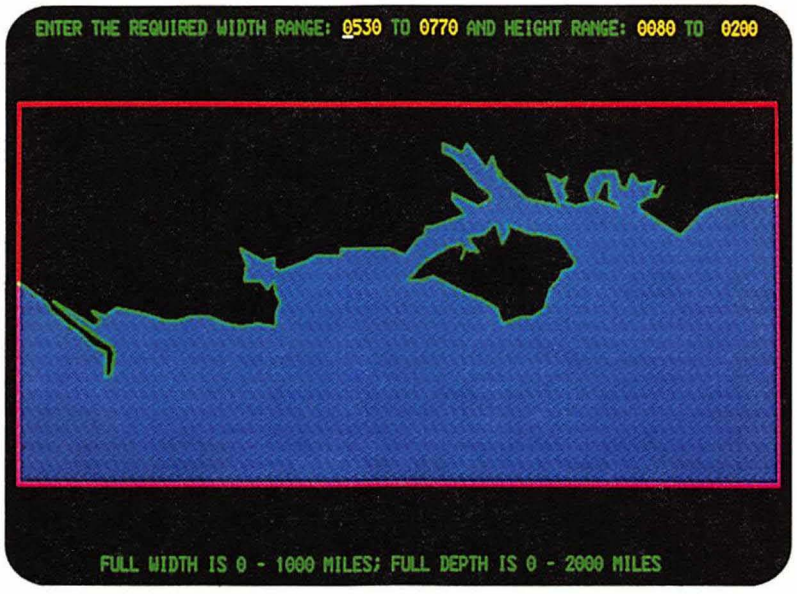


Figure 42. Second output from "Great Britain map" sample program

```
/*********************************************************/
/*  NOW CALL SUBROUTINE THAT DRAWS THE GREAT BRITAIN MAP  */
/*********************************************************/
CALL GBMAP;
CALL ASFCUR(2,1,1);          /* Position cursor in 1st input field*/
CALL ASREAD(TYPE,MOD,CNT); /* Send new map section, await reply */
IF TYPE¬=0 THEN GOTO EXIT; /* Exit if operator presses PF key   */
CALL ASCGET(2,4,W1);         /* User has typed in new requested   */
CALL ASCGET(4,4,W2);         /* ranges, retrieve them from        */
CALL ASCGET(6,4,H1);         /* associated fields                 */
CALL ASCGET(8,4,H2);
WW1=W1;         /* Convert from character(4) to float decimal(6) */
WW2=W2;
HH1=H1;
HH2=H2;
GOTO LOOP;

/*************************************************/
/**** SUBROUTINE TO DRAW MAP OF GREAT BRITAIN ****/
/*************************************************/
GBMAP: PROC;
CALL GSCOL(1);               /* Set color of sea to blue          */
CALL GSPAT(3);               /* Set sea's shading pattern         */
CALL GSAREA(0);              /* Open graphics area (the sea)      */
CALL GSLINE(0.0,2000.0);     /* Draw round the whole-map boundary */
CALL GSLINE(1000.0,2000.0);
CALL GSLINE(1000.0,0.0);
CALL GSLINE(0.0,0.0);
```

```
CALL GSMOVE(X01(1),Y01(1)) /* Move to start of mainland          */
CALL GSPLNE(2117,X01,Y01); /* Call polyline to draw the          */
                           /* coastline of the mainland, using   */
                           /* the 2117 points in the arrays      */
                           /* X01 & Y01. The interior of the     */
                           /* mainland is then not shaded blue   */
CALL GSMOVE(X02(1),Y02(1));/* Move to start of 1st island        */
CALL GSPLNE( 8,X02,Y02);   /* Call polyline to draw 1st island   */
        .                  /*                                    */
        .                  /* ..and so on..                      */
        .                  /*                                    */
CALL GSSCLS;               /* Close the graphics segment         */
END GBMAP;                 /* End of map-drawing subroutine      */

EXIT: CALL FSTERM;         /* Terminate GDDM                     */

%INCLUDE ADMUPINA;         /* Include declarations               */
%INCLUDE ADMUPINF;         /* of GDDM entry points               */
%INCLUDE ADMUPING;
END ZOOMMAP;               /************************************/
```

Chapter 10. Debugging aids

Almost every call made to GDDM may result in an error of some sort. The parameters you pass to GDDM may be incorrect or the resultant processing may meet a problem.

GDDM provides several facilities to aid debugging:

- Error messages and error records
- FSQERR, a call that queries the most recent error message and returns an error record to your program for analysis
- FSEXIT, a call that tells GDDM to pass control to an exit routine whenever an error occurs above a specified threshold
- An external default option that lets you trace the flow of GDDM calls in your program, and output the values of the call parameters to a file.

This chapter gives you some typical examples of how to use the above facilities. For a full description of each, you should read the *GDDM Diagnosis and Problem Determination Guide.*

GDDM error messages

When an error occurs, GDDM typically sends a pair of error messages to the user terminal. Here is an example of such a pair of messages:

```
ADM0055 E DSUSE, AT '4E0202FE'X
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, the error severity code, and the error message text (possibly including inserts). The severity code can be any of the following:

`I` Informational

`W` Warning

`E` Error

`S` Severe error

`U` Unrecoverable error.

A full list and description of the error messages, and suggested programmer responses to them, are given in the *GDDM Messages* manual.

After issuing the error messages, GDDM returns control to the application program and execution continues with the next statement.

Querying the last error record using call FSQERR

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. Informational messages are not counted as errors, so the error will be of warning level or above. Here is the error record layout, and a typical call:

```
DCL 1 ERROR_RECORD,
     2 SEVERITY FIXED BIN(31),       /* Severity code 0|4|8|14|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         */
```

The first parameter specifies the length of the second. This is a variable in which 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. The program might possibly want to maintain a record of errors on auxiliary storage.

More commonly, you may decide to test whether a particular GDDM call (or group of calls) executed successfully. You need request (and declare) only that part of the error record in which you are interested.

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 all critical GDDM calls or groups of calls and invoke your own error-handling routines as required. DSOPEN calls, for instance, are usually critical. The examples in other chapters of this guide do not in general test the return codes because it would tend to 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 ABORT; /* If alpha redefine failed,  */
                                 /* then end run.              */

    Continue normal processing...
```

Specifying error exit and threshold using call FSEXIT

This call specifies a user routine that will gain 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 above. If the reentrant or 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.

A few points you should note:

- If no error exit is explicitly specified (by calling FSEXIT), the default error exit applies. This exit is called following all errors of severity 4 or higher (8 or higher on IMS). It merely presents the error message to the user console and returns control to the program.

- The default error exit can be specified in an FSEXIT call by setting the first parameter to zero.

  To ensure the correct data type for this parameter, the following call should be made 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 will in effect send a trace to the user console 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 */
```

- In PL/I programs, 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.

- User error exits cannot be specified when using COBOL, but you can still use FSEXIT to specify a threshold for invoking the default error exit.

There are more details in the *GDDM Base Programming Reference* manual.

There is an example of an error exit routine in Figure 43.

```
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;                       /****************************/
```

Figure 43. Error exit routine

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;
```

GDDM tracing

To produce a file containing trace output from your program, you have to specify some GDDM external defaults before executing your program.

You do this using the GDDM defaults mechanism, which is similar to the nicknames mechanism described in "Nicknames" on page 378. The defaults can be passed to GDDM in any of the ways described in "How to pass nickname statements to GDDM" on page 384. If an ESSUDS or ESEUDS call is used, it should be executed immediately after the FSINIT.

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'
ADMMDFT CMSTRCE=(ADM00001,ADMTRACE)
```

Under VM/CMS, coding the above statements in your defaults file will produce a print file called ADM00001 ADMTRACE on your A disk. The first statement specifies that the parameter values of every GDDM call in your program are to be output to a trace print file. The second statement gives the filename of the print file. To direct the output direct to the system printer, the second statement would have to be:

```
ADMMDFT CMSTRCE=(,)
```

You may not want to trace every call in a program. For example, you may want only to trace an individual call. The following statements trace the parameter values of a single ASREAD call:

```
ADMMDFT TRCESTR='IF API THEN'
ADMMDFT TRCESTR='   IF (1 GR+4)%%=X''C100000'' THEN PARMSF'
ADMMDFT CMSTRCE=(ADM00001,ADMTRACE)
```

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, C100000 for ASREAD. The RCP codes of all GDDM Base **and** PGF calls, and full information about the defaults mechanism are given in the *GDDM Base Programming Reference* manual.

Or, if you want to trace a call only every nth time that a particular set of conditions occurs, the following 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       '
ADMMDFT CMSTRCE=(ADM00001,ADMTRACE)
```

Format of 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, are given in the *GDDM Diagnosis and Problem Determination Guide.*

Other debugging aids

Returning error information 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. Details are given in the *GDDM Base Programming Reference* manual.

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.

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 Programming Reference* manual.

Part 2. Advanced graphics

Chapter 11. Graphics segments

A segment is a group of graphics primitives that can be handled as an entity, separate from other segments and primitives. This chapter 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 12. Storing graphics" on page 157 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. Well-structured data has similar advantages to well-structured programs, for example, 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 allow the terminal operator 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 allow the terminal operator to position, rescale, and otherwise manipulate each of these items independently.

"Chapter 14. Interactive graphics" on page 177 describes calls and techniques for making a graphics application interactive.

Creating segments

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 "Loading graphics from external storage using call GSLOAD" on page 159.

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 will appear 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 means that you do not intend to add any further primitives to the segment. It does not delete the enclosed graphics.

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 9. Hierarchy of GDDM concepts" on page 89). 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 154.

You can still draw primitives when there is no segment open. The effects are described in "Primitives outside segments" on page 153.

Within a page you may have as many unnamed or named segments as you choose, but no more than one segment with a given 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) will be 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" on page 148.

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, say, 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 44 on page 129 is comprised of only three segments. For identification, all the primitives of each one are the same color.

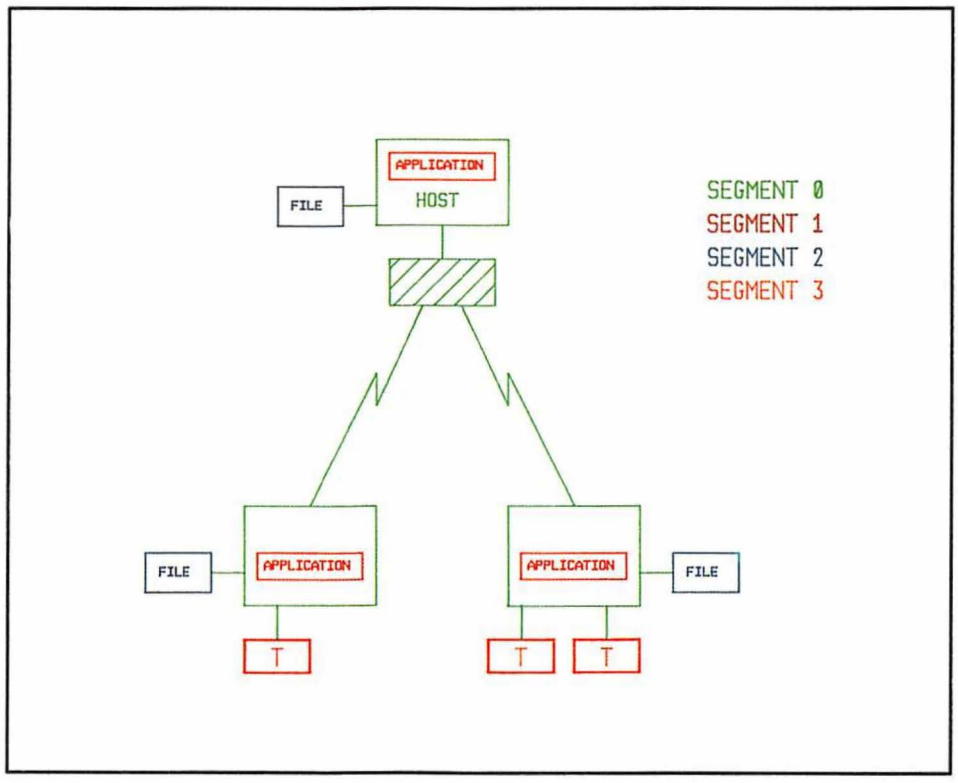


Figure 44. Segments are collections of primitives

Deleting segments

You may delete only a named segment – 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 will cause 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. A default set of segment attributes becomes current initially when a graphics field is defined or created by default. Opening a segment creates a default graphics field if none exists already.

You set the current segment attributes with a 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. The valid types and values are as follows:

1 Detectability. This determines whether a segment can be selected by a pick graphics input device (described in "Chapter 14. Interactive graphics" on page 177). 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 147). 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" on page 148 for more details. The second parameter means:

0 The segment is excluded from the drawing chain. It will 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.

Transforming segments

Segments can be transformed in four ways, as shown in Figure 45 on page 132:

Displaced	Moved to another x,y location
Scaled	Made larger or smaller in the x or y direction, or in both
Rotated	
Sheared	Sloped to one side.

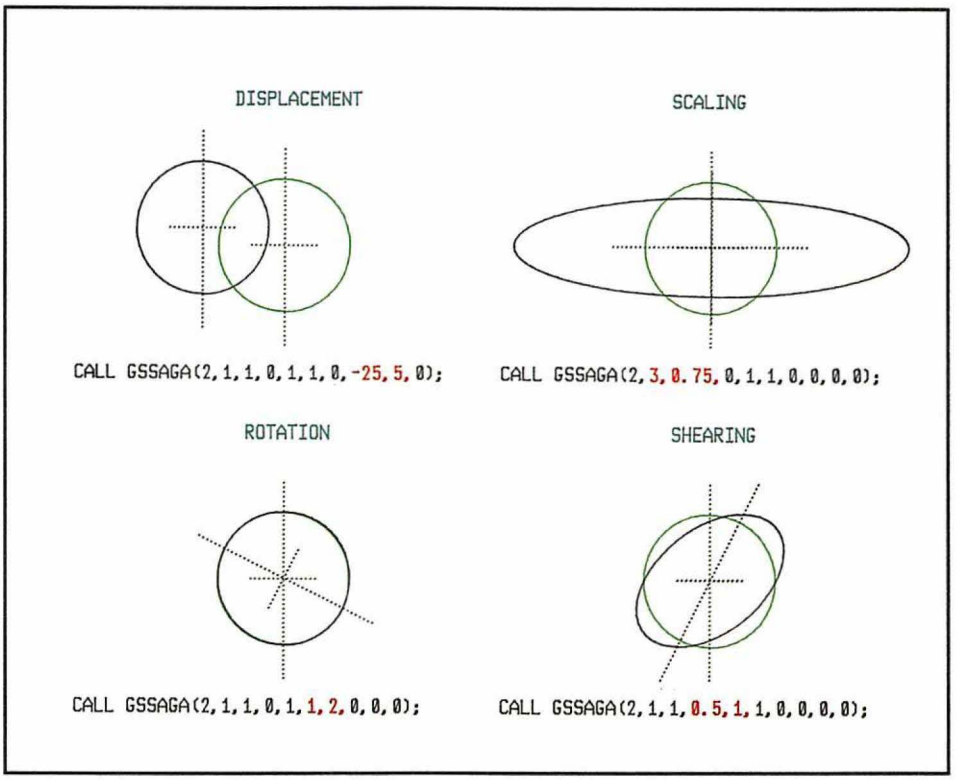


Figure 45. The four segment transformations

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 137). A transformation can also be applied when you call a segment (see "Calling segments from other segments" on page 148).

Typical GSSAGA calls for single transformations are shown in Figure 45. 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 the following 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 46 on page 134 shows the effect of shearing by dx and dy.

The shearing is carried out about the segment origin, the position of which remains unchanged.

If dx and dy have the same sign, the shear is rightward (clockwise), as shown. If they have different signs, the shear is leftward.

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 47 on page 135 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 to 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.

  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 all previous transformations for a segment, allowing it to be displayed as originally drawn, you can execute a new-type call (last parameter 0) with all transformations set to null:

```
   /* Segment-id Scaling  Shearing  Rotation  Displacement Type */
CALL GSSAGA( 7,    1,1,     0,1,      1,0,        0,0,       0);
```

Another call you may use that has the same effect is described in "Transforming segments using call GSSTFM" on page 137.

Notice that GSSAGA does not move the segment origin. All the transformations in Figure 45 on page 132 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" on page 142).

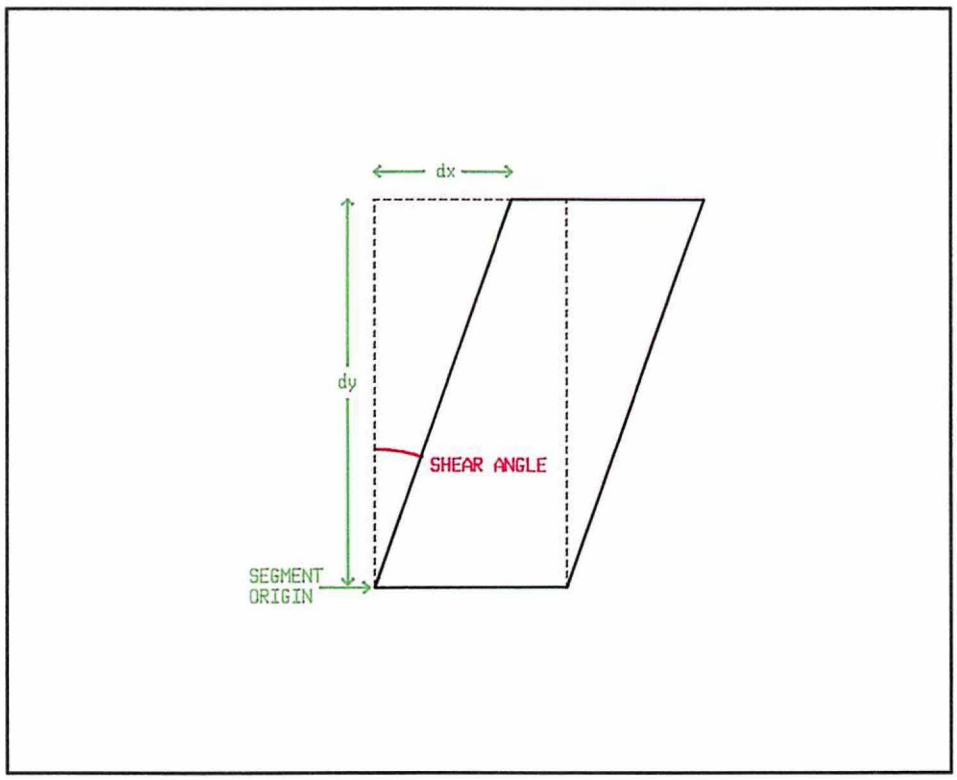


Figure 46. Shearing

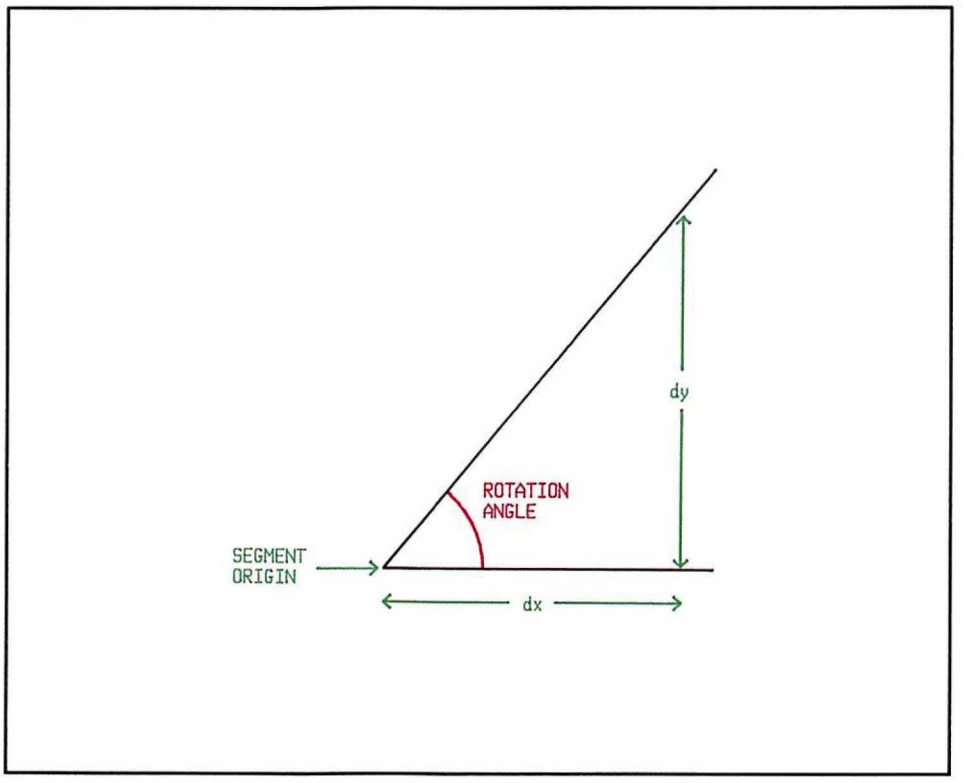


Figure 47. Rotation

How and when transformations take effect

GDDM applies transformations when a page is sent to the terminal (by an ASREAD call, for instance), not when the transformation calls are executed.

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 "Chapter 13. Picture handling in graphics data format" on page 171).

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 7. Basic graphics text" on page 55.

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 will extend from (30,-20) to (45,-5) as shown in Figure 48 on page 137.

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 will move from (35,-15) to (-20,-20). The line will then extend 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" on page 142) 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.

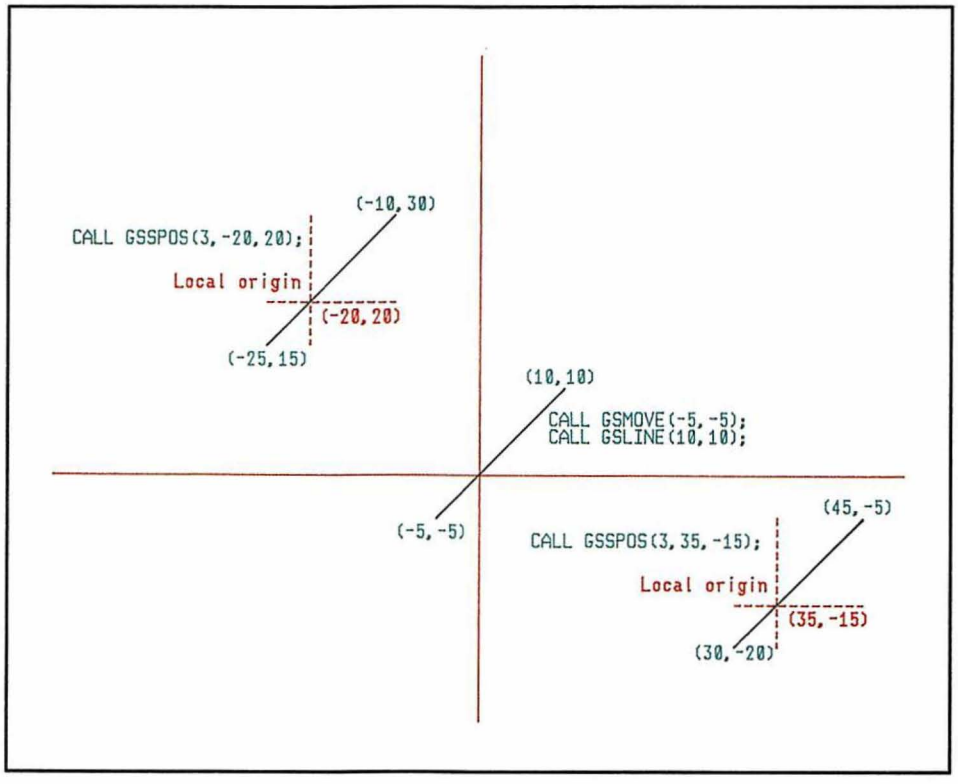


Figure 48. Effects of GSSPOS calls

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:

```
a b c   x
d e f   y
0 0 1   1
```

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(3) = ...;      /*  e  */
MATRIX(6) = ...;      /*  f  */

    /* Segment-id Elements  Values  Type */
CALL GSSTFM(3,        6,    MATRIX,   0);
```

The parameters mean the following:

- 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.

The default values for the third parameter correspond to the identity matrix:

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

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 allows the following simple means of setting the transformations to null, 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 will 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 49 on page 140 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 will ensure 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 pre-scaling is performed.

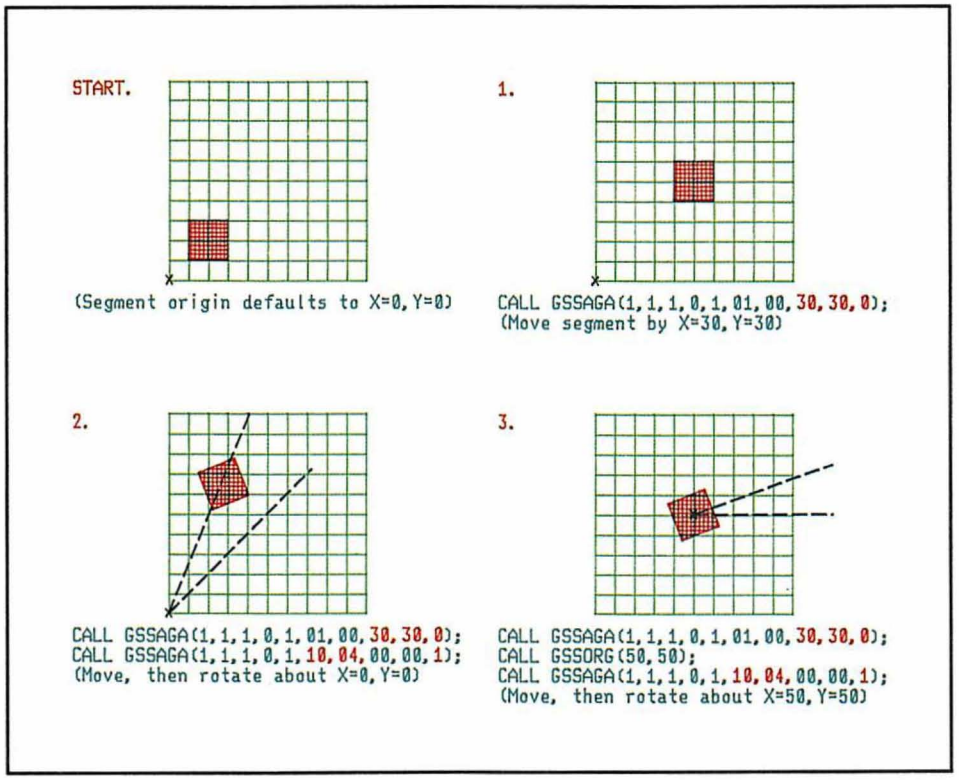


Figure 49 (Part 1 of 2). Results of example transformations

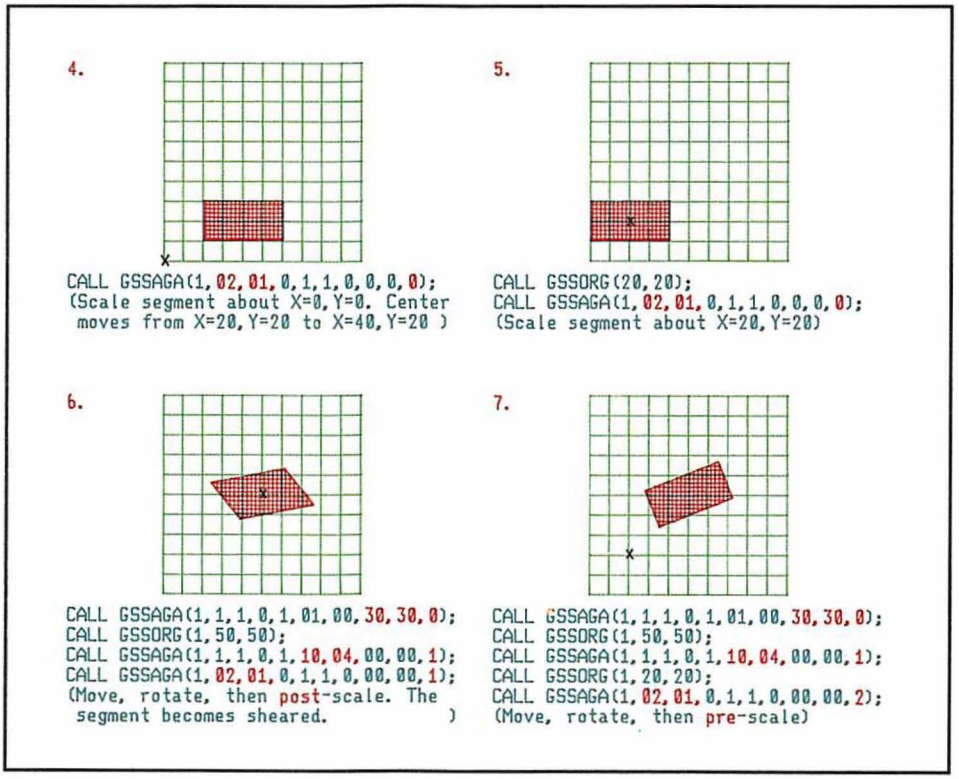


Figure 49 (Part 2 of 2). Results of example transformations

Moving the origin of a segment

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 50. This shows the origin of the world-coordinate system, and the local origin of the segment before and after the GSSORG.

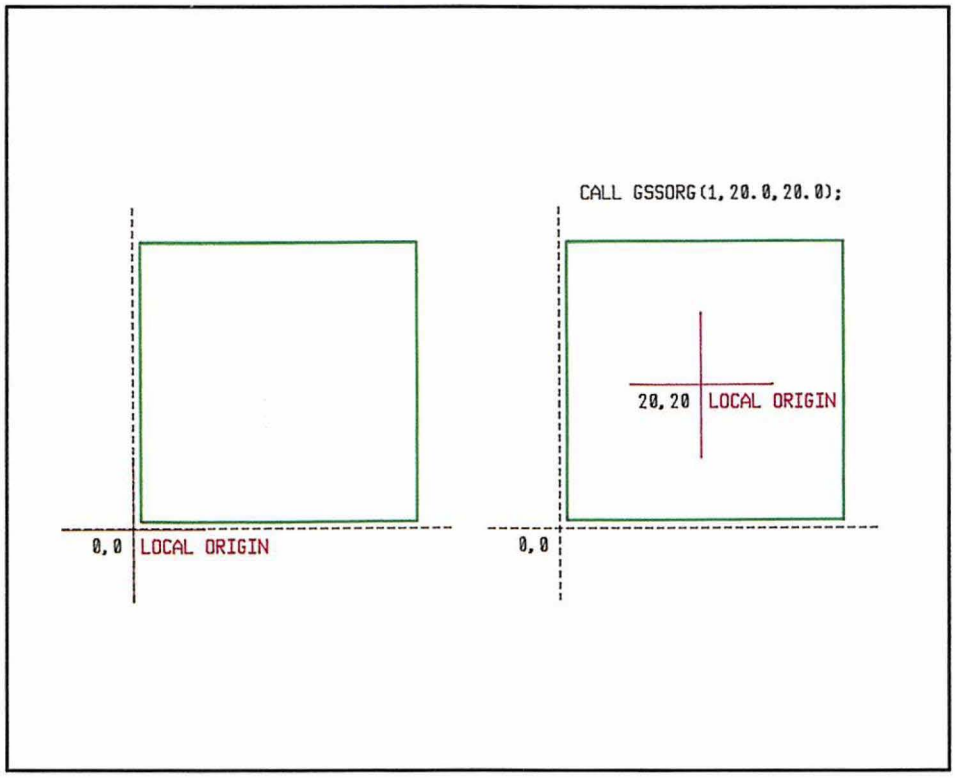


Figure 50. The GSSORG call

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 14. Interactive graphics" on page 177. 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

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 covered in this chapter 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" on page 131. 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 will then be stored when you call GSSCT, and can subsequently be restored using GSPOP. GSAM and GSPOP are covered in "Pushing and popping graphics attributes, using calls GSAM and GSPOP" on page 48.

Copying segments

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. In addition, the following happen:

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.

For example, the effects of the following calls are shown in Figure 51 on page 145:

```
/*****************************************************************/
/*                      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          */
```

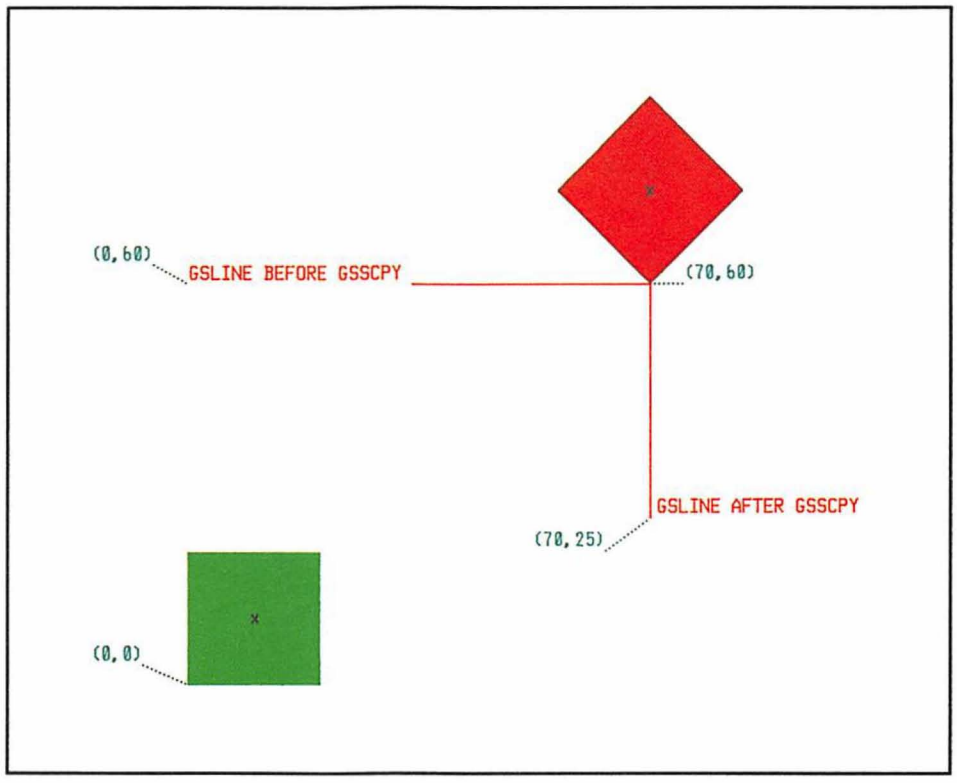


Figure 51. Copying

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

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:

1. The segment is not transformed before it is included.

2. 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).

3. The current position changes to the one that was in effect when the included segment was closed.

4. 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

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.

Segment 0 cannot be copied, so you must not use this 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 chapter, 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 will obscure 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 14. Interactive graphics" on page 177).

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

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

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 will apply.

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.

```
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  */

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 52. Example program using called segments

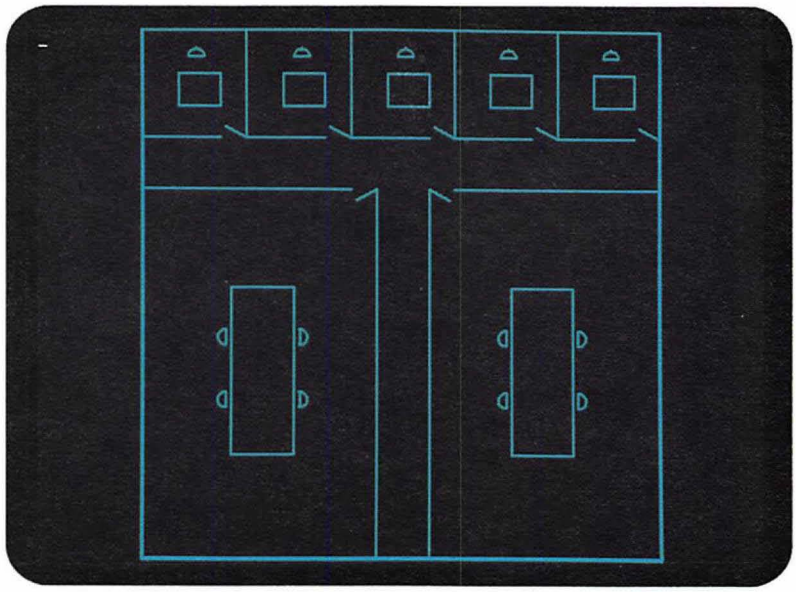


Figure 53. Building plan produced by called segments

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 53, containing standard-size offices and meeting rooms, that themselves contain several standard-size desks, tables, and chairs. You need 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 program that produced Figure 53 is listed in Figure 52 on page 149.

The table and chair segments are created at `/*G*/` and `/*H*/` respectively, and result in segments with their local origins as shown in Figure 54 on page 151.

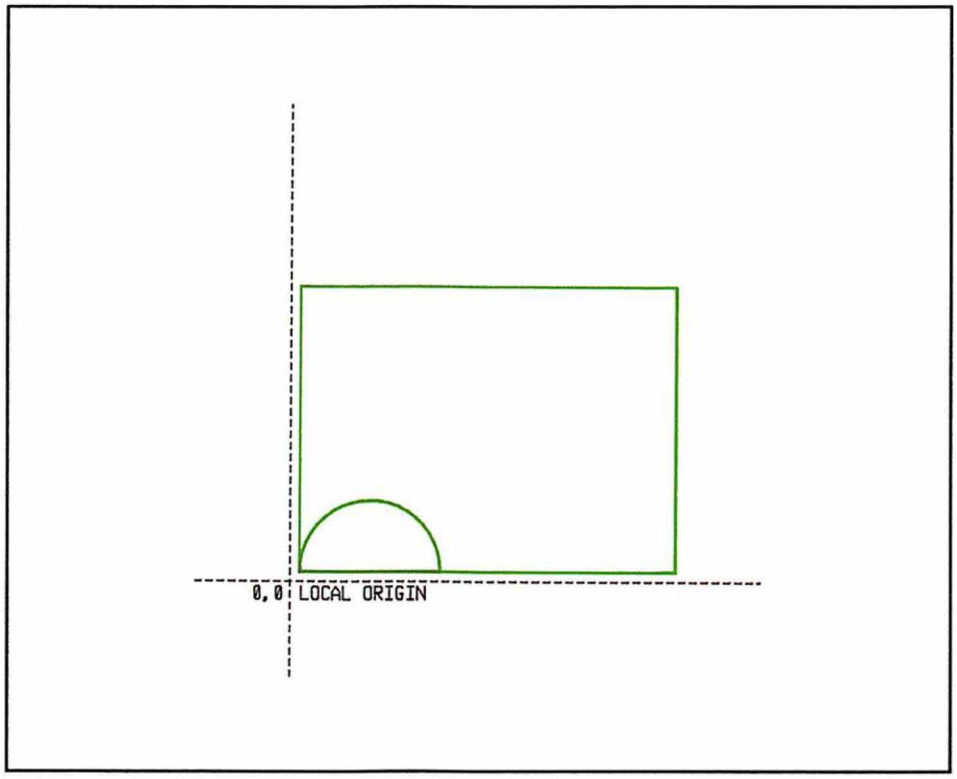


Figure 54. Table and chair segments with origin

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 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, as 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 will 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 49 on page 140. The type parameter on GSCALL controls the way that the the associated transformation combines with a preceding transformation in exactly the same way.

You should bear in mind two points concerning practical applications of called segments:

1. Your called segments could exist on a segment library containing standard items of furniture. Your program would have to restore them before calling them. This may mean you do not know the position of the origin of a restored and called segment.

2. In an interactive graphics program, you can position the called segments with, for example, the graphics cursor. You can also interactively select the point about which scaling and rotation take place. These facilities make experimenting far quicker and simpler, and avoid the need to resort to squared paper and mental arithmetic to calculate transforms.

If you attempt to produce a loop of called and calling segments, GDDM will detect it and issue 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. The following section of example code illustrates the use of GSAM to control the preservation of attributes:

```
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 allows 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. They may be useful in animation applications.

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. Various circumstances under which the screen is completely regenerated are given in the *GDDM Base Programming Reference* manual.

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 will probably need to put all the graphics into segments. Conversely, if you want to exploit the advantages of primitives outside segments, you will 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 14. Interactive graphics" on page 177); they are not saved by the FSSAVE call; they are not printed by a GSCOPY or FSCOPY call (see "Printer as an alternate device" on page 402); 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 will again become 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 will be defaulted.

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).

Chapter 12. Storing graphics

This chapter tells you how to store complete pictures and segments on external storage using the GSSAVE call, and retrieve them using the GSLOAD call.

Saving graphics on external storage using call GSSAVE

This 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');
```

and this one stores segments 7, 8, and 20:

```
DCL SEG_IDS(10) FIXED BIN(31);
DCL DUMMY(1) FIXED BIN(31);
SEG_IDS(1) = 20;
SEG_IDS(2) = 7;
SEG_IDS(3) = 8;
        /* Segments  Filename   Control       Description */
CALL GSSAVE(3,SEG_IDS, 'SEGS3', 0,DUMMY,   16,'GSSAVE EXAMPLE 2');
```

The parameters are as follows:

- The first parameter is the number of elements in the second. If it is zero, GDDM stores all segments in the current page, in priority order.

- The second parameter is an array of segment identifiers. GDDM reads the number of elements specified in the first parameter from this array, and stores the segments that they identify. They are stored on the file in the order in which they are specified. Segment 0 cannot be specified: GDDM stops reading the segment identifiers when it finds an element with a value of 0.

  A 0 in the first element has the same value as a 0 in the first parameter: all segments (including segment 0) are stored.

- The third parameter is the name of the file on which the segments are to be stored. Naming conventions vary according to the subsystem. They are explained in the *GDDM Base Programming Reference* manual. On VM, GDDM creates a file on your A-disk with the specified value as the file name, and a file type of ADMGDF. So the example would create a file called:

  ```
  PIC1 ADMGDF A1
  ```

  GDDM manages the file creation and access entirely when you execute a GSSAVE: there is nothing else you need to do.

- The fourth parameter specifies the number of elements in the fifth.

- The fifth parameter is a one- or two-element array. The first element defines whether GDDM is to be permitted to overwrite an existing file with the same name:

  0 Permit overwriting of an existing file. This is the default.

  1 Do not permit overwriting of an existing file. If such a file exists, GDDM issues an error message and saves nothing.

  The second element defines whether GDDM is to store floating-point or fixed-point coordinate data:

  2 Store two-byte fixed-point data.

  4 Store four-byte floating-point data. This is the default.

  If the fourth parameter is zero, as in the example, GDDM assumes the default values for both elements.

  Floating-point data is recommended. Fixed-point data can cause problems when retrieved. It is less accurate than floating-point, which means retrieved pictures may become distorted, especially if they are enlarged after retrieval. And the fixed-point data may be more severely clipped. This is because, in addition to any clipping that may occur at the time the primitives are drawn, fixed-point GDF is clipped at the boundary of the graphics field at the time of saving, whereas floating-point is not. The advantage of fixed-point data is that the files are usually shorter.

- The last parameter is a character-string descriptor to be stored with the segments, and the second last, the descriptor's length.

There must be no open segment when the GSSAVE call is executed.

It is an error to specify a non-existent segment in the second parameter. 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" on page 148).

GSSAVE is supported whatever the current device, with the restriction that fixed-point data cannot be generated with family-4 devices (see "Chapter 21. Device support" on page 367).

The segments are stored in Graphics Data Format (GDF) (see "Chapter 13. Picture handling in graphics data format" on page 171). In addition to the primitives and their attributes, the file contains 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.

When a GSLOAD 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 for every segment that you add to the library, and a GSLOAD for every one you retrieve.

Loading graphics from external storage using call GSLOAD

This call retrieves all the segments stored in the file called PIC1, and adds them to the current page:

```
DCL DUMMY(1) FIXED BIN(31);
DCL COUNT FIXED BIN(31);
DCL DESC CHARACTER(20);
        /* Filename   Control  No. of segments   Description */
CALL GSLOAD( 'PIC1',   0,DUMMY,     COUNT,          20,DESC);
```

The GSLOAD is equivalent to opening a segment, reexecuting the calls that created the contents of the first of the specified saved segments, closing the opened segment, and so on for the next and later saved segments. There must be no open segment when the GSLOAD is executed.

The parameters are as follows:

- The first parameter is the name of the file on which the required segments are stored. The full name that GDDM searches for depends on the subsystem, as explained in *GDDM Base Programming Reference* manual. On VM, GDDM scans the accessed minidisks in the current search order for a file with the specified name and with a file type of ADMGDF.

- The third parameter is a one-, two-, three-, or four-element array, and the second is a count of the number of elements in the array. A value of 0 in the second parameter, as in the example, causes GDDM to take default actions.

  The first element allows you to specify a series of new identifiers for the segments. GDDM will assign the value you specify to the first segment it loads from the file, and consecutive values to the subsequent segments. This control allows you to avoid conflicts with the identifiers of segments already existing on the page at the time of the load. For any call segment order from a segment within the GDF to a segment within the GDF, the segment identifier is changed to the new identifier given to that segment. A value of 0 tells GDDM to take the default action, which is to use the original, saved, segment identifiers.

  Note that unnamed segments are **not** renumbered.

  The second element specifies what type of transformation, if any, GDDM is to apply to the segment when it is loaded:

  1 No transformation. GDDM preserves the original world-coordinate values of the primitives in the segments. This is the default. Because fixed-point coordinates are device-dependent, this type of load is not recommended for files in fixed-point format.

  2 Make the segments as large as possible, without altering their aspect ratio.

  3 Make the segments the same physical size as when they were saved.

  The three types of load are illustrated in Figure 55 on page 161 and Figure 56 on page 162. Figure 55 shows three segments (a yellow circle, a white square, and a set of blue triangles) as they were saved. Figure 56 shows them after each has been loaded with a different type of GSLOAD call.

All three types of load are explained further in subsequent sections, including their positioning algorithms and their major uses.

The third element specifies what type of action GDDM should take when an object containing drawing defaults definitions is loaded:

1 Ignore drawing default definitions in the saved data.

2 Use drawing default definitions in the saved data as the current defaults, if they have not been previously set. Do not change existing defaults if they are not in the saved data.

3 Use drawing defaults in the saved data as the current defaults, regardless of whether they have been previously set. Do not change existing defaults if they are not in the saved data.

4 Use drawing defaults in the saved data as the current defaults, regardless of whether they have been previously set. Change existing defaults not in the saved data to the standard defaults.

Types 2, 3, and 4 may affect existing primitives, if they were drawn using default attributes.

5 Give the loaded data the same drawing defaults that it had when it was saved. Do not modify the current drawing defaults. This will not affect existing primitives. In the loaded data, segments that are called and chained will 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. This is the default option.

The fourth element specifies the action to be taken when loading a GDF object containing call segment orders to segments that do not exist in the object. The options are:

0 The default, the same as 1.

1 Any call segment orders that cannot be resolved within the GDF object are ignored, and a warning message issued.

2 Any call segment orders that cannot be resolved within the GDF object remain unresolved. It is the responsibility of the application to resolve them.

- The fourth parameter is a variable in which GDDM returns the number of segments it has loaded.

- The last two parameters are a variable in which GDDM returns the descriptor saved with the segments, preceded by the length of descriptor text you require, in bytes. If the actual descriptor is longer, a truncated version will be returned; if shorter, it will be padded with nulls.

In general, segments are loaded in the order in which they were saved. The first segment saved on the file will therefore have the lowest priority (see "Drawing chain and segment priority" on page 147) and the last the highest.

GSLOAD is supported whatever device is current when it is executed, and whatever 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, will be set up with default values if they do not already exist when the GSLOAD call is executed.

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.

It is not possible to query, before a GDF is loaded, what symbol sets it uses.

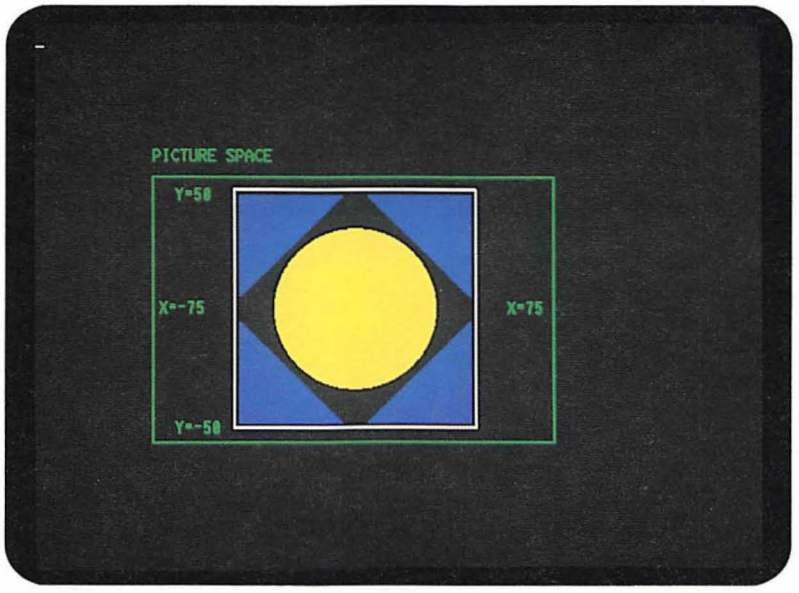


Figure 55. Segments as saved

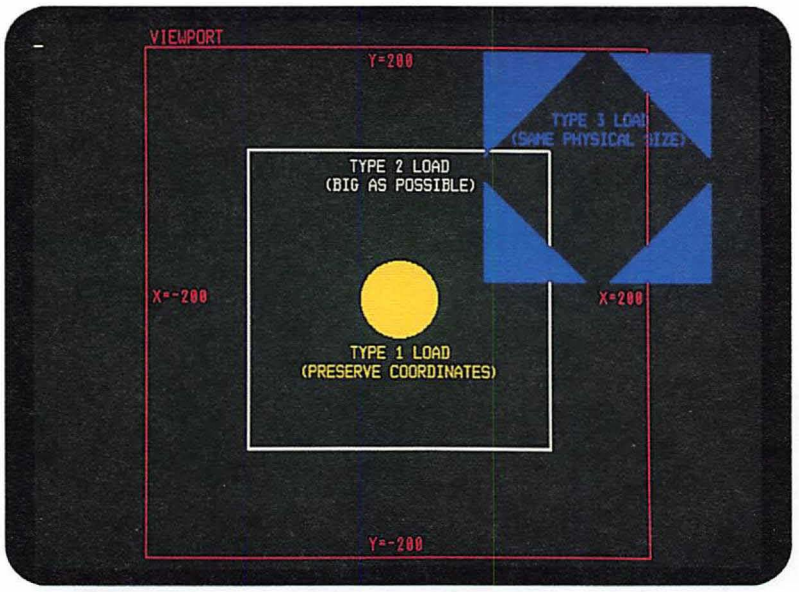


Figure 56. Segments as loaded

Type 1 load

The primitives in the loaded segment retain the same world-coordinate values as when they were saved. If the two graphics windows are the same, there will be no change in the appearance of the segment. If the physical sizes of the world-coordinate units differ, the segment will change size. If the two window origins are in different places, the position of the segment will change.

The world-coordinate units in Figure 56 are physically smaller than those in Figure 55, 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.

The major use of type 1 loads is in building pictures with segments retrieved from segment libraries. Another use is panning (scrolling) and zooming (moving the display window sideways and vertically over the graphics, and changing the physical size at which they are displayed). Both uses are explained further in subsequent sections.

Segment libraries

Many types of application 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 allow 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 will then 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 - the only requirement being that all segments use the same unit.

The picture built up from the segments using type 1 loads should employ the same metric unit as the segments. However, 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.

Even if the segments are symbols, such as flow chart components, rather than drawings of physical objects, a common metric unit is desirable. However, the unit is necessarily an arbitrary one. One solution is to create and store a segment containing a standard grid. This is always retrieved and displayed as the first step in creating a symbol. Then the user draws on top of the grid.

It is usually desirable to ensure that the graphics primitives in a loaded segment fall within the current window, so that they appear in the display. 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, no primitives may appear in the display.

To ensure that at least some of the primitives appear in 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 206).

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 of the segment that falls outside the window will be lost.

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;

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;

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;
```

Notice that the picture is saved only once, and a flag is set to record the action. Repeated saving and loading will distort the picture eventually, because rounding errors in the coordinates will accumulate.

Type 2 load

The segment is loaded and then the coordinates of its primitives are transformed so that the segment is as large as possible. This is illustrated in Figure 57 on page 167, using the white square from Figure 56 on page 162.

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.

The reason that GDDM maps the save-time picture space onto the load-time viewport, rather than mapping viewport to viewport, is that the latter mapping would not work when GSSAVE stores segments from multiple viewports.

Figure 57. 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 21. Device support" on page 367.)

```
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;
```

Type 3 load

After a segment is loaded, GDDM transforms the coordinates of its primitives to keep the size of the picture the same. All primitives will 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 58, using the blue triangles from Figure 56 on page 162. 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 164.

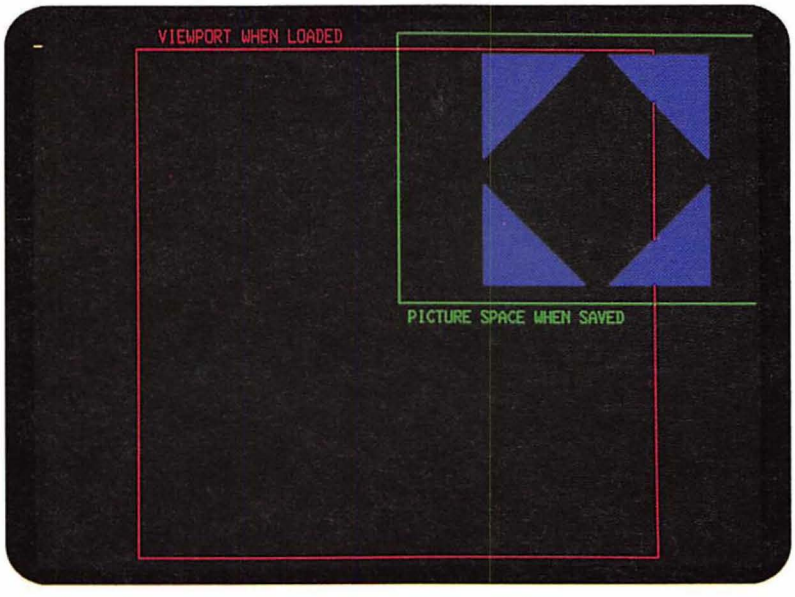


Figure 58. Type 3 load

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 "Type 2 load" will perform in this way if

```
CNTRL(2)=3;
```

is coded in place of

```
CNTRL(2)=2;.
```

Chapter 13. Picture handling in graphics data format

Graphics data format (GDF) is a way of storing pictures. GDDM uses it internally, and also makes it available to application programs. It consists of a set of orders with similar meanings to the GDDM graphics call statements. In many cases there is a one-for-one mapping between GDF orders and call statements.

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 all GDF orders is given in the *GDDM Base Programming Reference* manual.

An application program can access GDF in the following ways:

- By means of GSGET and GSPUT calls.

  GSGET copies GDDM's internal GDF that represents the contents of the current graphics field into a specified program variable. GSPUT does the reverse - adds the GDF contained in a specified variable to the current graphics field.

  More information is given in "GSGET and GSPUT" on page 172.

- By means of ADMGDF-type files generated by GSSAVE calls.

  Normally these are retrieved for display by GSLOAD calls. GSSAVE and GSLOAD are described in "Chapter 12. Storing graphics" on page 157. The file attributes are listed in the *GDDM Base Programming Reference* manual.

- By means of ADMGDF-type files generated by the Interactive Chart Utility (ICU).

  These files define charts created by the ICU. They are created by the ICU using internal GSSAVE calls. The *GDDM-PGF Interactive Chart Utility* introduces the functions that the terminal operator uses to generate them.

- By converting picture interchange format (PIF) files.

These store pictures in 3270-PC/G and /GX work stations, and can be interchanged between a work station and the host computer. PIF orders are similar to GDF.

In a typical case, a PIF file is created at the work station, sent to the host, converted to an ADMGDF-type file and retrieved by a host application program using a GSLOAD. The reverse route can also be followed: a host-created GDF file can be converted to PIF and transmitted to the work station.

Transmitting and converting these files is described in the *GDDM Base Programming Reference* manual. The same manual also describes the differences between GDF and PIF.

GDDM guarantees picture fidelity for converted Base PIF files, as defined in *IBM 3270 PC/G or /GX Supplementary Reference Information for PIF* (Order number GC33-0421).

Inter-Release compatibility

GDF generated by earlier releases of GDDM will be correctly interpreted by the current release. However, GDF 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 will generate. 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 for a particular picture may vary from release to release.

GSGET and GSPUT

GSGET obtains GDF from GDDM, and GSPUT supplies GDF to GDDM, in both cases by means of a program variable. You can write programs to interpret the GDF orders returned by GSGET, or to supply new or updated GDF orders to GDDM by means of GSPUT. 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" above.

- Transferring pictures to and from devices not supported by GDDM, and converting pictures to and from other application programming interfaces.

Retrieval of a picture in GDF form 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. The last GSGET must be followed by a GSGETE.

Here is an example of a GSGETS:

```
DECLARE GETSARRAY(3) FIXED BIN(31);
GETSARRAY(1) = 1;            /* Retrieve segment 1  */
GETSARRAY(2) = 4;            /* Floating-point form */
GETSARRAY(3) = 0;
CALL GSGETS(3,GETSARRAY);
```

The call has two parameters: the second is an array of up to three elements, and the first is the number of elements in the array. The array elements specify:

- The identifier of the segment to be retrieved. A 0 means all segments.
- The form of numerical data in the GDF orders. A 0 or 2 means two-byte integer (fixed-point) form, and 4 means four-byte floating-point form.
- Whether all the GDF held by GDDM is to be returned (including symbol-set names and a prolog to the whole picture), or just segment information and an initial comment order. A 2 means the full information, and a 0 or 1 just the comment order and segment information. The former is generally most suitable when a whole picture is being retrieved (with 0 in the first element of the array); the latter when a single segment has been specified.

GSGETS cannot be executed when there is an open segment. After the GSGETS, the picture cannot be changed until the GSGETE has been executed.

Here is an example of a GSGET:

```
DECLARE GETBUFF CHAR(400) INIT ('');
DECLARE LEN FIXED BINARY(31);
CALL GSGET(400,GETBUFF,LEN);
```

The first parameter specifies the length of the buffer variable into which GDDM is to write the GDF. The second is the buffer variable itself. The third is a variable in which GDDM returns the actual length of the GDF data; it is set to 0 when all the data has been returned. Only complete GDF orders are returned. The program is in error if there is not sufficient space in the buffer variable for the next order. The longest order can be accommodated in 260 bytes.

The GSGETE call is simple:

```
CALL GSGETE;
```

Here is an example of a GSPUT call to pass GDF to GDDM:

```
DECLARE PUTBUFF CHARACTER(400);
CALL GSPUT(4,400,PUTBUFF);
```

The first parameter specifies the type of numeric data being passed: 1 means one-byte fixed-point integer, 2 means two-byte fixed-point integer, and 4 means four-byte floating-point. (One-byte integer data is accepted, although the current release of GDDM does not generate it.) The second parameter is the name of the variable containing the GDF, and the third is its length.

The GDF is added to any that already exists for the current graphics field, whether this was supplied by a GSPUT or created by other GDDM calls. Similarly, more GDF can added after a GSPUT by other GDDM calls, and also by further GSPUTs.

Figure 59 on page 174 shows how to use GSGETS, GSGET, GSGETE, and GSPUT. The program 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 orders, is explained in the *GDDM Base Programming Reference* manual.

```
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                                             */
/*****************************************************************/
/*        .             */
/*        .             */
/*        .             */
/*****************************************************************/
/*   Data capture into GDF buffers                               */
/*****************************************************************/
CALL ASREAD(TYPE,MODE,COUNT);        /* Output the picture       */

/*   DATA CAPTURE START                                          */
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.        */
/*****************************************************************/
/*   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                   */
```

Figure 59 (Part 1 of 2). Handling GDF with GSGET and GSPUT

```
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 2). Handling GDF with GSGET and GSPUT

Device variations

Fixed-point GDF cannot be obtained by means of GSGET when the current device is a 3279, 5080, or a family-4 printer (printer families are explained in "Chapter 22. Using printers" on page 395).

Chapter 14. Interactive graphics

This chapter 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 terminal operator.

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 "Chapter 2. Drawing a simple picture" on page 7.

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 the 3270-PC/G and /GX work stations. This chapter tells you primarily how to write programs for these. Differences on other types of terminal are described in "Device variations" on page 213.

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 PF and ENTER 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 operator. The identifier is called a **tag**. The identifier of the segment to which the picked primitive belongs is also returned. The work station 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 operator 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 operator with the mouse or puck buttons or the stylus tip-switch.

The work stations provide several ways of controlling the graphics cursor: a mouse; a tablet with either a puck (four-button cursor) or stylus; 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; the application cannot, in general, determine which was used.

The operator has a separate alphanumeric cursor for typing into alphanumeric fields. It is positioned using the cursor keys. If the work station has neither a mouse, puck, nor stylus, these keys control the graphics cursor as well. The operator 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 work station provides for the operator. 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 operator has typed in.

After positioning the graphics cursor for pick, locator, or stroke data, or after typing string data, the terminal operator 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 operator may have typed into alphanumeric fields, in addition to graphics data.

Simple interactive graphics program

The routine in Figure 60 on page 179 illustrates graphics input in general, and pick input in particular. It displays a menu of vector symbols and allows the terminal operator to select one of them.

```
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

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 operator 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 will trigger the pick, too.

The MENU routine could be a subroutine of an application in which the operator 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.

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 will assign 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 that the operator 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 a logical input device" on page 192 and succeeding sections.

If the primitive is a string of graphics text, it is picked if any 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 will be picked. More information on priorities is given in "Drawing chain and segment priority" on page 147.

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 130. The GSSATI call at `/*A*/` makes detectable a current segment attribute. Visible is a default attribute.

The symbols used by the example all belong to one segment. It is opened at `/*B*/` and given 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 44 on page 129). 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 (say in the middle of a circle) 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" on page 188.

The example program uses only one type of input device, the pick type. It is enabled by the GSENAB at `/*E*/`, and 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 "The GSREAD call and the input queue" on page 189. 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 Programming Reference* manual 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);
```

The first parameter is set by GDDM 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 9. Hierarchy of GDDM concepts" on page 89). 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 179 to let the operator 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 operator 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 a logical input device" on page 192 and succeeding sections.

Choice input

Choice devices are associated with work-station 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 will cause an interrupt when pressed (provided a string device has not been enabled). The data keys are those that the operator uses for typing letters and

numerals, brackets, currency signs, and so on, including the space bar. On the work-station keyboards, they are colored differently from the other keys.

The data keys also include many of the keys you might normally use for alphanumeric editing, for instance the cursor keys and ERASE EOF.

Choice devices have no echo.

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 Programming Reference* manual). On the 5550, and 3270-PC/G and GX, PA3 cannot be a choice device, as 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 operator 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. If it is a choice device, the third parameter is set by GDDM to indicate the type of key that was pressed. (Further information about all the parameters is given in "The GSREAD call and the input queue" on page 189.) The single parameter of the GSQCHO call returns a fullword integer identifying the particular key. The meanings of this parameter and the third parameter of GSREAD are summarized in Figure 61.

Work station facility	**Parameter values**		
	GSREAD(1,DEV_TYP,DEV_ID)		GSQCHO(NUMBER)
ENTER key	1	0	0
PF key	1	1	Number of key (1-24)
PA key	1	4	Number of key (1-2)
CLEAR key	1	5	0
Data key	1	8	Character code (1-255),
Mouse button or puck button or stylus switch	1	10	Button number (1-2 or 1-3 or 1)

Figure 61. Choice data returned by 3270-PC/G and /GX terminals

The following code shows how GSREAD and GSQCHO can be used. Functions have been defined 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 operator pressed PF1..  */
  THEN CALL ENLARGE;                /* ..perform enlarge function.*/
ELSE IF PFKEY=2                     /* If operator pressed PF2..  */
  THEN CALL REDUCE;                 /* ..perform reduce function. */
ELSE IF PFKEY=3                     /* If operator 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, say.

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 will not return choice data. Similarly, enabling a string device overrides the effects of enabling the data keys as a choice device – they will 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, will 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.

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 operator 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 C1, and hexadecimal 'C1' is 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 Work Station 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 hexadecimal 'F0' through '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 operator. The characters are displayed on the screen in the same way as for ordinary data entry: they are the string device's echo.

The input is queried by a GSQSTR call:

```
DCL STR_IN CHARACTER(25);
DCL CURPOS FIXED BINARY(31);
        /* Length  Input data  Cursor position */
CALL GSQSTR( 25,      STR_IN,       CURPOS );
```

The first parameter of GSQSTR is the length of data requested, and the second is a character variable in which GDDM returns it.

The third parameter is the position of the cursor when the input was triggered. If the operator entered no data (and did not move the cursor), it has the value 1. If one character was entered (and the cursor was not then moved), it has the value 2, and so on. If the field was filled, the cursor remains under the last character. The parameter then returns the same value as if all except the last position was filled. In the case of the example, entering either 24 or 25 characters (and not then moving the cursor) returns a value of 25.

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'); /* Operator 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.            */
```

You can have only one string input area at a time. By default it occupies eight 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" on page 195). 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 terminal operator 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.

Stroke input

Creating stroke input

A stroke device is like a locator, but instead of returning one (x,y) position, it returns a series. The operator 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 at time by positioning the cursor and then pressing a puck or mouse button, or the stylus tip-switch. This is called **polylocator** sampling. The operator'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" on page 196. The default mode is polyline.

Stream sampling can be used to support freehand drawing with a stylus, and digitizing an existing picture by tracing over the outlines with a puck.

Polylines are suitable for drawing pictures that are comprised of straight lines, using a mouse, puck, or stylus.

Polymarkers are useful for allowing the terminal operator to select several items at once – say several primitives or several menu items. Your program can identify the selected items using GSCORR calls (see "Query primitives and segments in specified area using call GSCORR" on page 208).

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 terminal operator 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 operator stops moving the device. The sampling interval varies with the load on work-station 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 operator can draw a set of disconnected lines.

With polylocator sampling, the operator presses the mouse, puck, or stylus switch once for each (x,y) position.

In all cases, the (x,y) positions are stored in a buffer at the work station. The number of (x,y) positions that can be held in the buffer depends on two things: the storage available at the work station and a program-defined maximum. The

operator cannot store any more (x,y) values when the lesser of these two limits is reached. Your program can define its maximum in the GSISTK call (see "Initializing a stroke device" on page 196). The default is 64. If the storage at the work station is less than the program-defined maximum, a warning message is issued at the terminal.

When the limit is reached, the X marker reappears at the last recorded point and the alarm sounds. The operator must then trigger the device with the ENTER or a PF key, or, if the data keys are enabled as a choice device, with a data key. This causes the contents of the buffer to be sent to your program.

The operator can, instead, choose to trigger input to the program by pressing the ENTER or a PF key before the buffer is full. With stream sampling, the device must be deactivated using a mouse or puck button or stylus switch before the trigger (ENTER, PF, or data key) is pressed. 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.

In the first parameter, you must specify the maximum number of values that your program can accept. Typically, this equals the size of the arrays. This, in turn, typically equals the work-station buffer maximum – either the explicit maximum specified in a GSISTK call or the default of 64.

GDDM sets NUM to the actual number returned by the work station. This is always equal to or less than the buffer maximum. If the actual number returned in NUM is greater than the number specified in the first parameter, the excess x and y values are discarded. If it is smaller, any excess array positions are left unchanged.

GDDM sets each element of the second parameter, DFLAGS, to indicate the operator action that generated the corresponding x and y values, as follows:

- With polylocator sampling:

  1, 2, or 3 if mouse or puck was used. The number indicates which button was pressed.

  1 if the stylus tip-switch was used.

  -1 if no values have been returned in the corresponding x- and y-value arrays (XARRAY and YARRAY in the example). GDDM sets the trailing elements to this value when the actual number of points returned (NUM in the example) is less than the specified maximum (200 in the example).

- With stream sampling:

1 when the operator pressed a mouse or puck button or the stylus tip switch to activate the device and initiate sampling. In other words, 1 means a new line was started.

0 when the corresponding x and y values represent a point sampled from the trajectory of a moving cursor - a point within or at the end of a line.

-1 when no corresponding x and y values have been returned (as for polyline and polymarker modes).

The first position in the arrays is not necessarily the same as the initial position of the cursor. The work station starts recording (x,y) data when the operator activates the device using a mouse or puck button or stylus tip-switch. The operator can move the cursor from its initial position before doing this.

Simple polyline program

The program in Figure 62 uses a stroke device of the default type, namely polyline.

After reading the stroke input, the program redraws the line created by the operator. 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. The example redraws the line in red.

A second GSREAD sends the redrawn line to the work station. Before this call is executed, stroke input is disabled, and the ENTER key enabled as a choice device. When the operator presses ENTER 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 62. Program using polylocator stroke device

Enabling or disabling a logical input device

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 "The GSREAD call and the input queue" on page 189.

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 operator triggers a transmission in one of the ways described in "Choice devices as triggers" on page 183. 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 operator 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 pairs that trace the path of the cursor as it was moved by the operator.

- If the first parameter did not specify a choice device, the second parameter must be set to:

  1 The only permitted value for non-choice devices.

- If the first parameter did specify a choice device, this 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 197. You can query whether a logical input device is enabled, as explained in "Querying a logical input device" on page 198.

The GSREAD call and the input queue

A single action by the terminal operator 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 operator.
- 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 operator may have typed in.

A typical GSREAD call is:

```
      /* Delay  Device_type  Device_id */
CALL GSREAD(1,     D_TYPE,      D_ID);
```

The first parameter is normally set to 1, meaning that GDDM is to perform an I/O operation if the queue is empty. By setting it to 0, you can examine the input queue without doing any I/O with the terminal. GSREAD will simply examine the queue, and if it is not empty, remove the topmost record and make it available. If

the queue is empty, GDDM sets the second and third parameters to zero, and returns control to your program.

The second parameter is set by GDDM to indicate the type of logical input device that generated the record currently being made available. 1 means a choice device, 2 a locator, 3 a pick, 4 a string device, and 5 a stroke device.

The third parameter is of interest only for choice devices, where it identifies the type of key that was pressed. Its possible values are listed in Figure 61 on page 182. For input from other devices, it is always set to 1.

The codes returned in the second and third parameters are the same as the ones you set in the first two parameters of GSENAB.

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 will 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."

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 operator causes an interrupt that does not create an input record, GDDM will reject it. No input record is created if, for instance, the operator 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 will not perform an I/O operation and 0 means that it will (unless the first parameter of the GSREAD is 0, in which case it will never perform 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 the next example, before attempting to process any of the data. Furthermore, this technique allows you to be sure whether the next GSREAD will update the screen. As already mentioned, GSREAD will 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 */

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 operator 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*/
/*****************************************************************/
```

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 a logical input device

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.
- One or two more optional calls, namely GSIDVF (initial data value, float) and GSIDVI (initial data value, integer). The purpose of these calls is 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 Programming Reference* manual.

Initializing a locator device

Specifying locator echo type and initial position using call GSILOC

The 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 will jump 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 is a code specifying the echo-type. The permissible values and their meanings are:

0 Default. The same as 2.

1 Null (invisible) echo.

2 The terminal's graphics cursor.

3 Small tracking cross.

4 Rubber band. This means a line with one end fixed and the other at the current locator position. Its length and orientation change as the operator moves the locator device (mouse, puck, stylus, or cursor keys). You can specify the position of the fixed end using the GSIDVF call, which is described in "Initializing a rubber-band locator." That section also describes the initial appearance of the line.

5 Rubber box. This means a box made of two lines parallel to the x axis and two parallel to the y axis, with one corner fixed and the other at the current locator position. Its width and depth change as the operator moves the locator device. You can specify the position of the fixed corner using the GSIDVF call. "Initializing a rubber-box locator" on page 194 tells you how to use this call, and describes the initial appearance of the box. An example of using a rubber box is given in Figure 67 on page 210.

6 Segment. This makes the locator device drag a segment around the screen. You specify the segment in a GSIDVI call, as described in "Initializing a segment locator" on page 194. By default, the segment will be positioned so that its origin is at the current locator position. For instance, if the segment is a rectangle that was drawn with its bottom left-hand corner at (0,0), then, if the origin has not been moved with a GSSORG call, this corner indicates the position of the locator on the screen. "Initializing a segment locator" gives further information, and explains how to use the GSIDVF call to make a different point in the segment indicate the locator position.

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);
```

In both these GSIDVF calls, the first parameter is the device-type (2 means that they refer to a locator device). The second parameter must always be 1. The third parameter indicates whether the x or y coordinate is being specified: 1 means x and 2 means y. The last parameter is the value of the coordinate.

If you omit one of the GSIDVF calls, GDDM will use a default value, namely, the value of the corresponding coordinate of the initial locator position − 20.0 or 30.0 in this case. The line will then be either vertical or horizontal when it first appears on the screen. If you omit both calls, the fixed end of the line will be at the locator's initial position, (20,30). Because the fixed and movable ends then coincide, the line initially appears as a point.

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);
```

The parameters have similar meanings to when they are used for a rubber-band echo.

If you omit one of the GSIDVF calls, GDDM will use a default value, namely, the value of the corresponding coordinate of the initial locator position − 20.0 or 30.0 in this case. The box will then be either a vertical or horizontal line when it first appears on the screen. If you omit both calls, the fixed corner of the box will be at the locator's initial position, (20,30). Because the fixed and movable corners then coincide, the box initially appears as a point.

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);
```

The first parameter must be 2 for a locator device. The second parameter is always 1. The value 1 in the third parameter specifies that the segment identified by the fourth parameter is to be the locator echo.

By default, the origin of the segment coincides with the locator position. For instance, if the segment is a circle drawn with its center at (50,-20), the center will be +50 x units and -20 y units from the locator position. If you want a different spatial relationship, you can specify x and y offsets using GSIDVF calls. In this instance, to make the center of the circle coincide with the locator, the segment must be offset from its default position relative to the locator by -50 x units and +20 y units. You would specify this as follows:

```
CALL GSIDVF(2,1,1,-50);
CALL GSIDVF(2,1,2,20);
```

In both these GSIDVF calls, the first parameter, 2, means that they refer to the locator device. The second parameter must always be 1. The third parameter indicates whether the x or y offset is being specified: 1 means x and 2 means y. And the fourth parameter is the offset.

An alternative method is to move the segment origin to the center of the circle with a GSSORG call (see "Local origin when dragging a segment" on page 204).

Initializing a pick device

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 using call GSIPIK

You can specify a primitive over which GDDM is to initially position the pick as follows:

```
CALL GSIPIK(1,0,SEGID,TAG);
```

The first parameter must be 1 and the second zero. The third and fourth parameters specify the segment to which the primitive belongs and its tag. The segment should be visible and detectable (see "Segment attributes" on page 130).

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

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');
```

The parameters are as follows:

- The first must always be 1.
- The second specifies whether the characters typed by the operator are to be echoed on the screen. A 1 means they are. A 2 means they are not – there is to be no visible indication of what the operator types. The 2 value is intended for confidential input, such as passwords.
- The next two parameters specify the position of the input area in world coordinates.
- The fifth parameter specifies the number of characters in the sixth.

- The last parameter is the text that is to be displayed initially.

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 operator.

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

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 will jump 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.

The last parameter specifies the maximum number of points that can be returned by a single GSREAD call.

The possible echo-type values in the second parameter are:

0 The same as 1. This is the default.
1 The echo is to be a line joining the sampled points.
2 The echo is to be a marker at each sampled point.

The possible sampling-method values in the third parameter are:

0 The same as 1. This is the default.
1 The cursor position is to be sampled once each time the operator activates the device. This means a polylocator mode – whether polyline or polymarker depends on the value of the second parameter.
2 Sampling is to be continuous while the device is active (stream mode).

All combinations are valid, except for an echo-type of 2 (marker) with a sampling method of 2 (stream mode).

Here are examples of how to specify polyline, polymarker, and stream modes:

```
          /*  Polyline mode  Initial position  Number of points */
CALL GSISTK(1,      1,1,        50.0,50.0,          50 );

          /* Polymarker mode Initial position  Number of points */
CALL GSISTK(1,      2,1,        10.0,90.0,          50 );

          /*  Stream mode    Initial position  Number of points */
CALL GSISTK(1,      1,2,         0.0,0.0,          100 );
```

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 will be superimposed on the locator echo, and the locator echo will show 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 will 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, ENTER, 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 operator moved the locator from the final stroke position before pressing the trigger key. The pick data will comprise 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" on page 96). 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 will have 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 will be displayed in its specified or default initial position, whereas the application may be easier to use if the echo remains where the operator put it.

Querying a logical input device

You can query a logical device using the GSQLID call. GDDM will indicate 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" on page 188). 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 179 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 terminal operator to pick segments rather than primitives. The program in Figure 63 on page 200 illustrates this. It draws several squares using GSLINE calls, each square being a separate segment. The operator can then select and delete any square.

To ensure that the squares really are square, the program executes a GSUWIN call at `/*A*/`. To allow them to be picked, the detectable attribute is set on at `/*B*/`. The squares are drawn by the subroutine DRAW_SQUARE, called at `/*C*/`. 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 operator 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 operator 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 will be   */
    CALL DRAW_SQUARE;                    /* in 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 will      */
                                         /* happen when a key       */
                                         /* is pressed but the      */
                                         /* pick is not over any    */
                                         /* line in a square.       */
  CALL GSQPIK(SEL,TAG);                  /* Get the segment id and  */
                                         /* tag. SEL will be 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      */
                                         /* will have 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. Segment picking example

Simple free-hand drawing program

The program in Figure 64 enables the terminal operator to draw on the screen. It will capture original drawings, for which the operator would typically use a tablet and stylus. It can equally well be used for digitizing existing drawings, which the operator 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 allows the terminal operator to record the maximum number of points that GDDM allows, namely 1024. If the operator 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 operator. 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 operator then presses ENTER, 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 operator by allowing 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 on page 203 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, `/*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 work station draws echoes" on page 203.

The GSREAD, `/*E*/`, displays the square onto the screen. The operator 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 will probably be 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 204.

```
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*/
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 (say,     *//*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 will move as      */
                                        /* cursor is moved.         */
                                        /* When trigger key is      */
                                        /* pressed, control will    */
                                        /* return 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. 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 work station draws echoes

Many echoes are drawn by the work station 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.

Although the echo conceptually overlies the rest of the picture, it does not overpaint or underpaint the primitives that conceptually underlie it. Instead, the echo color is combined with the underlying color using an exclusive-OR operation. Where the two are the same color, the result is invisible.

A noticeable effect of this algorithm is that if a segment echo is drawn on top of the original segment, both the original and the echo are invisible. However, as soon as the echo is moved slightly by the operator, both become visible.

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 will not inherit the invisible attribute. There will then only be one copy of the segment on the screen - the echo. Changing a segment's visibility attribute from visible to invisible causes the whole 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 amended code from the example:

```
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.   */
```

The following changes have been made to the way the square is drawn:

1. The statements /*J*/ to /*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 66.

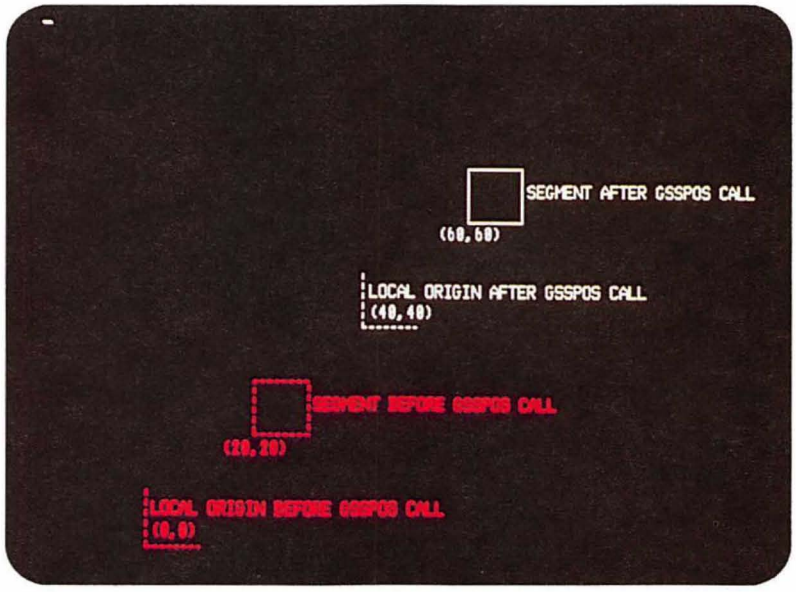


Figure 66. Local origin of echo segment

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 terminal operator to select a reference point before dragging or transforming a segment. The statements `/*M*/` to `/*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  */

/*                                 .                                 */
/*                                 .                                 */
/*                                 .                                 */
```

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 204.

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 following 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 "Sample pan and zoom program using clipping" on page 112.
- 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 164.
- Allowing the operator to use the user-control facility for panning and zooming – see "Panning and zooming" on page 388.

Retained and non-retained modes

The work station has two modes of operation: retained, in which the graphics orders that define the segments are stored by the work station; and non-retained, in which they are not. Retained is the normal mode of operation, but if there is insufficient segment storage available at the work station, non-retained mode must be used. This can be selected either by GDDM or by you, using a processing option. More information is given in "Retained and non-retained modes" on page 508.

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 terminal operator. 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 are as follows:

- The first indicates the type of correlation to be performed: 0 means correlate only visible and detectable segments. 1 means correlate segments whatever their attributes.
- The next two specify the position in world coordinates of the center of the correlation area.
- The fourth indicates how the size of the area is being specified: 1 means in terms of the default pick aperture (see "Initializing a pick device" on page 195). 2 means in world-coordinate units.
- The next two parameters specify the size of the area, in terms defined by the preceding parameter. The second one of the two is an array, and the first is the number of parameter elements it contains.

  If the value of the fourth parameter is 1, as in the example, a single-element array is required, containing a scaling factor to be applied to the default pick aperture.

  If the value of the fourth parameter is 2, a two-element array is required, containing the width and depth in world-coordinate units.
- The next parameter specifies the maximum number of hits (that is, unique primitive/segment identifier pairs) to be returned, and the two succeeding parameters are arrays to contain the returned identifiers and tags. For each tag returned in the tag array, the identifier of the segment that the primitive is in will appear in the corresponding element of the segment array. Therefore, where there are several tag numbers in a segment, the identifier of that segment will appear several times in the segment array.
- The last parameter is a variable in which GDDM returns the number of hits.

Primitives with a tag of zero, primitives outside segments, and primitives in segment 0 are ignored - they can never be hits. The tags and segment identifiers are returned in priority order, from highest to lowest (see "Drawing chain and segment priority" on page 147). Two or more primitives with the same tag within a segment count as a single hit; only the first instance is returned.

Correlation with GSCORR differs from selection with a pick device in several ways:

- GSCORR does not require action by the terminal operator. 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 67 on page 210 shows how to use GSCORR in an interactive context.

It displays an array of crosses. The terminal operator 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*/

  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 67. 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" on page 148. You will need to understand that call before you use GSCORS.

GSCORS is a more sophisticated version of GSCORR. Your program specifies a rectangular area, 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);
```

The parameters are as follows:

- The first indicates the type of correlation to be performed: 0 means correlate only visible and detectable segments, and 1 means correlate segments whatever their attributes.
- The next two parameters specify the position in world coordinates of the center of the correlation area.
- The fourth indicates how the size of the area is being specified: 1 means in terms of the default pick aperture (see "Initializing a pick device" on page 195), and 2 means in world-coordinate units.
- The next two parameters specify the size of the area, in one of the terms defined by the preceding parameter. The second one of the two is an array, and the first is the number of parameter elements it contains.

  If the value of the fourth parameter is 1, as in the example, a single-element array is required, containing a scaling factor to be applied to the default pick aperture.

  If the value of the fourth parameter is 2, a two-element array is required, containing the width and depth in world-coordinate units.
- The next parameter specifies the maximum number of hits (that is, unique primitive/segment identifier pairs) to be returned, and therefore has the same value as one of the dimensions of the segment and tag arrays.
- The eighth parameter, depth, has the same value as the other dimension of the segment and tag arrays. It specifies the extent to which GDDM traces the segment structure, starting with and including the correlated segments.

- The two succeeding parameters are arrays to contain the returned segment identifiers and tags. For each tag returned in the tag array, the identifier of the segment that the primitive is in will appear in the corresponding element of the segment array. Therefore, where there are several tag numbers in a segment, the identifier of the segment will appear several times in the segment array.

- The last parameter is a variable in which GDDM returns the number of hits.

As an example of a use of GSCORS, let's imagine, as we did in "Calling segments from other segments" on page 148, a structure consisting of a building segment, containing office segments, that themselves contain furniture segments.

If we have an interactive application that allows the user to interactively reposition items within the building plan, using a pick device, then the user could be allowed to pick all or part of a desk, but 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 full description of GSCORS, see the *GDDM Base Programming Reference* manual.

Interactive graphics with multiple partitions

The calls for defining logical input devices described in this chapter 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.

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 work station 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

The preceding sections of this chapter refer primarily to the 3270-PC/G and /GX. The following sections describe functional variations on other types of device.

Interactive graphics on 3179-G terminals

The main differences affecting interactive graphics are that the following are not supported:

- No tablet
- String and stroke devices
- Locator echoes
- Rubber-band locators
- Rubber-box locators.

Interactive graphics on ordinary 3270 terminals

Ordinary members of the IBM 3270 family that use programmed symbols for graphics, such as the 3279, have less graphics capability than the 3270-PC/G and /GX family of work stations. The main differences affecting interactive graphics are:

- The ordinary 3270 terminals have 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 ordinary 3270 terminals have 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 ordinary 3270 terminals, the keys that can trigger input or act as choice devices are different from those on a 3270-PC/G or /GX.

More information about devices is given in Appendix B, "Device-independent programming tips" on page 513.

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 Figure 68.

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		

Figure 68. Choice data returned by non-PC 3270 terminals

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 will trigger 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.

Interactive graphics on the IBM 5080 graphics system

The IBM 5080 Graphics System is designed for polyline CAD/CAM applications.

GDDM/MVS, GDDM/VM, and GDDM-PGF communicate with the 5080 through GDDM/graPHIGS, a separate IBM licensed program. This support allows graphics applications written for other devices to be run on a 5080. Interactive graphics applications written for other types of display will run on the 5080 but will not take advantage of its full capabilities.

The 5080, 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 must be explicitly opened by a DSOPEN call. See "Processing option for the 5080 graphics system" on page 390

- The 5080 does not have 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 chapter, with the following 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.

Part 3. Advanced text

Chapter 15. Symbol sets

Most of this chapter applies to devices with programmed symbols. For device variations, including the 3270-PC/G and /GX, see "Device variations" on page 233. GDDM has two different types of symbols or characters: image symbols and vector symbols. Printer fonts, which are not part of GDDM but which GDDM programs can use, are described in "Chapter 22. Using printers" on page 395.

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 allow the user to create his own image symbol sets. This editor is described in the *GDDM Image Symbol Editor*. 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 lines. When vector symbols are displayed, GDDM is able 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 allow the user to create his own vector symbol sets. This editor is described in the *GDDM-PGF Vector Symbol Editor*. Once created, both image and vector symbol sets are saved on disk for subsequent use by GDDM programs.

Figure 69 on page 220 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' to X'FF' for vector symbols or X'41' to X'FE' for image symbols), or as the EBCDIC character normally occupying that position. Most symbol sets contain representations of a 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) which has, say, 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 Programming Reference* manual, and illustrated in the user's guides for their respective symbol editors.

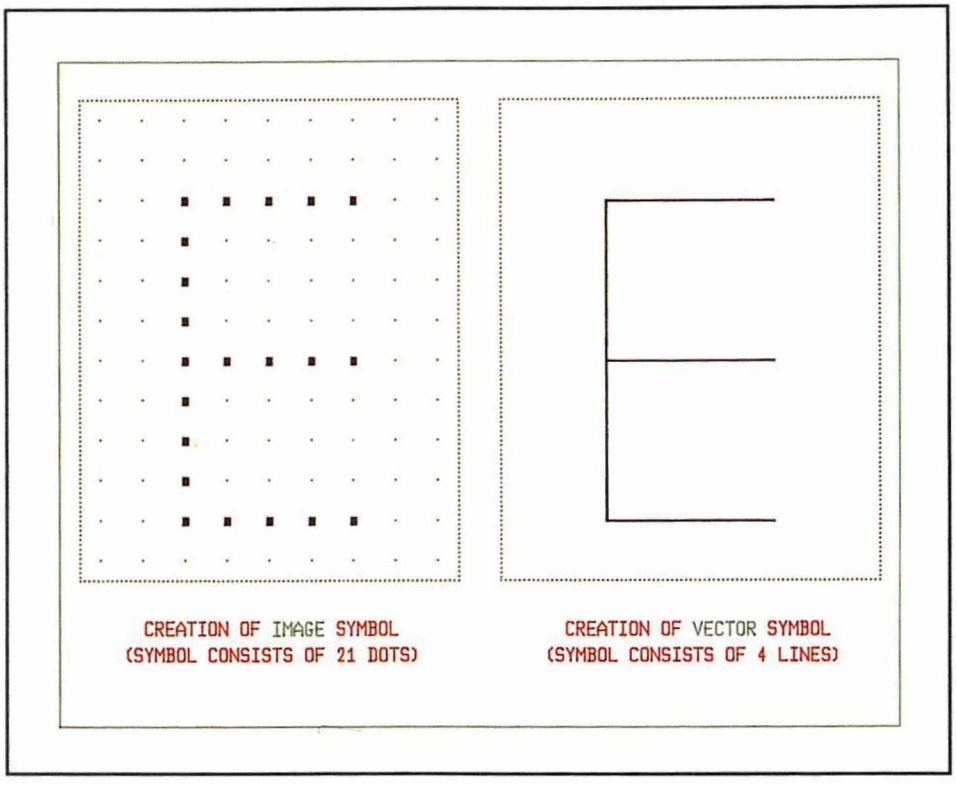


Figure 69. Comparison of image and vector symbols

Using symbol sets

You do not need to specify a symbol set for either graphics or alphanumeric text: GDDM will always supply a default. If you want to use a nondefault symbol set, there are two operations that your program must perform.

The first is to load the required symbol set into main storage. 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.

The operations are steps 1 and 3 in Figure 70 on page 221, which shows them in the context of the other major text output calls. Subsequent sections describe the calls used in these two steps.

ALPHANUMERICS

1. Load the symbol set
```
CALL PSLSS(0,'ADMITALC',199);
```

2. Define the field
```
CALL ASDFLD(33,10,20,2,8,0);
```

3. Specify which loaded symbol set to use
```
CALL ASFPSS(33,199);
```

4. Write the characters onto the page
```
CALL ASCPUT(33,4,'TEXT');
```

5. Send the page to the terminal
```
CALL ASREAD(TYPE,NUM,COUNT);
```

GRAPHICS TEXT

1. Load the symbol set
```
CALL GSLSS(1,'ADMITALC',199);
     or
CALL PSLSS(0,'ADMITALC',199);
```

2. Set the character mode
```
CALL GSCM(2);
```

3. Specify which loaded symbol set to use
```
CALL GSCS(199);
```

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);
```

Figure 70. Overview of symbol set calls

Loading symbol sets

Each symbol set has a name of up to eight characters. On most subsystems this will be 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'.

Within a GDDM program, symbol sets are referred to by name only at the time they are loaded. After that they are referred to by their identifier, which is a number that is allocated at load time.

Symbol sets for alphanumerics

Only image symbols are allowed with the alphanumerics API, and they should be the same size as the hardware cells of the current device.

This is a typical statement to load a symbol set from auxiliary storage. When the current page is sent to the terminal (typically, when the next ASREAD is executed), GDDM will load the symbol set into a PS-store at the terminal.

```
CALL PSLSS(3,'SCRIPT55',193);/* Load symbol set into PS-store 3 */
```

The three parameters have these meanings:

- 3 designates which of the device's PS stores should be used to hold the symbol set. A common setting of this parameter is 0, which tells GDDM to choose a PS store itself.
- SCRIPT55 specifies the name of the symbol set to be loaded from auxiliary storage. Remember that this symbol set must be of the type that matches the hardware cell size (if the alphanumerics will appear on a 3279, for example, the symbol set must be 9 pixels by 12).

- 193 is the number by which future reference will be made to this loaded symbol set. It is known as the **symbol-set identifier** and must lie in the range 65 through 223.

Information about using the symbol set you have loaded is given in "Specifying a symbol set for alphanumeric text" on page 223.

Symbol sets for graphics text

For mode-2, 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:

```
CALL GSLSS(1,'ADMITALC',194);/* Load image symbol set ADMITALC  */
                             /* from auxiliary storage, and     */
                             /* give it an identifier of 194    */
```

The three parameters to GSLSS are as follows:

- The first parameter indicates the **type** of symbol set being loaded:

  1 Indicates that an image symbol set is being loaded.
  2 Denotes a vector symbol set.
  3 A pattern set. This is a special type of image symbol set. Its use is described in "Setting the current pattern, using call GSPAT" on page 38.
  4 A marker set. This is a special type of image or vector symbol set. Its use is described in "Setting the current marker symbol, using call GSMS" on page 37.
  5 A printer font. These are described in "Chapter 22. Using printers" on page 395.

- ADMITALC is the 8-character name of the symbol set, as it was, for example, on PSLSS.

- 194 is the symbol set identifier.

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 "Differences on IBM 3270-PC/G and /GX work stations" on page 233).

Imagine, for example, that a subsequent request is made to send the characters XYZ to the screen of an IBM 3279 terminal using this mode-2 italic symbol set. Then GDDM will retrieve from the symbol set the dot patterns at positions X, Y, and Z. It will then merge these pixels (in the current color) with the pixels representing the rest of the specified graphics. All this processing takes place in the host, not at the device.

For mode-1 graphics text, only image symbol sets are allowed. 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 full details, see the *GDDM Base Programming Reference* manual.

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 will ensure 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" on page 226.

PS-stores for symbol sets and graphics

On the 3279 and most 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 chapter.)

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 is performed, or (2) the PSLSS statement itself can be issued before any graphics is performed, or (3) the first PSLSS parameter may be set to 0 to ask GDDM to choose a PS store not currently in use.

Specifying a symbol set for alphanumeric text

This section explains how to use a symbol set for alphanumerics after you have loaded it, as described in "Symbol sets for alphanumerics" on page 221. To use a loaded symbol set for graphics text, see "Specifying a symbol set for graphics text" on page 226.

To use a symbol set, you specify it as an alphanumeric attribute.

Field symbol-set attributes

The ASFPSS call sets the field symbol-set attribute:

```
CALL ASFPSS(8,193);          /* Set field symbol-set attribute */
```

This call specifies that all subsequent output to alphanumeric field 8 should use the loaded symbol set that was given the identifier 193. Should the field have been

defined to accept input, any characters entered into the field will appear on the screen as symbols from the same loaded symbol set.

Most commonly the symbol set loaded into the PS store will be a font of some sort. If it is, say, a Gothic font, the effect of the ASFPSS and a CALL ASCPUT(8,6,'ABC123') will be to send a Gothic version of 'ABC123' to the screen. Any input to the field will also appear in Gothic characters immediately it is typed.

Setting the symbol set attribute to 0 requests the hardware non-loadable symbol set (in other words, the standard character set of the device). This symbol set is also the one used if no ASFPSS call is executed for a field.

Character symbol-set attributes

When it is required 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). This is not a suitable method for character attributes. The symbol-set identifiers are therefore converted to 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 will use whichever symbol set was specified in the field attribute (the ASFPSS call, if any); this is the meaning of the blanks here. Should the field have more than 6 characters in it, the remainder will also take their attribute from the field-attribute specification.

All character-attribute specifications must follow the corresponding ASCPUT statement for that field. They act on the data in the field rather than on the field itself.

The effect of typical ASFPSS and ASCSS calls may be seen in Figure 28 on page 81.

Here is an example of how to use symbol sets:

```
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 will all appear in italic    */
 /* FIELD 2: PQRST will 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 will use          */
                                  /* s-set 'B' (X'C2', 194)      */
CALL ASREAD(TYPE,MOD,COUNT);      /* Send 2nd output to screen   */
 /*************************************************/
 /*                                               */
 /* FIELD 1: A...EFG  will appear in italic       */
 /*          .BCD...  will appear in Gothic       */
 /* FIELD 2: PQRST will 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 will use         */
                                  /* s-set 'C' ( X'C3', 195)    */
CALL ASFPSS(2,0);                 /* Reset field 2 to the       */
                                  /* hardware non-loadable set  */
CALL ASREAD(TYPE,MOD,COUNT);      /* Send 3rd output to screen  */
 /*************************************************/
 /*                                               */
 /* FIELD 1: HIJKLMN  will all appear in italic   */
 /* FIELD 2: PQR.. will appear in italic          */
 /*          ...ST will appear in the default     */
 /*          (hardware) symbol set                */
 /*                                               */
 /*************************************************/
```

Changing a field attribute will alter the appearance of the data in that field next time a screen output is performed. This applies even if the data of the field was first sent out on a previous screen output. The same is true of character attributes. The new attributes will be 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 will be 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 will appear on the screen in Gothic. A character attribute of PS-store 3 will be returned to GDDM for all such characters. You may query the input symbol-set character-attributes 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 will be in the same form as in a corresponding ASCSS call. In other words, there will be a B (X'C2', decimal 194) for all positions where the character attribute was set to PSB (the location of the Gothic font).

Specifying a symbol set for graphics text

This section tells you how to use a symbol set for graphics text after you have loaded it as described in "Symbol sets for graphics text" on page 222. Using a loaded symbol set for alphanumerics is described in "Specifying a symbol set for alphanumeric text" on page 223.

This is the call that specifies which symbol set should be used:

```
CALL GSCS(194);                 /* Set symbol set attribute to 194 */
```

The actual symbol set used depends on both this parameter and the character mode. It is possible to load three symbol sets (a hardware set, an image set, and a vector set), each with a symbol-set identifier of 194. On most types of terminal, the chosen character mode then determines which of these sets is used. However, this is not always the case on 3270-PC/G and /GX work stations - the selection depends on other factors and is not readily predictable. To ensure device-independence, duplicate identifiers should therefore be avoided in all programs.

If no GSCS call is made, GDDM will use the default symbol set for the current character mode. The defaults are:

- For mode-1 characters, the hardware symbol set of the device in use.
- For mode-2 the set named ADMDHIIx, where x is a code letter that depends on the device being used (see "Device-dependent symbol-set suffixes" on page 228).
- For mode-3, the set ADMDVSS.

The example program in Figure 71 on page 227 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 is used. It is a large and complex one, comprising a multicolored maze. It has a character code of X'C1', which corresponds to the letter A in EBCDIC. As explained in "Multicolored symbols" on page 228, 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    */

/*****************************************************************/
/*                     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 terminal               */

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

 END MAZE;
```

Figure 71. Program using symbol sets for graphics text

Multicolored 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 will all be in that color.

Symbols for pounds, dollars, and cents

Some EBCDIC character codes are reserved for "national use" characters. The appearance of these characters on the screen varies from country to country. This is achieved by configuring the controller appropriately and modifying the display unit. When such character codes are displayed using GDDM-supplied symbol sets, apparent conflicts may occur.

In the U.S., for example, character codes X'5B' and X'4A' represent dollar and cent signs respectively. The GDDM-supplied symbol sets therefore reflect this. A British keyboard (and terminal) uses these codes to represent pound and dollar signs, respectively. Suppose that you code on a British terminal:

```
CALL ASCPUT(F_ID,3,'$85');
or
CALL GSCHAR(X,Y,3,'$85');
```

Character code '4A' will be used, as the terminal will associate the $ symbol with this character code. With the default symbol set, the ASCPUT would cause the string to appear as $85, and so would the GSCHAR if mode-1 (hardware characters) were in use.

This problem may be solved with loadable symbol sets by using the GDDM symbol editors to set all their national-use characters to the desired values.

Device-dependent symbol-set suffixes

To allow 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 will replace the substitution character with one from a set of device-dependent one-character suffixes.

A list of suffixes is given in the *GDDM Base Programming Reference* manual.

Manipulating symbol sets by program

You may need to manipulate GDDM symbol sets in your application programs. This section summarizes some useful calls. For full descriptions of them, see the *GDDM Base Programming Reference* manual.

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 symbol sets

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 will take 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 will not be performed. This scheme may be used even when the PS-store was loaded by a different instance of GDDM.

Querying, reserving, and releasing 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);
```

The following call releases the symbol set with identifier 194 from the PS-store containing it:

```
CALL PSRSS(194);
```

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);
```

Double-byte character set 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 5550 Multistation, you can display DBCS characters in alphanumeric fields, and receive DBCS alphanumeric input, as explained in "Double-byte character set alphanumerics" on page 245.)

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 two special multi-page symbol sets for Kanji text – an image symbol set for mode-2 and a vector symbol set for mode-3. To indicate that you require a double-byte symbol set, you can specify a special symbol-set identifier of 8 in a GSCS call. An example is given in the code below.

Here is an example of program source code:

```
/*****************************************************************/
/*           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);                          /* Special Kanji symbol set */

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           */
```

Another method, which allows ordinary single-byte and double-byte characters to be mixed in a single string, 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     */
UNSPEC(SO)='00001110'B;               /* Set shift-out codepoint  */
UNSPEC(SI)='00001111'B;               /* 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);
```

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 can have the following values:

0 Default, the same as value 1

1 SO/SI codes are displayed as blanks between the single-byte and double-byte characters

2 No blanks are displayed between the single-byte and double-byte characters.

Before executing such a program, you need to specify a GDDM external default.

GDDM default required for Kanji

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 defaults mechanism, which is similar to the nicknames mechanism described in "Nicknames" on page 378. This statement specifies the required default:

```
ADMMDFT MIXSOSI=YES
```

It can be passed to GDDM in any of the ways described in "How to pass nickname statements to GDDM" on page 384. 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 Programming Reference* manual.

Device variations

The preceding sections of this chapter refer primarily to members of the 3270 family that use programmed symbols for graphics, such as the 3279. However, most functions are device-independent, so most of the information applies to all graphics devices. The following sections describe functional variations on other types of device.

Differences on IBM 3270-PC/G and /GX work stations

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 work station 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 3270-PC/G and /GX do not use PS stores to draw such graphics primitives as straight lines and arcs. The section "PS-stores for symbol sets and graphics" on page 223 therefore does not apply.

If a program is transferred between a 3270-PC work station and an ordinary 3270 terminal such as the 3279, the aspect ratio of image symbols change because the aspect ratios of the display units' pixels are different. This is the case whether the symbol sets are loaded by a PSLSS or a GSLSS call.

The GSLSS call: The GSLSS call uses the same work-station storage as graphics orders. Unlike on the 3279, the GSLSS call generally loads the symbol sets into the work station. 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 a 3279, you should not use the same symbol-set identifier in a PSLSS as in a GSLSS because it is unpredictable which will 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 text.

Default symbol set: The work stations have built-in image and vector symbol sets. For mode-1 and -2, the work-station 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 "Default symbol sets for graphics text" on page 389). In this case, the default symbol sets are the same as on a 3279 (see "Specifying a symbol set for graphics text" on page 226), except for device-dependent suffixes.

Differences on composed-page printers

This section describes mode-1 and -2 text on composed-page printers such as the IBM 4250 and 3800 Models 3 and 8. In particular, it describes the differences between these devices and ordinary members of the IBM 3270 family, such as the 3279. There are no differences with mode-3 text.

Alphanumerics: Alphanumerics are not supported on these devices.

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 "Differences on composed-page printers" on page 71).

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.

Differences on plotters

Graphics text support for plotters is similar to that for 3270 terminals such as the 3279. The default symbol set for all text modes is the vector set ADMDVSS. Some further information is given in "Symbol sets" on page 439.

Plotters do not support alphanumerics.

Chapter 16. Advanced procedural alphanumerics

Several of the simpler alphanumeric calls were discussed in "Chapter 8. Basic alphanumerics" on page 75. This chapter covers the rest of the alphanumerics API, dealing with these topics:

- Defining multiple fields
- Specifying default field attributes
- Querying modified fields
- Using light-pen fields.

Defining multiple fields using call ASRFMT

This call defines several alphanumeric fields at the same time. When a number of fields are logically connected, it is useful for the definitions to be grouped together in this way.

The call also permits the field attributes to be set (unlike ASDFLD – define single field). This is an example of the call:

```
DCL ASR_ARRAY(36) FIXED BIN(31) INIT(
   4, 12, 50, 1, 14, 2, 1, 6, 194,   /* Parameters for field  4 */
   7, 12, 68, 1,  4, 0, 1, 4,   0,   /* Parameters for field  7 */
   8, 23,  1, 2,  8, 2, 2, 4,   0,   /* Parameters for field  8 */
  19, 14, 60, 6, 10, 2, 1, 2, 195 ); /* Parameters for field 19 */

CALL ASRFMT(4,9,ASR_ARRAY);  /* Define four alphanumeric fields */
```

The call has these three parameters:

4	The number of fields to be defined.
9	The number of attributes that will be provided per field. It must be at least 5 and not more than 17.
ASR_ARRAY	An array of fullwords containing the attributes for the fields concerned. The number of elements in the array will be the product of the first two parameters (in this case, 36). As the example shows, the parameters for one field precede the parameters for the next field.

You may specify the following parameters for each field. Any not specified, or given a value of zero, take their default values.

1. Field identifier

2. Row – a zero value here will request deletion of the field (if it already exists)
3. Column
4. Depth
5. Width
6. Type – protected, unprotected, and so on, as normally specified by ASFTYP or the last parameter of ASDFLD
7. Intensity – as for ASFINT
8. Color – as for ASFCOL
9. Symbol set – as for ASFPSS
10. Highlight – as for ASFHLT
11. Field end attribute – as for ASFEND
12. Blank to null conversion – as for ASFOUT
13. Null to blank conversion – as for ASFIN
14. Translation table number – as for ASFTRN
15. Transparency – as for ASFTRA
16. SO/SI shift control codes – as for ASFSEN
17. Field outlining – as for ASFBDY.

The above example specified that field 4 should have its top left cell at row 12 column 50. The field should be 1 row deep and 14 columns across. The type should be 2 (protected) and the intensity 1 (normal). The color should be 6 (yellow) and the symbol set used should be that with identifier 194. Unspecified attributes such as the highlighting would be set to default.

If any of the field identifiers match those of existing fields, the existing fields will be replaced by the new ones.

Define multiple fields, deleting all previous fields using call ASDFMT

This call has exactly the same format as the call to ASRFMT. The only difference is that ASDFMT causes the deletion of all existing alphanumeric fields (in the current page) before creating the new ones.

```
DCL ASD_ARRAY(21) FIXED BIN(31) INIT(
   4, 12, 50, 1, 14, 2, 1,           /* Parameters for field  4 */
   8, 23,  1, 2,  8, 2, 2,           /* Parameters for field  8 */
  19, 14, 60, 6, 10, 2, 1 );         /* Parameters for field 19 */

CALL ASDFMT(3,7,ASD_ARRAY); /* Delete all existing alphanumeric */
                           /* fields and create three new ones */
```

Defining multiple field attributes using call ASRATT

This is the third of three calls with an identical format. The purpose here is to redefine the attributes of one or more existing fields.

```
DCL ATT_ARRAY(50) FIXED BIN(31) INIT(
   4, 12, 50, 1, 14, 2, 1, 6, 194, 2,   /* Attribs for field  4 */
   7, 12, 68, 1,  4, 0, 1, 4,   0, 2,   /* Attribs for field  7 */
   8, 23,  1, 2,  8, 2, 2, 4,   0, 1,   /* Attribs for field  8 */
  12, 20,  1, 1, 17, 2, 1, 5, 194, 1,   /* Attribs for field 12 */
  19, 14, 60, 6, 10, 2, 1, 2, 195, 2);  /* Attribs for field 19 */

CALL ASRATT(5,10,ATT_ARRAY);      /* Redefine the attributes of  */
                                  /* fields 4, 7, 8, 12, and 19  */
```

The first attribute for each field gives the field identifier. The next four attributes (row, column, depth, and width) are ignored, because changing these four attributes would cause a redefining of the field, which is not the purpose of this statement. The remaining attributes (representing type, intensity, and so on) may be set to the values permitted for ASRFMT.

It is an error if an identifier is given for which no field exists.

Setting default field attributes using call ASDFLT

If you do not specify an attribute such as the color of an alphanumeric field, then a default attribute value will be taken. In the case of color, for example, the default will be green on a color display and black on a printer.

You may wish to change the default value for some of the attributes. If most of your fields are to be blue, say, then setting a default color of blue would save you several calls to ASFCOL. This is a typical call:

```
DCL DEFAULT_ATTRS(5) FIXED BIN(31) INIT(
     2,              /* Type = protected       */
     1,              /* Brightness = normal    */
     1,              /* Color = blue           */
     0,              /* Symbol set = default   */
     4 );            /* Highlight = underscore */

CALL ASDFLT(5,DEFAULT_ATTRS);  /* Define new default values for */
                               /* the first five attributes     */
```

All fields subsequently defined will be subject to the new defaults. Note that the new defaults apply only to the current page.

The first parameter, 5, gives the number of elements in the attribute array. It must be a number from 1 through 12. The twelve parameters that may be specified are the same as the last twelve (that is, from type onward) of those listed in "Defining multiple fields using call ASRFMT" on page 235.

To set just the fifth parameter, you must also set the first four (they can be set to -1 if the existing default is satisfactory). The remainder of the attributes (6-12, in this case), will resume their normal default.

Querying modified fields using call ASQMOD

Your application program may offer the terminal operator several alphanumeric fields for input and may then need to know which fields the operator has modified. The ASQMOD call returns details of all fields modified since the previous ASQMOD call. This is the format of the call:

```
DCL FIELD_IDS(6) FIXED BIN(31);  /* Array to hold field ids     */
DCL LENGTHS(6) FIXED BIN(31);    /* Array to hold field lengths */
DCL I_LENGTHS(6) FIXED BIN(31);  /* Array to hold input lengths */

CALL ASQMOD(6,FIELD_IDS,LENGTHS,I_LENGTHS);  /* Query the first */
                                             /* six modified fields */
```

This will return information on up to six modified fields. If, say, only four fields have been modified, then the first four elements of each of the three arrays will be set on return. The identifiers of the fifth and sixth fields would be set to zero.

The parameters of the call are:

6
: The maximum number of fields to be queried. Obviously this number must not exceed the dimension of any of the three receiving arrays.

  If more than six fields have been modified, information on only the first six will be returned. These six fields will then be set to unmodified status. A subsequent ASQMOD call would be issued to retrieve information on the remaining modified fields.

FIELD_IDS
: A fullword array to receive the identifiers of the fields that have been modified.

LENGTHS
: A fullword array to receive the total defined length of the fields in question.

I_LENGTHS
: A fullword array to receive the input lengths. For example, a field may be of length 5 but the operator may type only ABC into it. The input length for that field would then be 3.

  For the 5550 family, the input length of the string includes SO/SI codes inserted by GDDM.

You can discover how many modified fields there are on the current page by executing an ASQNMF call:

```
DCL COUNT FIXED BIN(31);
CALL ASQNMF(COUNT);
```

The number will be returned in COUNT.

Alphanumeric field status

The status of an alphanumeric 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 set to modified status by issuing an ASFMOD call, for example:

```
CALL ASFMOD(23,1);                    /* Mark field 23 as modified */
```

The first parameter is the field identifier. The second is set to 1 to set the status to 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, say, 6 fields*/

CALL ASQMOD(4,F_IDS,LENGS,ILENGS);           /* Query first four  */
                                             /* modified fields   */

 /*The 4 queried fields will now be marked unmodified, */
 /*leaving just 2 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 4 queried fields will now be 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, 4 at a time                                   */

END;
```

One of the uses of ASQMOD is in applications that use a menu. Some fields in a menu are protected, acting as prompts to the operator. Others are unprotected, and the operator may type into them. The program must determine which fields have been entered before it can process them appropriately.

Alphanumeric menu sample program

Here is a program that manipulates alphanumeric fields with the help of an ASQMOD call.

Sample output from the program is shown in Figure 72 on page 243.

```
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);             /*****************************/
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  */
                                        /* 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(
           101, 6,35,1,25,2,1,0,193, /*************************/
           102, 7,35,1,25,2,1,0,193, /* Attributes for multiple */
           103, 8,35,1,25,2,1,0,193, /* definition of 9 fields  */
           104,12,35,1,25,2,1,0,193, /*************************/
           105,13,35,1,25,2,1,0,193,
           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*/
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'); /***************/
CALL ASDFLD(50,24,14,1,42,2);
CALL ASFPSS(50,193);                    /* Italic symbol set       */
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 */
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);  /* Query modified fields */
DO I=1 TO 3;                             /* Process the order  */
                                         /* 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);
                                       /*Chosen item to yellow */
IF FIELD_IDS(I)=12
  THEN WINE=CHAR1;                 /* Wine to match main course */
END; /* VALID ENTRY */
END; /* I-LOOP */

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;
```

Note the following points about the program:

- **Discovering how many fields were modified.** The logic of the program allows for the operator having entered no fields at all or up to three fields. There are two techniques used to discover how many fields were entered. The simplest one is to inspect the COUNT variable returned by the call to ASREAD. Immediately after the ASREAD call, the program tests to see if COUNT is zero. If so, it reissues the ASREAD.

  If there are fewer modified fields than the number requested on the ASQMOD call, the remaining entries in the passed arrays will be 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.

- **Choosing field identifiers to advantage.** The identifiers of related alphanumeric fields are often chosen to be sequential so the fields may be processed in a loop.

Figure 72. Output from "Restaurant Menu" sample program

How to use 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, say, `TOTAL PROFITS`, you must issue:

  ```
  CALL ASCPUT(8,14,' TOTAL PROFITS');
  ```

  The field will then appear 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 non-light-pen fields) before the operator causes an interrupt, for instance, by pressing ENTER. All modified and selected fields will now be returned to GDDM. See "Querying modified fields using call ASQMOD" on page 238 for information on how to process the returned fields.

  As 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 will return 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 (type 2 = light-pen attention).

  Warning: 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 type parameter may be set as follows:

3 Light-pen attention field
4 Light-pen selection field
5 Light-pen enter field
6 General light-pen field.

The same parameter settings may be used to change the type of a field, using the ASFTYP call. (See "Field attributes" on page 79.)

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 shown in "Alphanumeric menu sample program" on page 240. Selected fields will be marked as modified and may be determined by a call to ASQMOD.

Double-byte character set alphanumerics

The IBM 5550 multistation is a special member of the 3270 family that will display and print, in alphanumeric fields, the double-byte character set (DBCS) characters used in some Asian countries. Each DBCS character is represented in the data stream by a two-byte code, instead of the single-byte EBCDIC code used for Latin characters. Both input and output of DBCS characters is supported by GDDM.

GDDM supports input and output of the double-byte character codes on other terminals, but these cannot display alphanumeric DBCS characters. However, DBCS text can be displayed on terminals that support graphics, as explained in "Double-byte character set graphics text" on page 230.

IBM 5550 multistation

To display a DBCS string in an alphanumeric field, you can specify the hardware double-byte character set in an ASFPSS call. It has the special identifier 248 (X'F8'). Then you must supply the hexadecimal codes for the required DBCS characters as data in an ASCPUT call. The length you specify must be the actual length in bytes of the data – twice the number of DBCS characters. The field size, as defined in an ASDFLD call, must be big enough to accommodate this number of bytes. Here is an example:

```
/*************************************************************/
/*              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 CREATE FIELD CONTAINING KANJI DATA       */
/*************************************************************/
    /* FIELD-ID  ROW  COLUMN  DEPTH  WIDTH  TYPE */
CALL ASDFLD(77,   7,    7,      1,     10,   0); /*10-byte field*/

CALL ASFPSS(77,248);             /* Specify Kanji character set */

DECLARE KANJI_DATA5 CHARACTER(10);  /* String for 5 Kanji chars */
KANJI_DATA5=KC(65)  || KC(192) ||   /* Assign                   */
            KC(...) || KC(...) ||   /*     five                 */
            KC(...) || KC(...) ||   /*       two-byte           */
            KC(...) || KC(...) ||   /*           Kanji          */
            KC(...) || KC(...);     /*               characters */
CALL ASCPUT(77,10,KANJI_DATA5);     /* Put characters into field*/
```

Input data returned by an ASCGET call will contain the double-byte representations of the DBCS characters entered by the operator.

If you want to mix SBCS and DBCS strings in a single field, you must use another method, where you define a **mixed field** in your program.

For input, the initial mode of the 5550 is SBCS mode, and the terminal operator can enter SBCS characters. To enter DBCS characters, the terminal operator first presses the DBCS key (Alt + SBCS) to force the input mode of the field to DBCS. DBCS characters can then be entered. After entering the DBCS string, pressing the SBCS key returns the terminal operator to SBCS mode, so further single-byte characters can be entered. The terminal operator can check the current input mode by looking at the width of the cursor (in DBCS mode, the cursor appears twice as long as in SBCS mode), or looking at the shift status field in the operator information area at the bottom of the screen.

If you want the application to accept mixed SBCS and DBCS input, you should use the ASFSEN call to specify a mixed field. Here is an example:

```
CALL ASFSEN(16,1);        /* Specify mixed strings for field 16 */
```

The first parameter is the field identifier.

The second parameter is used to specify whether the field can contain a mixed string. It can have the following values:

-1 Leave the mixed status of the field as it is

0 Nonmixed (the default)

1 Mixed with position.

For input, when the terminal operator changes input mode between SBCS and DBCS, enters one or more DBCS characters, and changes back to SBCS, shift-out (SO) and shift-in (SI) control characters are inserted by the terminal at the boundaries between SBCS and DBCS characters, or between the field boundary and DBCS characters. (If no DBCS characters are entered, no SO/SI codes are inserted.) The data between these two special characters is interpreted by the terminal as double-byte. Other characters are interpreted as single-byte.

For output, you specify SO/SI codes in the definition of the field contents in your program.

For input and output, the SO/SI codes occupy a character position each, that is displayed as a blank.

2 Mixed without position.

For input, the terminal user changes the input mode between SBCS and DBCS, as for mixed-with-position fields. However, no blanks appear between the SBCS and DBCS characters. Instead, GDDM interprets the changes in the symbol set character attribute, and inserts delimiting SO/SI codes around the DBCS portions of a string.

For output, you specify SO/SI codes in the definition of the field in your program. GDDM interprets the SO/SI control codes, and generates the necessary DBCS character attributes when the mixed field is sent to the device. So again, no blanks appear between the single-byte and double-byte characters.

If the field is defined as mixed, you cannot use an ASFPSS call to set the field to DBCS, but you must still supply the hexadecimal codes for the required DBCS characters, as shown in the previous example.

The SO code is X'0E' and the SI is X'0F'. In the definitions of the fields in your program, you must allow one byte for each of these and two bytes for each DBCS character. Within any field, only SO/SI pairs are allowed, in that order.

Here is an example of output using mixed without position. You can easily change it to mixed with position by changing the second parameter of ASFSEN to a 1.

```
/*************************************************************/
/*           SET UP SO/SI CHARACTERS                         */
/*************************************************************/
DCL (SO,SI) CHAR(1);                  /* Shift-out & shift-in      */
UNSPEC(SO)='00001110'B;               /* Set shift-out codepoint   */
UNSPEC(SI)='00001111'B;               /* Set shift-in codepoint    */

/*************************************************************/
/*           CREATE FIELD FOR MIXED KANJI AND LATIN DATA     */
/*************************************************************/
    /* FIELD-ID  ROW  COLUMN  DEPTH  WIDTH  TYPE */
CALL ASDFLD(81,   8,    1,      1,     24,   0);/* 24-byte field*/
CALL ASFSEN(81,2); /* String attribute of mixed without position*/

DECLARE MIXED_DATA28 CHARACTER(28);  /* String for mixed chars  */
                                     /*                   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 ASCPUT(81,28,MIXED_DATA28);
```

If you use the mixed-string attribute of "mixed without position," then:

- For output, 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.

  For calls ASCCOL, ASCHLT, ASCSS, the character attributes corresponding to SO/SI control codes have no effect.

- For input, if you use the mixed-string attribute of "mixed without position," 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 the last reason, it is good practice to issue an ASQLEN call before issuing an ASCGET call, so that your program will know how much storage is needed to hold the returned data. Here is a typical call:

```
CALL ASQLEN(81,FIELD_LENGTH,INPUT_LENGTH,SCREEN_LENGTH);
```

The parameters are as follows:

The first parameter is the field identifier.

In the next three parameters, GDDM returns the following values:

- The length in bytes of the input field as specified in ASDFLD.
- The length of the string (excluding trailing nulls) as held by GDDM, that is, including SO/SI codes inserted by GDDM
- The length of the string (excluding trailing nulls) as it appears to the terminal user, that is excluding SO/SI codes inserted by GDDM

If you use the mixed-string attribute of "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.

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 247:

```
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.

A value of -1 in the second parameter of ASFCUR specifies that the value in the third parameter refers to the byte position of the field contents in your program. Any other value in the second parameter specifies that the value in the third parameter is the column position of the field on the screen.

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 */
```

Specifying a value of 2 in the first parameter will return a value of -1 in ROW. The second parameter is the field identifier. The value returned in COLUMN will be the byte position of the cursor in terms of the contents of field 81 as described in your program.

Other terminals

Even on devices that will not display DBCS characters in alphanumeric fields, the application program can send and receive DBCS codes. The fields must be defined as mixed using the ASFSEN call – this applies for both input and output. And the MIXSOSI GDDM default parameter must be specified (see "GDDM default required for Kanji" on page 232). The programming is then the same as for the 5550.

On output, the program must precede a string of DBCS double-byte codes with an SO character and follow it with an SI. The terminal will display the double-byte codes in hexadecimal.

On input, 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 Programming Reference* manual).

For output, you can use DBCS graphics text (see "Double-byte character set graphics text" on page 230) as an alternative to the alphanumeric output functions described here. The text is displayed in DBCS characters, rather than hexadecimal codes. The operator could specify DBCS text using hexadecimal alphanumeric input, and the application could then display it in DBCS characters using graphics text.

Field outlining on the IBM 5550 multistation

The 5550 will draw a partial or complete outline around a field. You specify outlining with the ASFBDY call:

```
CALL ASFBDY(33,15);/*Type 15 outline (complete box) for field 33*/
```

You specify the type of outlining in the second parameter. Possible values are:

-1 Leave outline attribute for this field unchanged

0 None (the default)

1 Underline

2 Vertical line on right

3 Underline and vertical line on right

4 Overline

5 Overline and underline

6 Overline and vertical line on right

7 Overline, underline, and vertical line on right

8 Vertical line on left

9 Underline and vertical line on left

10 Vertical lines on left and right

11 Underline and vertical lines on left and right

12 Overline and vertical line on left

13 Overline, underline, and vertical line on left

14 Overline and vertical lines on left and right

15 Complete box

The purpose of partial field outlining is to allow you to build up outlines around rectangular blocks of fields.

Chapter 17. Mapped alphanumerics

If you create a display that includes 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*. Initially you should refer to the User's Guide.

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 it 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 will contain any input typed in by the operator.

A simple mapping program is shown in Figure 73 on page 253 and its associated display in Figure 75 on page 254.

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 73. 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, and no details of its presentation to the end user. It demonstrates the major advantage of GDDM mapping - that you concentrate on data processing when you write your program, and leave the presentation entirely to GDDM.

The facilities described in "Chapter 8. Basic alphanumerics" and "Chapter 16. Advanced procedural alphanumerics" are sometimes known as **procedural**

alphanumerics, to distinguish them from the mapping facilities. As for 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 work station, mapped data appears on the alphanumerics screen. On the 5080 graphics system, it appears on the 3270 screen.

Full GDDM mapping support is limited to programs written in PL/I, COBOL, and System/370 Assembler, because only these languages allow 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.

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.

Comparison with procedural alphanumerics

Both procedural alphanumerics and mapping provide an alphanumeric input/output service. When should you use each one?

Procedural alphanumerics are likely to be best for simple data displayed in a small number of fields. In such cases, the overhead of a separate map definition operation may not be justified. And you may find procedural alphanumerics best if you need to alter the layout of the display during execution. Otherwise, you will probably find it well 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. The calls require field locations to be defined in terms of rows and columns, whereas GDDM-IMD allows 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 will be easier to change. If you need to alter the display format, say to take advantage of a new device, 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.

A simple mapping application

The application, called MAPEX01, is for order entry. Initially it displays the fields shown in Figure 75 on page 254.

The terminal operator 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 operator to enter some more information.) If they are not both numeric, the program puts a message on the screen so that the operator can correct the error. The position of the message (the line below the heading) was defined when the map was created.

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 74 on page 254.

The *GDDM Interactive Map Definition* tells you how to invoke GDDM-IMD, and how to use the quick-path tutorial. You can work through the tutorial either before or after reading the following 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 260.

Description of the program

The source code of the program is shown in Figure 73. It illustrates several basic concepts of GDDM mapping.

```
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 73. Source code of MAPEX01

```
          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 74. Field definitions for map used by MAPEX01

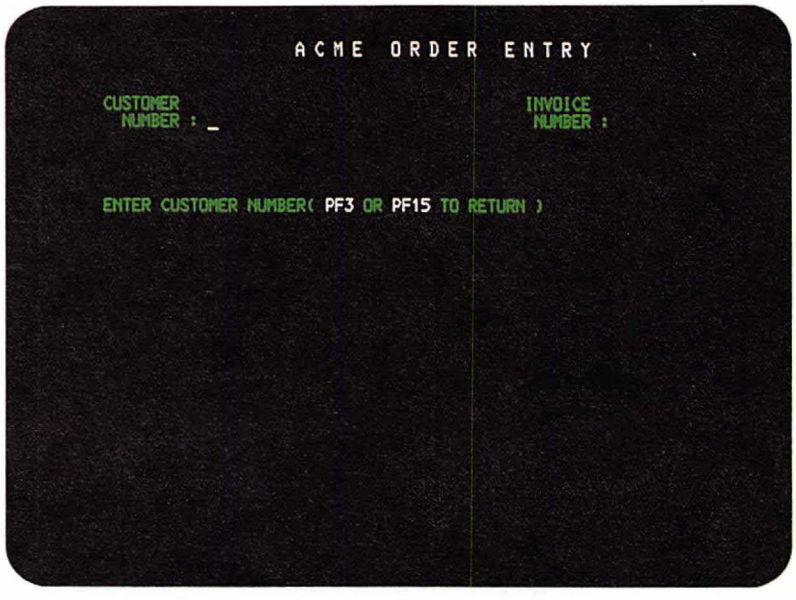


Figure 75. Initial display of MAPEX01

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 will not have to code declarations of ADSs in your programs: they are generated by GDDM-IMD. This example, and all the others in this chapter and "Chapter 18. Variations on a map," show the ADS declarations in full to make the programs easier to follow. All you will 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 whose value 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*/`.

Output and input: The GDDM call that handles mapped I/O is MSREAD, shown at `/*C*/`. It sends the map to the terminal, and then waits for the operator to cause an interrupt, for instance by pressing ENTER. 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.
- 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 have the same meanings as the first two parameters of ASREAD. They are set by GDDM to indicate the type of interrupt received from the terminal. Full details of the possible values are given in "Send output and await reply using call ASREAD" on page 13.

The MSREAD 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 75 on page 254 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.

Compilation and execution

After you have created an ADS using GDDM-IMD, you must store it in a library. More information is given in "Steps in creating a mapping application" on page 260 and in the *GDDM Interactive Map Definition*. To compile a mapping program like the one in Figure 73 on page 253, you must make the library available to the compiler. Under CMS, the following commands will 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 GDDM Program under CMS" on page 11.

Dialog with the terminal operator

MSREAD is limited to simple output and input of single maps. For more complicated dialogs, such as ones that require more than one map in a display, the various functions of MSREAD must be done separately, using a different call for each function. For comparison, Figure 76 on page 257 shows how the program in Figure 73 on page 253 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');
   ```

   The first parameter is the page identifier.

   The last parameter is the name of a mapgroup. All maps used on this page must belong to this mapgroup.

   The second and third parameters are the page depth and width. A -1 means use the dimension specified to GDDM-IMD when the mapgroup was created.

   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');
   ```

   The first parameter is the identifier of the area being mapped, to be used in later references to it.

   The last parameter is the name of the map.

   The second and third parameters define the location of the map on the page. They specify the row and column position of its top left-hand corner. A -1 means use the position specified to GDDM-IMD when the map was created. A value of 0 for either parameter means delete the map from the page.

A mapped area of a page is similar in many respects to a procedural alphanumeric field, and is known as a **mapped field**. However, it is commonly called simply a map, except when this would cause confusion with the format definition created by GDDM-IMD.

3. Copy data from the ADS into the variable data alphanumeric fields contained in the mapped field.

   ```
   CALL MSPUT(1,0,ORDER1_ASLENGTH,CUSTINV);
   ```

   The first parameter is the identifier, as specified by an MSDFLD call, of the mapped field into which the data is to be copied.

   The second parameter is an operation code that specifies which fields are to be updated with data from the ADS. A 0 means update all fields; this is called a write operation. A 1 or 2 means update certain fields only; the operations are called rewrite and reject. The meanings of rewrite and reject are further explained in "Write, rewrite, and reject" on page 276. In the example in Figure 76, the operation code has been put into a mnemonic variable at `/*A*/`.

   The third operand is the length of the ADS, and the last its name.

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. It is described in "Send output and await reply using call ASREAD" on page 13. The parameters have the same meanings, except that the last one gives the number of maps changed by the operator, not the number of alphanumeric fields. In the case of a page with only one map on it, this parameter will always have a value of 0 or 1. 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);
   ```

   The first parameter is the identifier, as specified by an MSDFLD call, of the mapped field from which the data is to be extracted.

   The second parameter is nearly always zero. Its meaning, and the other values, are explained in "Character attributes" on page 295.

   The third parameter is the length of the ADS, and the last its name.

```
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*/
```

Figure 76 (Part 1 of 2). Source code of MAPEX02

```
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 76 (Part 2 of 2). Source code of MAPEX02

Typical mapping cycle

The diagram in Figure 77 shows some of the major steps that a typical mapping program goes through.

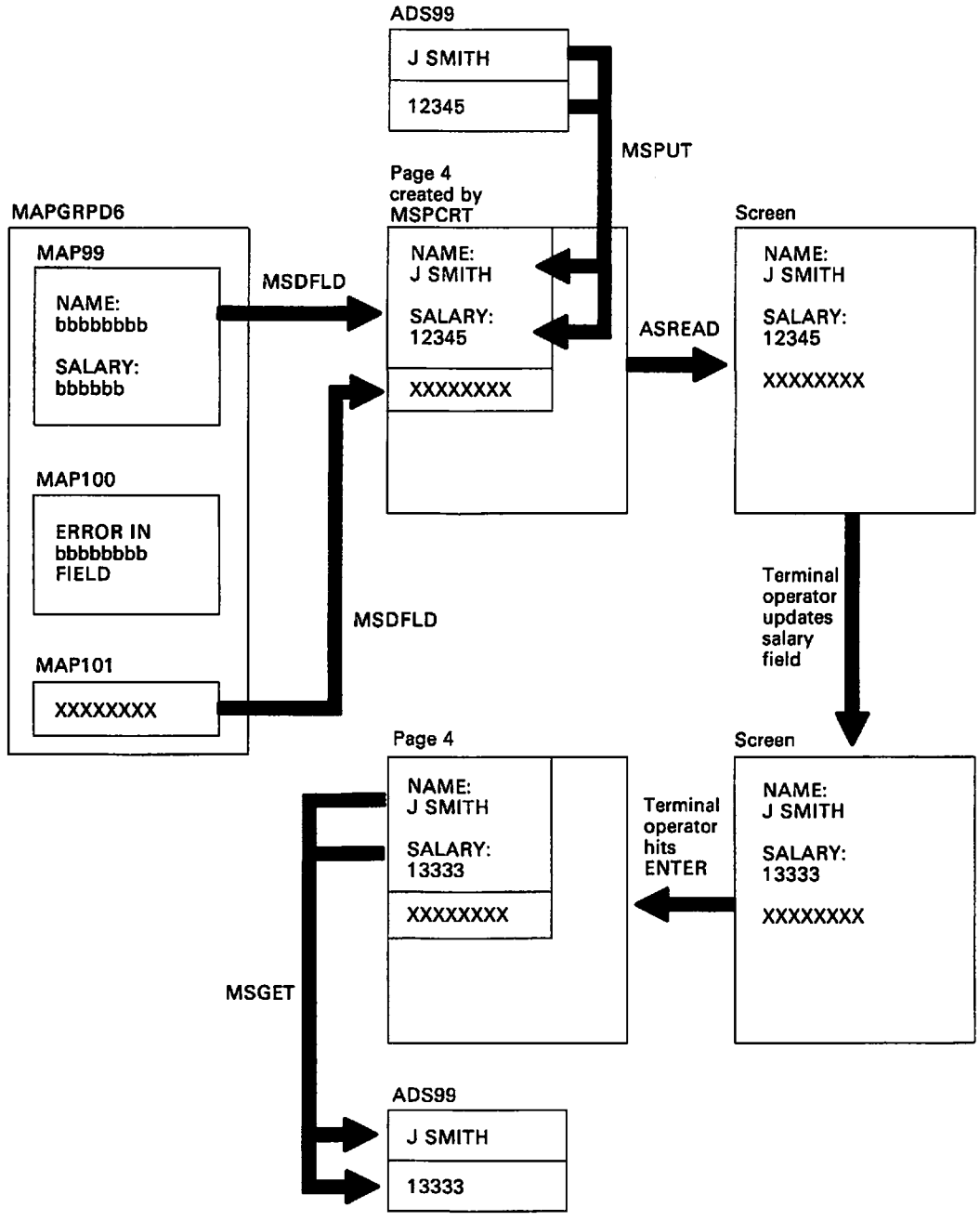


Figure 77. Typical cycle of mapping operations

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 "*********"

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.

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 76.

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.

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. If you are using GDDM-IMD under TSO, allocate the files required to hold the ADSs and the generated mapgroups. More information is given later in this section, and full details of the files are given in the *GDDM Interactive Map Definition*.

   If you are using GDDM-IMD under CMS or CICS/VS, ignore this step.

2. Create the required maps using GDDM-IMD:

   a. Create the mapgroup using the mapgroup editor.

      A simple application such as MAPEX01 requires only one mapgroup, containing only one map. But often you will need several maps in a mapgroup, as explained in "Multiple maps" on page 263. 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 will appear, and the **presentation area,** in rows and columns, that it will occupy.

      For instance, if your application is to run on an IBM 3279 Model 3B 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 will

occupy the whole of the screen. A full list of device classes is given in the *GDDM Interactive Map Definition.*

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 will assign 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 will be 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 will be able to 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 will generate, 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" on page 271.

   GDDM-IMD allows 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.*

   - 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 suffix, and with a default file type of ADMGGMAP. GDDM-IMD will create 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.*

- Under CICS/VS, 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/VS 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/OS/VS, or to a library using the MAINT program for CICS/DOS/VS. The members of the partitioned data set or the books in the library will 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/VS File Control Table (FCT), the default FCT name being ADMF. This definition is usually done when GDDM is installed.

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. But this would mean re-editing the source whenever the ADS changed.

   In theory, you could make yet another choice, to code the ADSs yourself as part of the source code. But unless you use only the most basic functions of mapping, this is more difficult than it might seem, and it is not advised. The reason is that many functions require rather complex ADSs that would be difficult to code without errors. These ADSs, and the functions they support, are described in "Chapter 18. Variations on a map" on page 273.

4. 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/VS applications, you need to execute the IEBUPDTE program (CICS/OS/VS) or MAINT program (CICS/DOS/VS) 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/OS/VS) or ASSGN statement (CICS/DOS/VS) if you compile in batch mode, or a suitable ALLOCATE or GLOBAL MACLIB command if you compile under TSO or CMS.

5. Execute the program.

   GDDM will find the generated mapgroups required by the program with no further action by you (except under IMS, when you must import the generated mapgroups, as explained in the *GDDM Installation and System Management* manual).

   The various GDDM mapping calls that a typical program may need to execute are summarized in "Dialog with the terminal operator" on page 256.

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/VS, to update the secondary source library with any new or changed ADSs.
- If the ADS has changed, to recompile (or reassemble) the program.

Multiple maps

In many circumstances it is convenient to format the display using two or more maps. For instance, you might want to use one map to allow the terminal operator 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 allows you to put many maps into a page if there is space, and provided they do not overlap with other maps. The section "Fixed maps" on page 264 gives further information.

In some types of display, you will 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 allows you to define **floating maps**. You do not have to calculate where to put these on the page. GDDM will position 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 "Floating maps" on page 264.

Fixed maps

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.

When you define a map's characteristics you must take care, when specifying its size and position, that it does not overlap any other map that will be displayed at the same time. It is not an error to define overlapping maps, but you must not try to display them together. Your application program would be in error if it executed an MSDFLD call specifying a map that overlapped one specified in an earlier MSDFLD call.

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 in 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.

Floating maps

The program in Figure 79 on page 267 could be part of an order-entry application. It displays information about as many orders as the screen can accommodate. A typical display is shown in Figure 81 on page 268. It uses one fixed and one floating map. Their formats are shown in Figure 80 on page 268.

It is an output-only application. For handling input data from floating maps, see "Input from multiple copies of a map" on page 270.

Creating floating maps: 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.

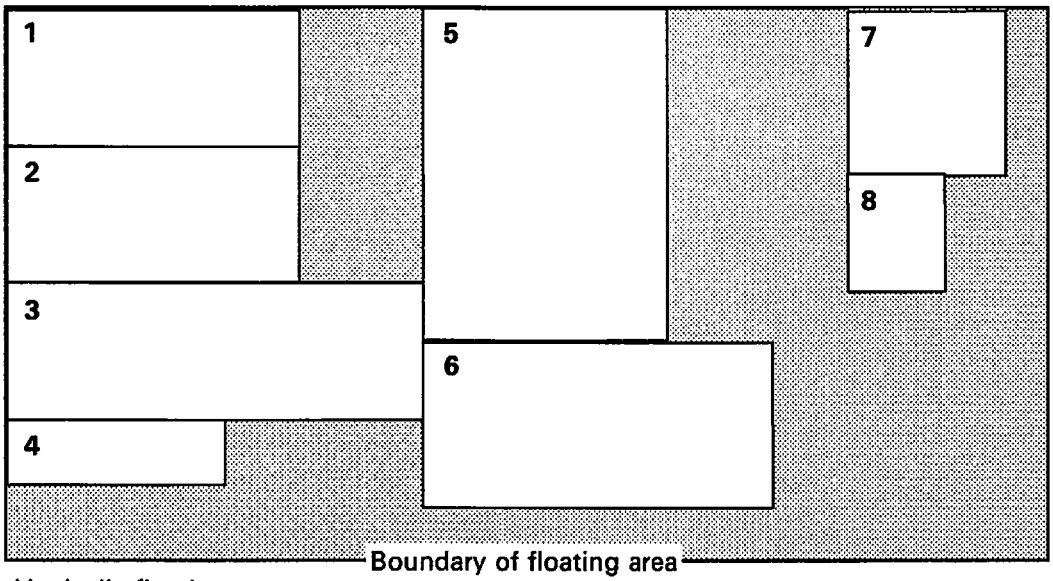
Positioning of fully floating maps is summarized in Figure 78 on page 266. Most applications do not use maps of such varied size as those illustrated. The sizes have been chosen to illustrate GDDM's positioning algorithm, in particular where the second and subsequent horizontal and vertical stacks are 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 (the one specified as SAME) will be the same as for the previous map. If the semifloating map is the first floating map on the page, it will be 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.



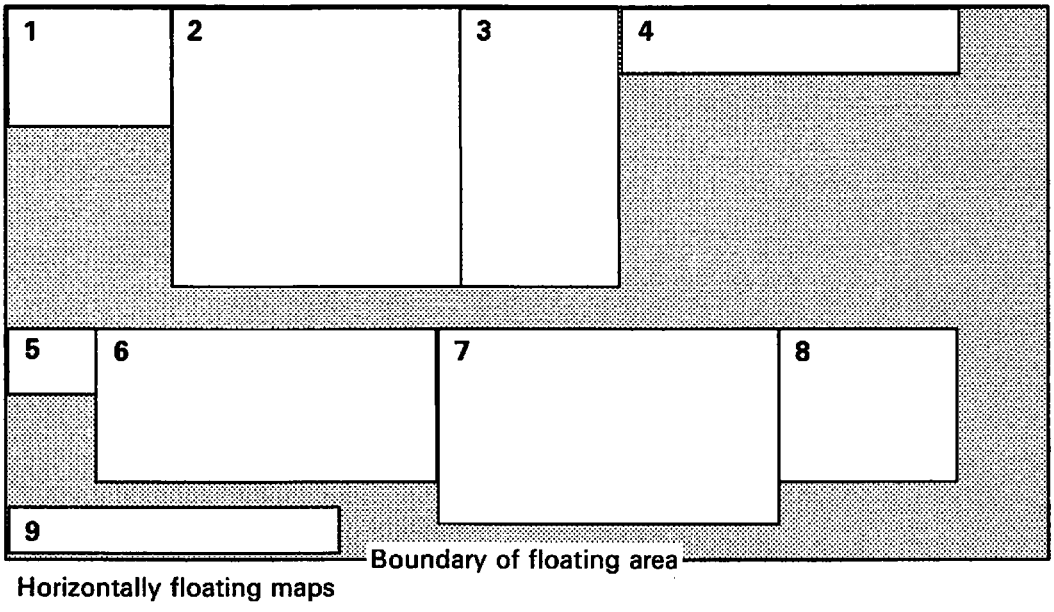


Figure 78. Positioning of fully floating maps

```
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 NUMBER 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.     */

CALL MSDFLD(1,                     /* Format header area of page */ /*D*/
            -1,                    /* at GDDM-IMD defined row   */
            -1,                    /* and column position,      */
            'HEADER');             /* using map header          */

CALL MSQFIT('FLOATER',NUMBER);     /* How many maps to fill page?*/ /*E*/

DO MID = 2 TO NUMBER+1;            /* Put number copier 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 79. 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 80. Field definitions for map used by MAPEX04

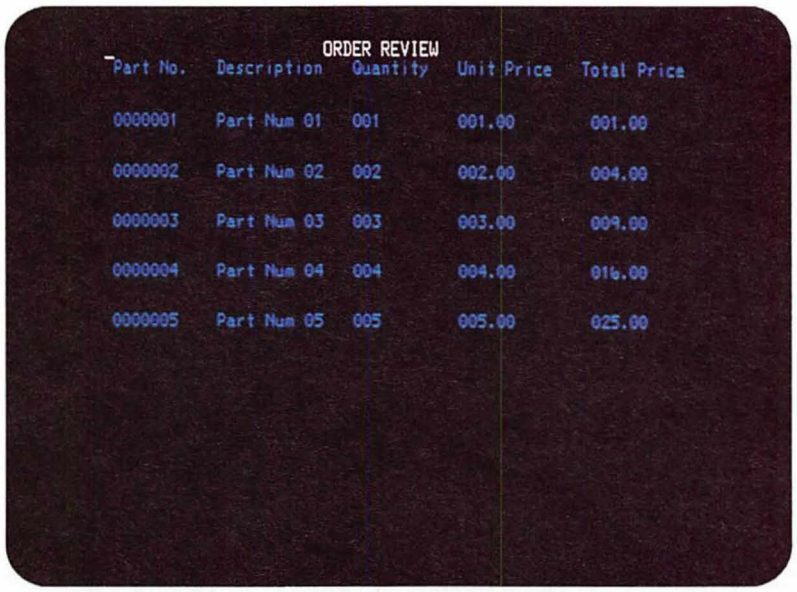


Figure 81. Typical display by MAPEX04

Description of the program: The program in Figure 79 on page 267 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*/`. This call is described in "Calls that operate on pages" on page 94. 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, NUMBER, 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 will 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 NUMBER. 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 NUMBER+1, at `/*F*/`.

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 79 on page 267, 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 NUMBER+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

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 will run 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 will 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*. 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 will 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 459).

If you know that your program will run 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. 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 ENTER, say).

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 269. 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 Programming Reference* manual.

Chapter 18. Variations on a map

"Chapter 17. Mapped alphanumerics" describes how to use maps to supply the basic framework of a dialog with the terminal operator. This chapter 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 facilities for varying the format may seem complicated if you are new to the techniques of mapping. But they are designed to simplify the programming of complex dialogs, by allowing GDDM to do more of the work. They 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 17. Mapped alphanumerics."

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.

The facility works as follows. 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, called a **selector adjunct**, associated with it in the ADS. You must tell GDDM-IMD when you create a map 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 82 on page 275 has a selector adjunct at `/*A*/`. GDDM-IMD gives selector adjuncts the same names as the associated fields, with "_SEL" appended in PL/I, "-SEL" in COBOL, and "S" in Assembler. Selector adjuncts are one byte long.

There are several types of adjunct in addition to selector adjuncts. They are a general control mechanism used for several different purposes. Their uses include: setting field attributes; positioning the cursor; extended highlighting; setting the

color of fields; and programmed symbol selection. Most types are introduced in this chapter; a full list is given in the *GDDM Base Programming Reference* manual.

Error message example using a selector adjunct

The program in Figure 82 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 75 on page 254.

Although their output is similar, the maps used by the two programs differ. In addition to having a selector adjunct, the one 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, will be blank.

If the terminal operator 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);

DECLARE 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);
DECLARE (ATTYPE,ATVAL,COUNT) FIXED BIN(31);/* ASREAD arguments*/
DECLARE WRITE FIXED BIN(31) INIT(0); /* MSPUT write operation */
DECLARE 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*/
                                    /* put out 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 operator input.   */
            COUNT);
IF ATTYPE=1 & (ATVAL=3 | ATVAL=15) /*Operator 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.            */
```

Figure 82 (Part 1 of 2). Source code of MAPEX05

```
IF ¬VALID THEN DO;                  /* Error found, so redisplay  */
  VALID = '1'B;                     /* the map.                   */
  MESSAGE_SEL = '2';                /* Set selector, so that map  */ /*E*/
END;                                /* message will appear.       */
GO TO LOOP;

FIN:

CALL FSTERM;                        /* Terminate GDDM.            */
%INCLUDE ADMUPINA;                  /* GDDM entry declarations.   */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINM;

END MAPEX05;
```

Figure 82 (Part 2 of 2). Source code of MAPEX05

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:

A " " (A blank character)	No further change to the field.
A 1 character	Update the field with variable data from the ADS.
A 2 character	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.
A 3 character	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 " ", as 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 277.

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 operator. 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:

A " " (A blank character)	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.
A 1 character	The field has a new value set by the terminal operator. 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."
A 2 character	Not used on input.
A 3 character	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 operator 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 operator 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 operator, 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 operator

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 operator, but it was not necessarily supplied during the most recent ASREAD.

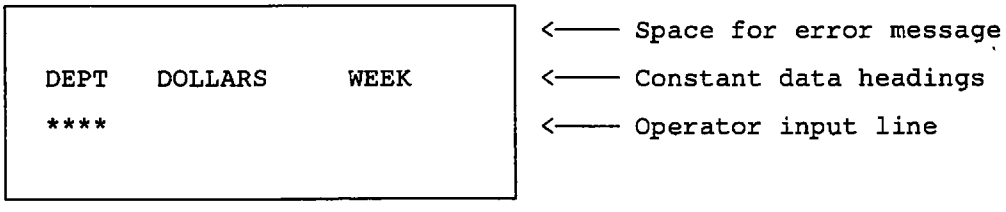
In hardware terms, a reject does not reset the modified data tags (MDTs) of the previously modified fields, whereas write and rewrite do. The possibly different value of the selector adjunct on input is the only way in which this difference is apparent to your program.

The reject facility allows 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:



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 created the display by setting the ADS to all-blanks. The four blank selectors cause the ASREAD to send the default data to the screen, which is asterisks in the department code field, and blanks in the other three (because these have no default data specified in the map).

The operator updated the display, as follows, and pressed ENTER:

```
DEPT    DOLLARS      WEEK
*876    HABY
```

The program executed 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, firstly, that none of the three input data fields has a blank selector code, and secondly, 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 will send the message text to the terminal. In this case, the ADS had the following values, before the MSPUT:

MSG_SEL	MSG	DCODE_SEL	DCODE	EXP_SEL	EXP	WEEK_SEL	WEEK
1	Message text	1	*876	1	HABY	Blank	Blank

The program executed an ASREAD after the MSPUT, putting this display on the screen:

```
WEEK NUMBER MISSING
DEPT    DOLLARS      WEEK
*876    HABY
```

The operator then updated the display as follows:

```
WEEK NUMBER MISSING
DEPT    DOLLARS      WEEK
*876    HABY         22
```

In the program, control returned to the MSGET at the top of the loop, resulting in the ADS being updated 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 found that all the selectors in the input fields were set, but that the department code and expenditure were invalid. It therefore set the message selector to a 1 character again, and put the text of another message into the message field. After the reject and ASREAD, the screen looked like this:

```
ERROR(S) IN DEPT, DOLLARS
DEPT    DOLLARS      WEEK
*876    HABY         22
```

The operator corrected the input as follows:

```
ERROR(S) IN DEPT, DOLLARS
DEPT    DOLLARS      WEEK
HABY    876          22
```

After the MSGET, the ADS this time had the following data in it:

```
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 had processed the input data, it redisplayed the page for the operator 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 changed 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 looked like this:

```
DEPT    DOLLARS      WEEK
HABY    876          22
```

The operator amended the screen to:

```
DEPT    DOLLARS     WEEK
HABY    1234        23
```

After an MSGET, the ADS will be 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 had a selector character of 3, meaning that it contains old data, there was no need for the program to check it.

Alarm and keyboard locking

Effects of maps

GDDM-IMD allows you to specify for each map whether the alarm is to be sounded when it is sent to the terminal, and whether the keyboard is to be locked or freed. When the keyboard is locked, the terminal operator has to press RESET before the terminal will accept any more input.

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 will be sounded; and similarly if keyboard locking has been specified. Otherwise, no action will be taken.

The GDDM-IMD defaults are that GDDM should sound the alarm and lock the keyboard only after a reject. If these defaults apply to all maps on the page, the keyboard will be locked and the alarm sounded if any of the maps is updated by a reject-type MSPUT.

Other considerations

The FSALRM call (see "Sample alphanumerics program" on page 83) 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 21. Device support" on page 367) 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 terminal operator

The 3270 display unit allows 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 former type of field is called **protected**, and the latter **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 operator, 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 GDDM-IMD's map editor. Like selector adjuncts, they appear to the application as extra elements in the ADS.

Base attribute adjuncts consist of two elements, each one byte long. There is an example in "Base attribute adjuncts" on page 283. The second byte indicates what the program-defined 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 is used to define several types of attribute, not just the one concerned with protecting the field from operator input. The other types are listed in "Base attribute adjuncts" on page 283.

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 "Base attribute adjuncts."

The write-type MSPUT call then sets each field's base attributes according to the first byte of the adjunct, as follows:

A " " (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.

Base attribute adjuncts

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 adjunct must be set to the bit pattern representing the required 3270 attribute byte. The bit patterns are described in the *GDDM Base Programming Reference* manual.

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:

```
ADMUPIMC - for PL/I
ADMUCIMC - for COBOL
ADMUAIMC - for Assembler.
```

They are stored in the library called ADMLIB on VM/CMS, or in the sample library (GDDMSAM) on OS/TSO or CICS/VS. (ADMLIB and GDDMSAM also hold the PL/I declarations of the GDDM entry-points.) You will need to make the library available to your program, as outlined in "How to compile and run a GDDM Program under CMS" on page 11. 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 are 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 cursor

You can control the position of the cursor on output, and find out where the operator 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.

Output

You can often improve the usability of your displays by putting the cursor where the terminal operator 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 will use 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 will use a default position.

Dynamic Positioning: 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 will affect only a field with a cursor adjunct character of 1. It will have no effect if you have no cursor adjuncts or if they are all set to blank.

Static positioning: 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 position: 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 operator, 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.

Input

You discover in which field the operator left the cursor by inspecting the cursor adjuncts after an MSGET. This call sets the adjunct of the field that contains the cursor to 1, and all the other fields to " ". With this facility, you can create menus from which the terminal operator 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);
```

The 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 of -1.

In some applications, the terminal operator 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.

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 operator 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 adjunct to the length of data in the field, and GDDM pads the remainder of the field with nulls. If the operator modifies the field, then on input, GDDM sets the 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 operator 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 operator moves the cursor to the field and presses CURSR SEL.

To allow the operator to use the light pen (or CURSR SEL key) on a field, you must first give it the detectable 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 "Base attribute adjuncts" on page 283.

You must specify that GDDM-IMD is to create selector adjuncts for detectable fields, because GDDM uses this adjunct to indicate which fields have 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. These characters indicate the precise action that the terminal must take when a field is selected. A full description is given in *GDDM Base Programming Reference* manual. Here is a summary:

? Delayed detection. Nothing is transmitted to your program until the operator takes some other action that causes an interrupt, such as pressing ENTER, 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 ENTER.

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 277.

If an immediate light-pen field contains the cursor when it is selected, its cursor adjunct, if it has one, will be set to a 1 character.

Example of selection with cursor, light pen, and PF key

The program in Figure 83 on page 290 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 84 on page 291. 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 ENTER
- 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 allows 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 will use 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 operator used, by testing the first ASREAD parameter, ATTYPE. If the value 0 is found at /*F*/, meaning that ENTER 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 /*J*/, meaning that the light pen (or CURSR SEL key) was used, the selected field is discovered in the statements at /*K*/.

If ATTYPE has some value other than 0, 1, or 2, the operator must have pressed an invalid key, so the error flag is set at /*L*/.

```
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.             */

  CALL ASREAD(ATTYPE,                   /* Send mapped page to        */
              ATVAL,                    /* terminal and wait for      */
              COUNT);                   /* operator input.            */

  CALL MSGET(1,0,                       /* Get response into ADS.     */
             INITSEL_ASLENGTH,
             INITSEL);
```

Figure 83 (Part 1 of 2). Source code of MAPEX08

```
  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  */ /*J*/
           IF OPTION_SEL(1) = '1'       /* the selector adjuncts  */ /*K*/
             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.     */ /*L*/
  END;                                  /* End select group.      */
END;                                    /* End DO WHILE loop.     */

CALL FSTERM;                        /* Terminate GDDM.            */

%INCLUDE ADMUPINA;                  /* GDDM entry declarations    */
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINM;

END MAPEX08;
```

Figure 83 (Part 2 of 2). Source code of MAPEX08

```
         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 84. Field definitions of map used by MAPEX08

Alphanumeric input by PF key

The terminal operator's input is sometimes one of a number of predetermined character strings. GDDM provides a facility to save the operator 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*. 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 operator 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 operator 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 operator 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. More information is given in "Send output and await reply using call ASREAD" on page 13.

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 85 on page 293. It is the same program as the one shown in Figure 73 on page 253, except that after receiving correct input, it redisplays the map, instead of terminating.

To terminate, the operator 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 will put the character string END in this field. No field corresponding to USER_FIELD appears on the screen.

```
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('ACME01D6',           /* 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 85. Source code of MAPEX09

Highlighting, color, and symbol sets

Often, and particularly when the operator has made a mistake, it is desirable to highlight a field or change its color. You can create adjuncts that will allow 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 "Base attribute adjuncts" on page 283. 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 "Character attributes" on page 295), 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 terminal operator" on page 282. 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.

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 mapped field, 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 attribute to one of these values:

" " (Blank character) or hexadecimal '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. Notice, though, that in ASFCOL the parameter is a fullword integer, whereas the 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 "Symbol sets for alphanumerics" on page 221. 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 bytes of all three types of adjunct are set to 3.

Here is an example of using a color adjunct to draw the operator's attention to invalid input. At the same time, a message is put out. 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);
```

Character attributes

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 one 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 "Character attributes" on page 81. 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 will put the characters 1982 into the display. 19 will have 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. Then the following statement will assign 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:

A " " (a blank character)	Ignore the character-attribute string; in other words, leave the character attributes unchanged.
A 1 character	Take the character attributes from the ADS.
A 2 character	Use all-blank character-attribute characters. This will cause 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.

Input character attributes

Character attributes can be changed by the operator. To get information about these changes, you must first enable 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:

A 0 character	Supply information about the data. All the previous examples of MSGET calls in this guide use this value.
A 3 character	Supply information about highlighting character attributes.
A 4 character	Supply information about color character attributes.
A 5 character	Supply information about PS character attributes.

A type 3, 4, or 5 MSGET call will update the specified ADS with the current character attributes of all variable data characters. It will also set adjuncts in the same way as a type 0 MSGET. The meanings of selector adjunct codes on input are, for each type of attribute:

A " " (a blank character)	No character attributes of this type have been set for this field.
A 1 character	The field has character attributes of this type that were set by the operator in the last ASREAD.
A 3 character	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 will 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" on page 96). 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 will show the graphic area by filling it with % signs, or some other specified symbol.

Whatever the method of creation, GDDM will never allow more than one graphics field on a page.

There will always be 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 work station, 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 will 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 "Coordinate system" on page 19 and "Chapter 9. Hierarchy of GDDM concepts" on page 89.

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 86 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 88 on page 302. A typical display is shown in Figure 87 on page 301.

The program uses several calls, marked /*A*/, that refer to the graphics concepts of segment and picture space. The concepts are described in "Chapter 9. Hierarchy of GDDM concepts" on page 89.

```
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.           */

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. */
```

Figure 86 (Part 1 of 3). Source code of MAPEX11

```
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;
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;
```

Figure 86 (Part 2 of 3). Source code of MAPEX11

```
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 86 (Part 3 of 3). Source code of MAPEX11

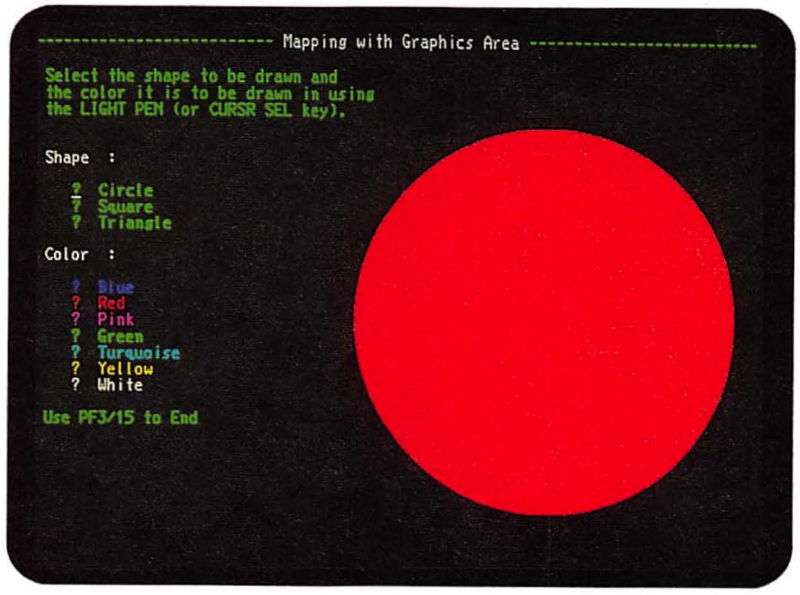


Figure 87. Typical display by MAPEX11

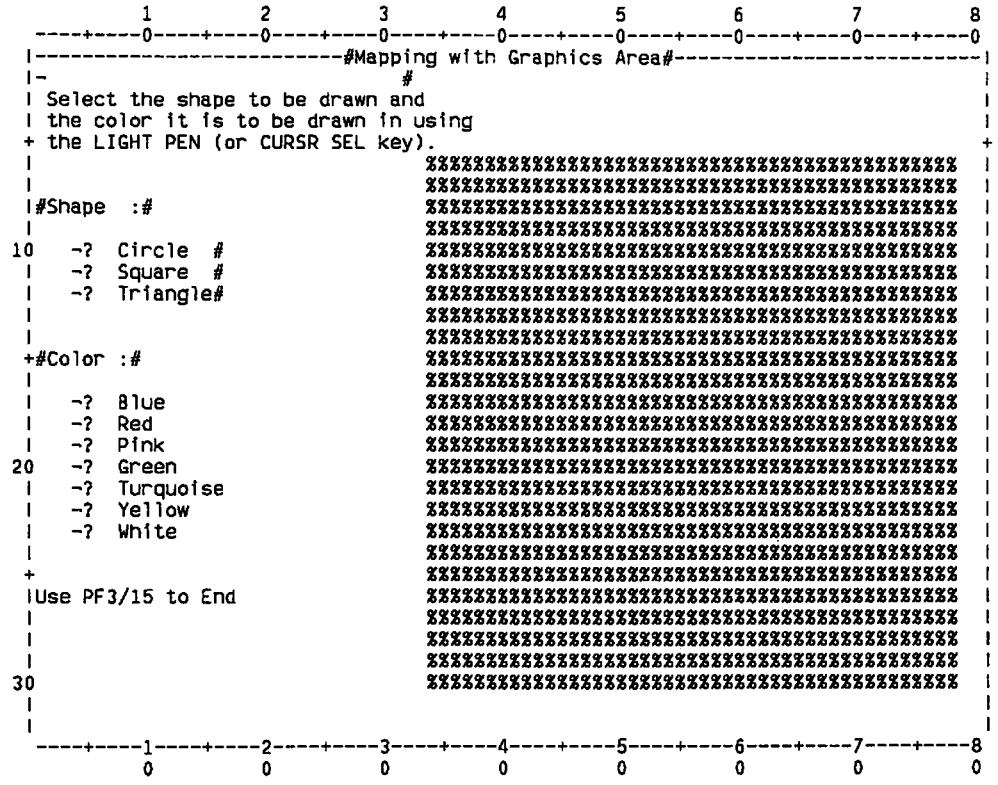


Figure 88. Field definitions of map used by MAPEX11

Part 4. Image processing

Chapter 19. Image basics

Introduction

This chapter introduces the basic concepts of GDDM image processing and illustrates them with small sections of sample code.

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.

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 a document and convert it 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 chapter and the next, the 3118 is the scanner assumed as the input device, and the 3193 the output device, except where stated otherwise.

Another device primarily 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" on page 363.

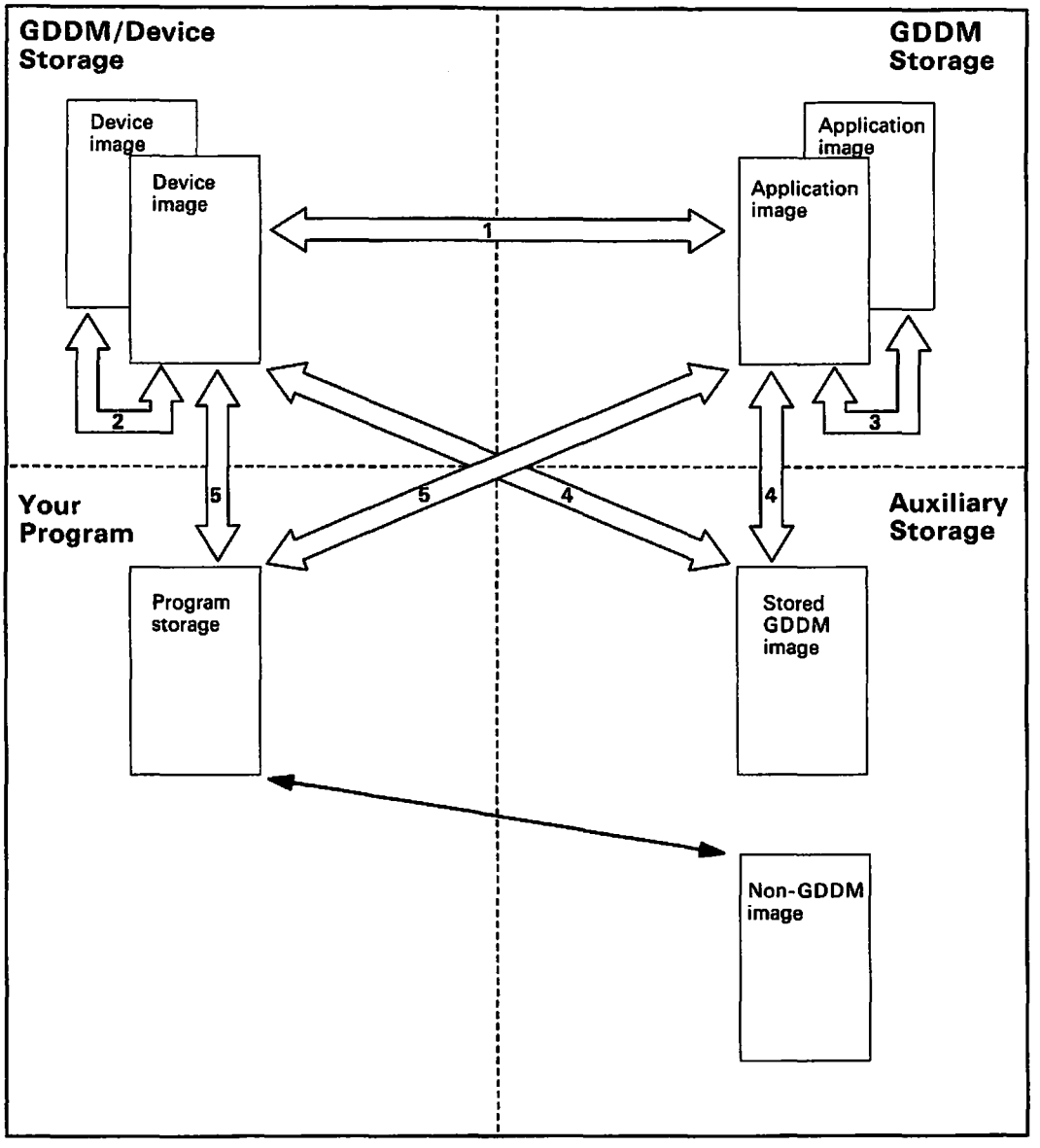


Figure 89. Image processing

The diagram in Figure 89 introduces **images** and **transfer operations**:

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."

There are three kinds of images in GDDM – device images, application images, and stored 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.

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 non-GDDM 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.

Transfer operations, as illustrated in Figure 89 on page 306, are the 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 chapter. The fifth operation is described in "Transferring images into and out of your program" on page 347 in the next chapter.

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 chapter describe projections and transforms in more detail.

How to scan, display, and save an image

The following sample 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 331, describes some of the scanner error conditions that can arise, and what you can do about them.

```
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 90. Simple image program – scan, display, and save an image

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.

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 will be 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 will be done by the 3193, and so will require 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 will receive the image data from the scan of the document. This scanner image will be 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. These are as follows, all in pixels per inch units:

  1. 120 horizontally and 120 vertically
  2. 240 horizontally and 120 vertically
  3. 240 horizontally and 240 vertically.

  The program uses 240/240, so the size of the image in pixels is always going to be 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. A value of 1 has the same effect. No other values are valid.

- The fifth parameter indicates whether the image has a defined resolution.

  Permitted values are:

  `0` Undefined resolution. This is used for raw image data, for example as in a computer-generated array of pixels, of no particular physical size.

  `1` Defined resolution, as specified by the next three parameters.

- The sixth parameter indicates the units of resolution of 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 will align the top edge of the paper with the top of the (empty) created scanner device image, and will center 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 will be 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 chapter.

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 is alphanumeric input consisting simply of ENTER or PF key use.

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:

```
CALL IMASAV(12,0,'MYIMAGE',16,'CLIENT SIGNATURE',1);
```

The parameters are as follows:

- The first parameter specifies the source image identifier.

- The second specifies the projection identifier, in this case 0 again for the identity projection.
- The third is the filename. Naming conventions vary according to the subsystem. They are explained in the *GDDM Base Programming Reference* manual. On VM, GDDM creates a file on your A-disk with the specified name as the file name, and a file type of ADMIMG. So the example would create a file called:

  ```
  MYIMAGE ADMIMG A1
  ```

- The fourth parameter specifies the length of the character string following.
- The next parameter gives a description of the file. The description is saved with the file, and 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.

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                    */
CALL IMARST(0,0,'MYIMAGE',30,DESCR);
```

The parameters are as follows:

- The first parameter is the identifier of the image into which the saved image is to be restored. A value of 0 restores the image to the display device. Alternatively, you can restore an image to an application image.
- The second parameter is the identifier of the projection. Once again, to simplify matters, the call applies the identity projection, but could have specified the identifier of any existing projection.
- The third parameter specifies the name of the file to be restored. The same remarks apply as for the equivalent parameter of the IMASAV call, described above.
- The fourth parameter gives a count of description characters to be used.
- The fifth is the variable name, DESCR, into which GDDM returns the description of the image.

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 chapter.

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 would be 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, say, 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 91. A projection can contain more than one transform. This is illustrated in Figure 92 on page 315.

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**.

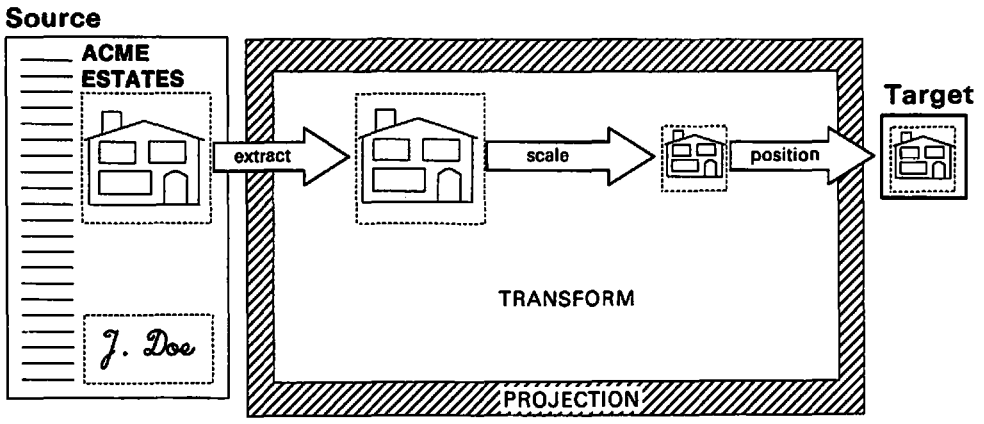


Figure 91. Projection containing a transform

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 "Converting gray-scale images to binary data" on page 332.)

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.

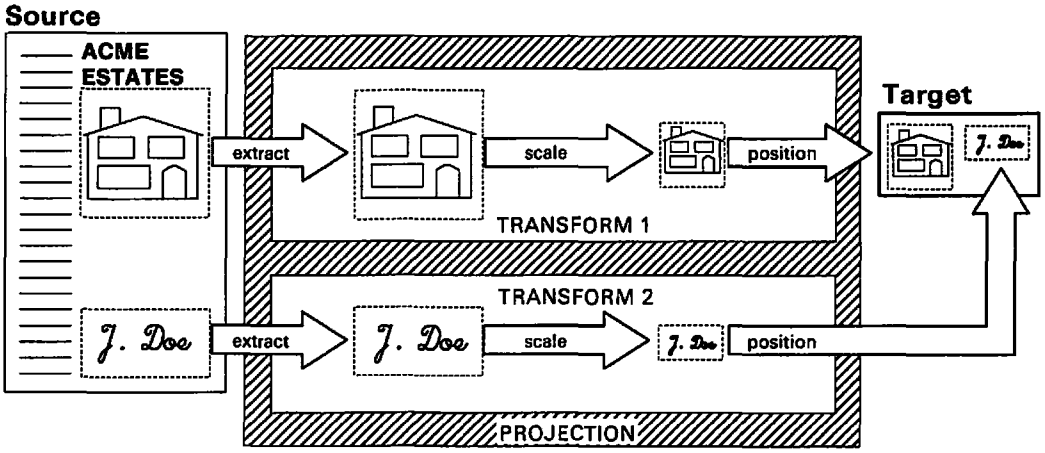


Figure 92. Projection containing two transforms

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 91 on page 314. 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 will restore and use 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);

     /* 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           */
```

The calls perform the following tasks.

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 will 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 326 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 that will 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 coordinate type of 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 allows you to specify the sub-image boundary in pixel coordinates: Here is an example of its use:

```
CALL IMREX(24,0,499,20,249);
```

The parameter list is as follows:

- The first parameter, 24, is the projection identifier.
- The next two parameters, 0 and 499, are the left edge and right edge of the required image, in pixel coordinates. (0 is the left hand edge of the source image).
- The last two parameters, 20 and 249, are the top edge and 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 will be 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.

Positioning an extracted image in the target image using call IMRPLR

You can use the IMRPLR call to position the result of your image processing within the target image space. The parameters are as follows:

- The first parameter, 15, is the projection identifier.
- The next parameter, 0, specifies coordinate type:

  0 Inches, as used here
  1 Meters
  2 Fractional.

- The next two parameters are the required offsets in the specified coordinate type. In the example, 0.0,0.0 places the extracted image in the top-left-hand corner of the target image.

- The last parameter specifies the mixing mode – the mode for mixing the pixels of the transformed image into those of the target image.

  0 is the default value, the same as **1**, which specifies overpaint mode.

  2 specifies merge mode, in which a "logical OR" operation is performed on the extracted image pixels and the target image pixels.

  Other settings for this parameter are fully described in the *GDDM Base Programming Reference, Volume 1.*

Instead of IMRPLR, you can use IMRPL to position the extracted image in the target image. IMRPL defines the position in pixel coordinates:

```
CALL IMRPL(15,0,0,0);    /* Position image in pixel coordinates */
```

The parameter list is as follows:

- The first parameter is again the projection identifier.

- The next two parameters define where the top left corner of the transformed image is to go in the target image. 0,0 simply aligns the top left corners of the transformed image and the target image.

  This position, together with the horizontal and vertical size of the transformed image, defines a rectangle within the target image.

- The last parameter, 0, is mixing mode, as for IMRPLR.

The IMRPLR call ends a transform. Either this call, or an IMRPL call, must always be used to complete the transform. That is, it is **mandatory** for a transform to contain one or other of these calls. No other transform call is mandatory. Until a transform has been completed with one of these calls, it 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 325.

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 Programming Reference* manual. 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 program example in this chapter contained 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.

The following example 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 315. 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 331 for how you can check for those conditions.

Here is the second example program:

```
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)   */

                                 /* 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;
```

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 GDDM calls, for example ASRFMT, or by mapped alphanumerics, whichever is preferred. See the appropriate alphanumerics chapter of this manual for examples and details of these calls.

The rest of the code in IMPROG2 assumes that the ANR1 routine has created the necessary alphanumerics fields using only the top two rows of the screen. The code from /*B*/ through /*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 the next chapter – see "Querying image devices" on page 331.

The ISFLD call, used at /*C*/, is also discussed in the next chapter – see "Combining an image with text or graphics" on page 356.

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 326, 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 310) – 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 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 chapters of this manual 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 will 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 Programming Reference, Volume 1.*

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 93 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 will then be different effects according to whether " black" or "white" pixels are deleted when they are adjacent.

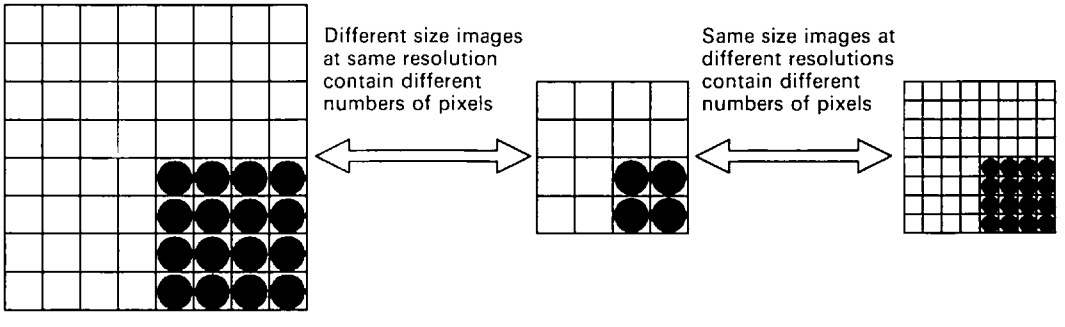


Figure 93. Resolution conversion

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, that are further defined in the *GDDM Base Programming Reference, Volume 1*:

  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 "Converting gray-scale images to binary data" on page 332 for calls that allow 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 will not then be 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 will 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:

```
CALL IMARES(51,0,300,180,3);
```

- The first parameter, 51, is the image identifier.

  -1 can be coded to specify the image scanner, in which case you can only specify scanner-supported resolutions in the third and fourth parameters (see below). You can use the ISQRES call to query the scanner-supported resolutions. See "Querying image-related device characteristics" on page 335.

- The second parameter, 0, specifies the unit of measure for the next two parameters:

  0 Inches
  1 Meters.

- The third parameter, 300, is the horizontal resolution in the chosen unit.

- The fourth parameter, 180, is the vertical resolution in the same unit.

- The fifth parameter, 3, specifies algorithm 3 for the resolution conversion process.

  The permitted values and their meanings are as follows:

  0 The default, same as 1
  1 Pixel replication
  2 Black pixel retention
  3 White pixel retention

  The meanings of the above values, the same as for the IMRRAL call described earlier (see "Defining the resolution conversion algorithm, using call IMRRAL" on page 324), are more fully defined in the *GDDM Base Programming Reference, Volume 1.*

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 will reflect 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 will replace the source image in application image 12.

This is true if the projection specifies overpaint mixing mode. Otherwise, the source and target images will be merged according to the mixing mode specified (in IMRPL or IMRPLR).

Chapter 20. Advanced image functions

This chapter covers the following topics:

- 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 and plotting an image
- Device variations.

Querying image devices

The scanner current status, such as 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 after such transfer operations.

The ISQSCA call and its parameter settings for various error states are fully covered in the *GDDM Base Programming Reference, Volume 1.*

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 last error record using call

FSQERR" on page 118. 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.

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 Programming Reference, Volume 1.*

Converting gray-scale images to binary data

The previous chapter 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 the following:

- Brightness conversion algorithm definition
- Contrast conversion algorithm definition
- Image type conversion algorithm definition.

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 – 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;
CALL IMRBRI(20,0,1,ARRAY);      /* Define brightness conversion */
```

The parameters are as follows:

- The first parameter, 20, is the projection identifier.
- The second, 0, specifies that the default algorithm is to be used. 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.
- The third parameter, 1, is a count value, specifying the number of elements in the array parameter that follows.
- The last parameter is 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. ARRAY(1) values specify the required brightness level as a number in the range −1 to +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 150. The new value, for the call above, will be:

```
150 + (-0.1 * 255) = 125
```

Negative values for the conversion factor reduce the gray-level, so darken the image, and positive values do the opposite.

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 following 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, will be

```
(( 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  */
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 following. 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, as 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. Now you will meet three more query calls, for determining the image data formats, compression algorithms, and resolution values supported by scanner, display, printer, or plotter devices.

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 this will incur a performance overhead in the host, as GDDM automatically converts to/from the format you specify.

```
DCL ARRAY(3) 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)  */
COUNT=ARRAY(1);
CALL ISQFOR(-1,COUNT,ARRAY);       /* Query 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 following array parameter.
- The last parameter is the array in which GDDM is to return the supported format codes, that can have the following values:

  1 Unformatted data
  2 3193 data-stream structures
  3 CPDS structures.

The image format(s) thus determined could 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 347.

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 this will incur a performance overhead in the host, as GDDM automatically converts to/from the compression you specify.

```
DCL ARRAY(4) 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) */
COUNT=ARRAY(1);
CALL ISQCOM(0,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 following array parameter.

- The third parameter is the array in which GDDM is to return the codes for the compressions supported.

  1 Uncompressed
  2 MMR
  3 4250
  4 3800.

Querying resolutions supported by a device, using call ISQRES

ISQRES 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 p.p.i. (pixels per inch) horizontally and 150 p.p.i. 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 (say 8 inches wide by 11 inches deep), using the IMACRT call met earlier, 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.

The following example program shows you how to do this:

```
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     */
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;
```

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 consists of 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 you 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 will respond to cursor move key use.)

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 first example 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 second example shows how a part-screen image field can be used.

Here is the first example:

```
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*/
```

```
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  *//*J*/
                BOX_TOP,BOX_BOTTOM);
                   END;             /* Trim image to box size     */
END LOOP;                           /* Cursor process loop        */

CALL ISENAB(2,0);                   /* Disable box cursor         *//*K*/

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;

END IMPROG5;
```

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 allows the terminal user to reposition and change the size of the box cursor, and press ENTER, 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 /*J*/, to trim the displayed image to the box size.

At /*K*/, 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.

Here is an extension of the above example to show you 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                     */
    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;
```

In the above program, at /*A*/, 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 allow 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 an **Yes** in the following table. **Yes – direct** signifies that the format/compression combination permits **direct transmission**, discussed under "Direct transmission" on page 355. Note that the values indicated by **Yes – direct** are those returned by the ISQFOR and ISQCOM calls.

	Unformatted	**3193 data stream**	**CPDS**
Uncompressed	**Yes**	**Yes – direct**	No
MMR	**Yes**	**Yes – direct**	**Yes**
4250	No	No	**Yes**
3800	No	No	**Yes**

Figure 94. Acceptable combinations of format and compression

For further information see the *GDDM Base Programming Reference* manual.

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 3800.

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 Figure 94 on page 347 give **direct transmission**, if the projection can be offloaded to the device. See "Controlling host offload by specifying image quality" on page 351 and "Direct transmission" on page 355 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, in the buffer named in the following parameter, to be transferred.
- 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 - say 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 Figure 94 on page 347 give **direct transmission**, if the projection can be offloaded to the device. See "Controlling host offload by specifying image quality" on page 351 and "Direct transmission" on page 355 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 the following, for example:

- Extract one or more rectangular sub-image(s) from the source image
- Apply a scaling factor to the extracted image
- Choose the scaling/resolution conversion algorithm
- Place 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 357.

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 non 3193-supported scaling factor, say 0.1 or 1.25, 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, say, 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 324.

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.

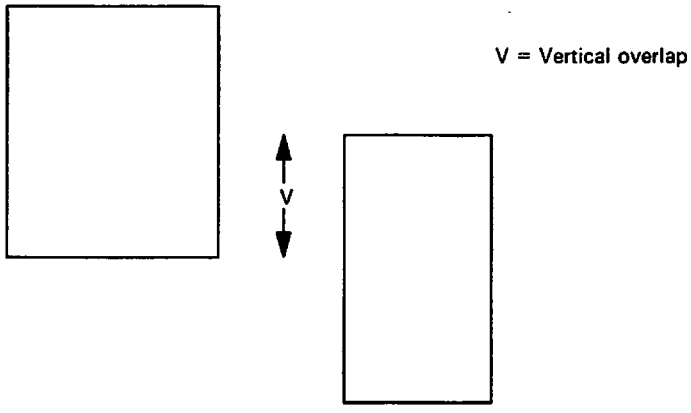


Figure 95. Vertical overlap

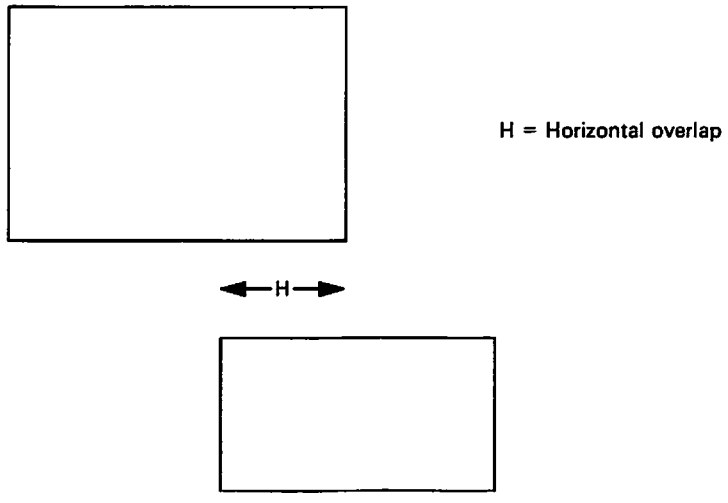


Figure 96. Horizontal overlap

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 the previous chapter. 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 352.

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, which 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 and 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 following.

- 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, with the meanings indicated:

  -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 following.

- 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, with the meanings indicated:

  -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 allows 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 351, the application can specify, by the ISFLD call, whether the image on the current

GDDM page must be read-write. 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 restrictions 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 doesn't.
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."

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 restrictions are met:

1. Offload of the target image (see "Controlling host offload by specifying image quality" on page 351).
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 9. Hierarchy of GDDM concepts" on page 89 introduced the concept of an image field similar to a graphics field. In addition, several sections in this chapter the previous one mention the use of image identifier 0 to refer to the image field on the current GDDM page, assuming this field to exist.

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 will display or print 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 for 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 and plotting images

The most convenient printer for image is the 4224, which is a desk-top device able to print graphics, alphanumerics, and image at a resolution of 144 pixels per inch.

Printing an image on a 4224 printer

4224 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 program that uses DSOPEN to establish the 4224 as the primary output device. You will meet the DSOPEN (open) and DSCLS (close) device calls in "Chapter 21. Device support" on page 367.

```
IMPROG8 : PROC OPTIONS(MAIN);

DCL PLIST(1) 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*/
CALL DSOPEN(4224,1,'X4224SS',0,PLIST,1,NLIST);                    /*B*/
CALL DSUSE(1,4224);                        /* 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(4224,0);               /* Close the printer          */
CALL FSTERM;

%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;

END IMPROG8;
```

At /*A*/, in the above program, *cuu* is a CMS device address, just as an example. This will need to be changed appropriately for your subsystem and installation.

At /*B*/, the value 4224 is used just by choice – any unused device identifier could be used. The values 0 and 1 should be avoided. The device token 'X4224SS' is an example. This will need to be changed appropriately for your installation.

Another topic introduced in the device support chapter is the use of **nicknames**. By using nickname statements in a file external to your program, you 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 "Nicknames" on page 378.

4224 as the secondary output device: "Chapter 22. Using printers" on page 395 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 FSCOPY call:

```
CALL FSCOPY;
```

to copy the displayed output to the printer or plotter.

Usually it is preferable to use the 4224 as the primary device, as described in the previous section. This is because secondary device use has the following implication.

Image and alphanumeric fields are copied in their correct row and column positions; the cells of the primary output device are mapped to the cells of the alternate device.

Because the primary and secondary device cells usually differ in aspect ratio, the aspect ratio of the image field will change, and so will the physical positioning of the image relative to text strings displayed as alphanumerics. However, the aspect ratio of the image will be preserved by resolution conversion.

Printing an image on 4250 or 3800-3

The following steps are required:

- Scan the image or restore it from auxiliary storage, to establish it as a GDDM application image.
- Transfer the image to one that has a size and resolution to suit the printer used. This is done either by maintaining the real size of the original image, or scaling it to fit the printer image size.

  Remember that in-place resolution conversion can be done using the IMARES call.
- Get the image from GDDM into your own application program storage, using a format of 3 or −3 as appropriate. Remember that, during this step, a projection can be applied.
- Write the image out to a file, in CPDS "page segment" format. It is then known as a PSEG file. See notes following the code below, on file-naming conventions.
- Use this file as a "secondary data stream" for input to other IBM program products, as described in the later chapter on use of printers (see "Composed-page printer as a family-4 primary device" on page 399).

The example program below shows you how to build the PSEG file, in this case from a restored GDDM image object called IMAGNAME. The program builds and writes out the necessary "Begin Page Segment" and "End Page Segment" CPDS orders at the start and end respectively.

Here is the code for printing a restored image on 4250, and the notes that follow explain the small changes required for printing on 3800-3.

```
IMPRG10: PROC OPTIONS(MAIN);
/* Declarations */
DCL SUBSTR BUILTIN;                  /* Substring builtin function */
DCL MIN BUILTIN;                     /* Minimum  builtin function  */
DCL APPL_ID FIXED BIN(31);           /* Restored image ID          */
DCL NEW_APPL_ID FIXED BIN(31);       /* Target image ID            */
DCL PROJ_ID FIXED BIN(31);           /* Projection ID              */
DCL DESCR CHAR(30);                  /* For image description      */
DCL TYPE FIXED BIN(31);              /* Program control:           */
                                     /*  set to 1 for 'real size'  */
                                     /*  set to 2 for scale-to-fit */
DCL HC_H_SIZE FLOAT DEC(6);          /* Hard copy image size in    */
DCL HC_V_SIZE FLOAT DEC(6);          /*  inches                    */
DCL HC_H_RES FLOAT DEC(6);           /* Printer resolution         */
DCL HC_V_RES FLOAT DEC(6);           /*  in pixels per inch        */
DCL HC_H_PIXELS FIXED BIN(31);       /* Hard copy image size in    */
DCL HC_V_PIXELS FIXED BIN(31);       /*  pixels                    */
DCL HC_H_POS FLOAT DEC(6);           /* Hard copy image position   */
DCL HC_V_POS FLOAT DEC(6);           /*  relative to top lh corner */
DCL SOURCE_H_SIZE FLOAT DEC(6);      /* Source image size in       */
DCL SOURCE_V_SIZE FLOAT DEC(6);      /*  inches                    */
DCL SOURCE_H_RES FLOAT DEC(6);       /* Source image resolution    */
DCL SOURCE_V_RES FLOAT DEC(6);       /*  in pixels per inch        */
DCL SOURCE_H_PIXELS FIXED BIN(31);/* Source image size in      */
DCL SOURCE_V_PIXELS FIXED BIN(31);/*  pixels                   */
DCL COMPN FIXED BIN(31);             /* 4250 or 3800 compression   */
DCL IM_TYPE FIXED BIN(31);           /* Image type                 */
DCL BI_LEVEL FIXED BIN(31) INIT(1);/* ,,   ,,  is BI-level     */
DCL SOURCE_RES FIXED BIN(31);        /* Defined/undefined resolution*/
DCL DEFINED FIXED BIN(31) INIT(1);/*Defined resolution          */
DCL INCHES FIXED BIN(31) INIT(0);/* Inch units                  */
DCL OVERPAINT FIXED BIN(31) INIT(0);/* Overpaint mix mode       */
DCL (H_RATIO,V_RATIO) FLOAT DEC(6);/* Size ratios               */
DCL SCALE FLOAT DEC(6);              /* Scale factor               */
DCL IMAGFIL FILE RECORD OUTPUT ENV(V RECSIZE(404));                 /*A*/
                          /*  recsize 4 bytes more than buffer*/
DCL DATABUF CHAR(400);               /* For image transfer to file */
DCL DATASUB CHAR(400) VARYING;       /* For short records          */
DCL DATALEN FIXED BIN(31);           /* Length of data in databuf  */
DCL BUFLEN FIXED BIN(31) INIT(400);/* Length of databuf          */
DCL 1 SF UNALIGNED,                  /* Structured field           *//*B*/
    2 SF_LENGTH FIXED BIN(15)        /* Structured field length,   */
                INIT(16),            /* always 16 here             */
                                     /* (does not include sf_cc)   */
    2 SF_RESERVED BIT(8)             /* Reserved - set to X'D3'    */
          INIT('11010011'B),
    2 SF_CODE BIT(16),               /* Set to required order code */
    2 SF_FLAG BIT(8)                 /* Flag - always X'00'        */
          INIT('00000000'B),
    2 SF_SEQUENCE FIXED BIN(15)      /* Sequence field, not used   */
          INIT(0),                   /* so set to 0                */
    2 SF_NAME CHAR(8)                /* Name field set to 00000000 */
          INIT('00000000');

DCL BPS BIT(16)                      /* Begin page segment order,  */
      INIT('1010100001011111'B); /* X'A85F'                    */
DCL EPS BIT(16)                      /* End page segment order,    */
      INIT('1010100101011111'B); /* X'A95F'                    */
```

```
CALL FSINIT;
/* Restore the image to be printed                                  */
CALL IMAGID(APPL_ID);              /* Get a new identifier          */
CALL IMARST(APPL_ID,0,'IMAGNAME',30,DESCR);
                                   /* restore GDDM image            */
/* Choose 'real size' (type=1) or 'scale-to-fit' (type=2)           */
TYPE=2;                            /* Set as you wish               *//*C*/
/* Choose size of image to be printed                               */
HC_H_SIZE= 5.0;                    /* 5 inches wide  (say)          */
HC_V_SIZE= 7.0;                    /* 7 inches deep (say)           */
/* Set up 4250 or 3800-3 parameters                                 */
COMPN=3;                           /* (Use COMPN=4 for 3800-3)      *//*D*/
HC_H_RES=600;                      /* P.P.I. resolution             *//*E*/
HC_V_RES=600;                      /*  for 3800-3 or 4250           */
/* Create a printer image to suit                                   */
HC_H_PIXELS=HC_H_SIZE * HC_H_RES;
HC_V_PIXELS=HC_V_SIZE * HC_V_RES;
CALL IMAGID(NEW_APPL_ID);          /* Get an identifier             */
CALL IMACRT(NEW_APPL_ID,           /* Create the new image          */
            HC_H_PIXELS,HC_V_PIXELS,BI_LEVEL,DEFINED,
            INCHES,HC_H_RES,HC_V_RES);
/* Negate the image                                                 *//*F*/
CALL IMPGID(PROJ_ID);              /* Get an identifier             */
CALL IMPCRT(PROJ_ID);              /* Create a projection           */
CALL IMRNEG(PROJ_ID);              /* Negate (invert) the image     */
CALL IMRPLR(PROJ_ID,INCHES,0,0,OVERPAINT);/* End of projection      */
CALL IMXFER(NEW_APPL_ID,NEW_APPL_ID,PROJ_ID);/* Apply the         */
                                   /*                 projection    */
CALL IMPDEL(PROJ_ID);              /* Delete the projection         */
/* We now have an all-white-pixels image matching the              */
/* 4250 or 3800-3 printer characteristics.                          */
/* Now we can process the original, restored image according to*/
/* whether we have chosen 'real size' or 'scale-to-fit'             */
/* processing.                                                      */
IF TYPE=1 THEN                     /* 'Real size' required          *//*G*/
  CALL IMXFER(APPL_ID,NEW_APPL_ID,0);
ELSE IF TYPE=2 THEN
  DO;                              /* 'Scale to fit' required       *//*H*/
    CALL IMAQRY(APPL_ID,
                SOURCE_H_PIXELS,SOURCE_V_PIXELS,
                IM_TYPE,SOURCE_RES,
                INCHES,SOURCE_H_RES,SOURCE_V_RES);
    SOURCE_H_SIZE=SOURCE_H_PIXELS/SOURCE_H_RES;
                                   /* Source image width - inches*/
    SOURCE_V_SIZE=SOURCE_V_PIXELS/SOURCE_V_RES;
                                   /* Source image depth - inches*/
    H_RATIO=HC_H_SIZE/SOURCE_H_SIZE;/* Size ratios of hard copy*/
    V_RATIO=HC_V_SIZE/SOURCE_V_SIZE;/* Image to source image   */
    SCALE=MIN(H_RATIO,V_RATIO);    /* Required scale factor         */
    CALL IMPGID(PROJ_ID);          /* Get an identifier             */
    CALL IMPCRT(PROJ_ID);          /* Create another projection     */
    CALL IMRSCL(PROJ_ID,SCALE,SCALE); /* Scale the image to fit*/
    /* Now position the image centrally                             */
    HC_H_POS=(HC_H_SIZE-(SCALE*SOURCE_H_SIZE))/2;
    HC_V_POS=(HC_V_SIZE-(SCALE*SOURCE_V_SIZE))/2;
    CALL IMRPLR(PROJ_ID,INCHES,HC_H_POS,HC_V_POS,OVERPAINT);
                                   /*End of projection definition*/
    CALL IMXFER(APPL_ID,NEW_APPL_ID,PROJ_ID);/*Apply projection*/
  END;                             /* Scale to fit                  */
  ELSE GO TO FINISH;               /* Exit for type not =1 or 2     */
/* We now have the required image within GDDM, and can access       */
/* it, convert it to 'page segment' file format, and write it       */
/* out as a page segment file                                       */

OPEN FILE(IMAGFIL);                /* Open the output file          */
SF_CODE=BPS;                       /* Set BPS order in SF           */
```

```
WRITE FILE(IMAGFIL) FROM(SF);     /* Write structured field      */

CALL IMAGTS(NEW_APPL_ID,0,-3,COMPN); /*  Negated, CPDS format, *//*J*/
                                  /*  chosen compression        */
DO I=1 BY 1 UNTIL(DATALEN=0);     /* Image get-write loop       */
  CALL IMAGT(NEW_APPL_ID,BUFLEN,DATABUF,DATALEN);
                                  /* Get image into application */
                                  /* data buffer                */
  IF DATALEN=BUFLEN               /* Full buffer                */
    THEN
     WRITE FILE(IMAGFIL) FROM(DATABUF);
    ELSE
     IF DATALEN>0 THEN
         DO;                      /* Buffer part full           */
     DATASUB=SUBSTR(DATABUF,1,DATALEN); /* Keep only the data   */
     WRITE FILE(IMAGFIL) FROM(DATASUB);
         END;                     /* Buffer part full           */
      ELSE;                       /* End of data (DATALEN=0)    */
                                  /*     write it out to file   */
END;                              /* Image get-write loop       */

CALL IMAGTE(NEW_APPL_ID);         /* Terminate get process      */

SF_CODE=EPS;                      /* Set EPS order in SF        */
WRITE FILE(IMAGFIL) FROM(SF);     /* Write structured field     */

CLOSE FILE(IMAGFIL);              /* Close the output file      */
FINISH:;
CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPINI;

END IMPRG10;
```

In the above program, at /*A*/, the file records are declared to be of variable length, up to a maximum value of 404 bytes to correspond with the 400 byte length of DATABUF. (Variable length records have a 4-byte record descriptor word before the data itself.)

You are recommended to use a buffer length of at least 400 bytes; depending on the image data format, GDDM may use up to this value. Buffers other than the last may be only part-filled.

At program run time, the declared file name IMAGFIL must be associated with an external file name in a subsystem-dependent way. For example, on VM/CMS, before running the program you must issue the CMS command:

```
filedef imagfil disk fn ft fm
```

where "fn" is a file name of your choice, "ft" is PSEG4250 for a 4250 PSEG file, or PSEG38PP for a 3800-3 PSEG file, and " fm" is file mode (A1 for example).

At /*B*/, the declared structure is required for Begin Page Segment (BPS) header and End Page Segment (EPS) trailer records. These are in accordance with CPDS (Composed Page Data Stream) formats. For further information see "Chapter 22. Using printers" on page 395. In particular see "Composed-page printer as a family-4 primary device" on page 399.

This format, as coded, is required for 4250. For 3800-3 only, the BPS and EPS records must be prefixed with a one-byte control character, value X'5A'. For

programming convenience this can be added as the first element of the data structure SF, merely by adding to the declaration at /*B*/ the following line:

```
2 SF_CC BIT(8) INIT('01011010'B),
```

and leaving the SF_LENGTH field unchanged. (This is because this byte is not strictly a part of the structured field).

For 3800-3 therefore, you should modify the declaration for SF in this way.

At /*C*/, scale-to-fit processing has been chosen, although you could, instead, select real size processing.

At /*D*/, COMPN value 3 selects 4250 compression. For 3800-3, change this variable to value 4.

At /*E*/, the 4250 resolution of 600 p.p.i. has been set. For 3800-3, resolution must be set to either 120 p.p.i. or 240 p.p.i..

At /*F*/, it is necessary to negate the newly created image because GDDM sets this to all black pixels, whereas we require all white pixels, to give a white background to the image.

At /*G*/, real size processing means that the identity projection must be used, as it is in the following IMXFER call.

At /*H*/, scale-to-fit processing begins. This is similar to that given in an earlier example. Note that this code requires that the source image (that is, the image that is restored) has defined resolution. You could extend the code to cater for the undefined resolution case, indicated by SOURCE_RES on return from the IMAQRY call, by deriving scale factors based on the pixels size ratios rather than on actual size ratios as done here, and omitting the central positioning code; zero offsets could be used instead, in the IMRPLR call.

At /*J*/, negated (inverted) CPDS format must be used, for the image to print correctly, that is, as originally saved. This is true for both 4250 and 3800-3 output.

Instead of creating a page segment (PSEG4250 or PSEG38PP) file as in the example above, your program can build a " document" (LIST4250 or LIST38PP) file. This requires additional structured fields to be embedded, as the record sequence is a little more complex.

For reasons for doing this and for more information, see the CDPF and PSF publications listed under "Books from related libraries" on page v.

Device variations

The following notes are on differences in the use of devices other than the principal image processing devices covered in the preceding sections of this chapter and the previous one.

IBM 3179-G, 3270-PC including /G and /GX, 3279, 3290, 5080, 5550 displays

(That is, on all the GDDM device family-1 displays that support graphics except the 3193)

Image transforms and output are done entirely by emulation, with some associated performance overheads. Because GDDM uses graphics (GSIMG) for this, there must be no graphics on the same page as the image output.

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 given if any attempt is made to enable it.

Image input to GDDM: For display terminals other than the 3193, 3117 or 3118 scanner attachment is not possible. 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 347 for admissible formats). The image can then be transferred to the GDDM page in the usual way.

IBM 3268 and 3287 printers

(These are both device family-1 printers)

Image transforms and output are done by emulation, with some associated performance overheads. Because GDDM uses graphics (GSIMG) for this, there must be no graphics on the same page as the image output.

Plotters

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.

Part 5. Device support, printing, plotting, and windowing

Chapter 21. Device support

Many programs can be written without a knowledge of device support. But you may need to understand it to perform any of the following tasks:

- Defining a device's characteristics to GDDM
- Sending output to a device other than the invoking device
- Printing or plotting copies of the main display output
- Communicating with more than one device
- Saving data streams suitable for a device that has not yet been installed
- Specifying device-dependent or subsystem-dependent processing options.

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**.

A real device, or a virtual device, or both, can be opened with a DSOPEN call. For the default device, GDDM issues an internal DSOPEN. If you want to use a device other than the invoking terminal, you must either explicitly open it using a DSOPEN call, or modify the internal DSOPEN with a nickname statement (see "Nicknames" on page 378).

Before any output is created, a device must be made current using a DSUSE call. All later alphanumerics and graphics statements will apply to that device until a new device is made current. The scheme is the same as that for pages: you may open several devices, but at any time only one of them is current. For the user console, GDDM issues an internal DSUSE.

Opening a device using call DSOPEN

DSOPEN is GDDM's most complex call, with many device-dependent and subsystem-dependent parameters. For a full description, refer to the *GDDM Base Programming Reference* manual. It is possible to remove most of the complexities from your program if you or your installation sets up nickname files. You can then use simplified DSOPEN calls, as described in "Simple DSOPEN using nicknames" on page 370.

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)=2;                  /* Option code 2 denotes an */
                                    /* Output-only option group */
PROCOPT_LIST(2)=1;                  /* Option value 1 requests  */
                                    /* Device to be output-only */

NAME_LIST(1)='062';                 /* CMS device address       */

/******************/
/* OPEN DEVICE 11 */
/******************/
/*   DEVICE-ID  FAMILY  TOKEN  PROCESSING-OPTIONS  PHYSICAL-DEVICE*/
CALL DSOPEN(11,    1,  'L79A3',   2,PROCOPT_LIST,     1,NAME_LIST);
```

This is the meaning of the seven parameters:

`11` The **device identifier**. All future references to the device will use this identifier.

`1` The **device family code**, specifying the type of device.

These are the permitted settings:

1. 3270 family device (display or printer), 8775 display, 4224 printer, plotters, 5080 (with processing option).
2. Queued printer device
3. System printer device
4. High-resolution image files for composed-page printers.

`L79A3` The **device-token**, telling GDDM the properties of the device. The token L79A3 states that the device is a local 3279 (Model 3) with a screen size of 32 by 80. There are three sets of device tokens supplied with GDDM, called **device definition tables.** They are listed in the *GDDM Base Programming Reference* manual. (ADMLSYS1 is the table for family-1 and family-2 devices, ADMLSYS3 is the table for family-3 devices, and ADMLSYS4 is the table for family-4 devices. See *GDDM Installation and System Management* for details of their formats and how to change them.)

A token parameter of * tells GDDM to get the information itself - usually by querying the device. This is the most common setting of the parameter.

`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 of fullwords containing the **processing options list.** This may contain one or more **option groups** - each a request for a particular processing option. Some of these options are dependent on the device family, others are valid only in a particular subsystem.

The present example contains just one option group - an output-only option group. The first fullword in a group identifies the option type - here 2 indicates "output-only 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 the newly-opened device be placed in output-only mode. A setting of 0 would have set "not-output-only" mode, the default.

You can place several option groups in the processing options list, each with an option code in its first word. A brief list of the possible option groups is given in "Device processing options" on page 370.

`1` The number of 8-byte names in the seventh and last parameter.

`NAME_LIST` An array of 8-byte names, identifying which physical device should be opened. The naming scheme used in the **name list** is dependent on the device 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 printers under TSO.
2. Family-2, -3, and -4 devices under CMS, in which case second and third elements can be used to specify a file type and a file mode.
3. Auxiliary family-1 devices (usually plotters) under any subsystem, which have two part names, the first of which is the name of the family-1 terminal to which they are attached.

More information is given in "Chapter 22. Using printers" on page 395 and "Chapter 23. Using plotters" on page 421.

Here (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/VS the single name may be set to an "LTERM name." On all the other subsystems (CICS/VS, and TSO), it is not permitted to 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 "Using a dummy device" on page 376.)

In the example, the call to DSOPEN made known to GDDM the family-1 device with a subsystem name of 062. It told GDDM that the device is a local 3279 (Model 3) with a screen size of 32 by 80, and it assigned an identifier of 11 to the device for future reference. It requested that the device be processed in output-only mode.

Device processing options

The processing options available on the DSOPEN call are described in the *GDDM Base Programming Reference* manual.

Without going into details, some examples are listed here:

1. You can choose the CMS PA1/PA2 protocol and the method of attention handling.
2. On CMS you can spool and tag print files, and automatically invoke the GDDM Print Utility.
3. On TSO you can select the CLEAR/PA1 and reshow protocols.
4. On CICS you can select pseudoconversational mode. See "Pseudoconversational programming under CICS" on page 391 for details.
5. For a queued or alternate printer device, you can specify a set of print control parameters (see "Chapter 22. Using printers" on page 395).
6. For a family-4 printer, you can specify a set of print control parameters (see "Composed-page printer as a family-4 primary device" on page 399).
7. For plotters, you can specify a set of physical plotting parameters (see "Processing options for plotters" on page 422).
8. For 3270-PC/G and /GX work stations, you can enable local operations, specify how graphics data is to be stored, and change the default symbol sets for graphics text (see "Processing options for the 3270-PC/G and /GX" on page 388).
9. For family-1 and -2 printers and plotters, you can tell GDDM to add the user identifier, date, and time to the output.
10. You can associate a 5080 with a 3270.
11. You can request that the device operates in output-only mode. Control will return to the program immediately after an ASREAD.
12. You can specify that the keyboard should always be unlocked, even after an FSFRCE.

Simple DSOPEN using nicknames

The more complex parameters of DSOPEN can be specified in nickname files, rather than on the DSOPEN call. It is possible for an installation to set up system-wide nickname files containing standard device definitions. Or, instead, under CMS and TSO you can set up your own.

A typical DSOPEN would then specify simply a device-id, the device family, and a device name, with default or null values for the other parameters:

```
DCL PROCOPT_LIST(1) FIXED BIN(31);
DCL NAME_LIST(1) CHAR(8);
NAME_LIST(1) = 'COLPRT3';

    /* 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 further definition of a family-2 device with the name COLPRT3. The files would typically supply a device token and a set of processing options. They can also supply a different device name, and change the device family. For more information, see "Nicknames" on page 378.

If your installation has system-wide nicknames, you will need to get information about the available device definitions from a system programmer in your installation.

Specifying device usage using call DSUSE

It is possible to open (by using DSOPEN) several different devices. This action will have no effect on the program until GDDM is informed that the program requires to **use** a particular device.

DSUSE indicates that a particular device should be used for future output. DSUSE also performs an implicit DSDROP (see next section). 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 will be sent to an **alternate device**, usually a printer. The means by which these copies are made will be addressed in the next chapter.

  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 will have one current primary device and (optionally) one current alternate device.

Discontinuing use of a device, using call 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 will be just as you left it - you can even leave a segment open, if you choose.

How to use more than one primary device

When you do not issue an explicit DSOPEN for any device, but start drawing graphics immediately, GDDM issues an internal DSOPEN for the default device - the user console. It then requests, by means of an internal DSUSE, that this be treated as the primary device.

GDDM uses the device identifier 0 for the default primary device. You should therefore be very careful about using this identifier yourself. It may be used only if you are sure that you will at no stage use the default device. 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 want to send some output elsewhere, you must issue a DSDROP to device 0 before using the new terminal.

As explained in "Chapter 9. Hierarchy of GDDM concepts" on page 89, the device is at the top of the hierarchy. All the other graphics objects belong to the device. You cannot create some graphics and then decide to which terminal to send it. When you issue a graphics or alphanumerics command of any sort (creating a page, defining an alphanumeric field, or opening a segment, for example), it will be associated with the device current at that time. If there is none, GDDM will assume the default device, namely the user console, and associate the new graphics with it.

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. A better way might be to make one device an alternate one, in which case a simple copy call may do what is required.

Example program: Using two primary devices

This section contains an example program 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. There are several points to note about the program:

- **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 higher 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, say, 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. There is no ambiguity.

- **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 will fill the viewport. If the subroutine is reexecuted under a larger window as defined at /*E*/, the output will fill correspondingly less of the viewport. In the last section of the program, the grapefruit is redrawn in the central quarter of the viewport.

```
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 will therefore cause an        */
                          /* internal DSOPEN of the user console   */
                          /* (and a matching DSUSE), not device    */
                          /* 061. Alphanumerics and graphics will  */
                          /* be 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 will be       */ /*A*/
                                /* assigned to 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*/

CALL GRAPE_FRUIT;               /* Reexecute subroutine to draw   */
                                /* a picture of a grapefruit      */
CALL ASREAD(TYPE,MOD,COUNT)     /* Send output to device 15       */

/******************/
/* DROP 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;
```

Closing a device using call DSCLS

When a device will not be reused, 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 will retain 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 the values shown below.

- For a family-1 display device, the options have the following meaning:

  0 Erase the screen. (If in CICS pseudoconversational mode, unlock the keyboard, and save any device data that has changed.)

1 Do not erase the screen. (If in CICS pseudoconversational mode, unlock the keyboard, and save any device data that has changed.)

2 Erase the screen, and unlock the keyboard. (If in CICS pseudoconversational mode, erase the device data.)

3 Do not erase the screen, but unlock the keyboard. (If in CICS pseudoconversational mode, erase the device data.)

- For a non-family-1 printer, 0 cancels the print file, − 1 requests printing.

The effect of these options is subsystem-dependent. See the *GDDM Base Programming Reference* manual for more details.

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 will be sent to such a device. It is merely a convenience that would allow, say, the development of a 3279 program at a 3277 (non-PS) screen.

You must specify a device-token for a dummy device. You may recall that a device-token of * asks GDDM to determine the device characteristics itself (usually by querying the device). 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.

Sample program: Using a dummy device to create a stored picture

Dummy devices are frequently used in combination with FSSAVE (see "Saving current page contents using call FSSAVE" on page 16). This enables pictures to be saved on auxiliary storage that are suitable for later transmission **to a particular type of device.**

This example, which could be run without a user console in batch mode, illustrates the situation. It will create two saved representations of the architect's design − one for later display on a 3279, one for later display on a 3278.

```
SAVE2: PROC OPTIONS(MAIN);
CALL FSINIT;

NAME_LIST(1)=' ';    /* Set blank name to indicate dummy device */

/*************************/
/* OPEN DUMMY DEVICE WITH */
/* 3279 CHARACTERISTICS   */
/*************************/
/*    DEVICE-ID  FAMILY  DEV_TOKEN  OPTIONS        WHICH DEVICE */
CALL DSOPEN(11,    1,    'L79A3',  0,PROCOPT_LIST,  1,NAME_LIST );
```

```
/*************************/
/* MAKE DUMMY DEVICE THE  */
/* CURRENT DEVICE         */
/*************************/
CALL DSUSE(1,11);                       /* Use dummy device with */
                                        /* 3279 characteristics  */
CALL A_DRAWING; /* Call subroutine to create architect's drawing*/
                /*      (on the default page of device 11)      */

CALL FSSAVE('DIAG3279');    /* Save diagram for later FSSHOR-ing */
                            /* on a 3279 display screen          */

/*************************/
/* OPEN DUMMY DEVICE WITH */
/* 3278 CHARACTERISTICS   */
/*************************/
/*    DEVICE-ID  FAMILY  DEV_TOKEN   OPTIONS       WHICH DEVICE */
CALL DSOPEN(12,    1,    'R78A4',  0,PROCOPT_LIST,  1,NAME_LIST );

/*************************/
/* DROP DUMMY 3279       */
/*************************/
CALL DSDROP(1,11);                  /* Must drop one primary device */
                                    /* before using another         */
/*************************/
/* MAKE DUMMY 3278       */
/* THE CURRENT DEVICE    */
/*************************/
CALL DSUSE(1,12);                   /* Use dummy device with        */
                                    /* 43-line 3278 characteristics */

CALL A_DRAWING; /* Call subroutine to create architect's drawing*/
                /*      (on the default page of device 12)      */
CALL FSSAVE('DIAG3278');    /* Save diagram for later FSSHOR-ing */
                            /* on a 43-line 3278 display screen  */

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;
```

Nicknames

You can code a **nickname** in the name list parameter of a DSOPEN call, and thereby identify a device definition supplied elsewhere. The "elsewhere" is typically either the GDDM external defaults module or a defaults file.

Nicknames have two principal uses:

- Simplifying DSOPEN calls. An installation can supply a set of standard device definitions by means of the external defaults module, and so relieve application programmers of the complexities of DSOPEN.

- Deferring the definition of a device until execution time. A major application of this is to allow each run of a program to use a different device **without having to change and recompile the program.** For instance, you can change the destination of a program's output from a printer to a plotter simply by changing a nickname file.

Nicknames are part of GDDM's generalized default and exit facilities, which are described in full in the *GDDM Base Programming Reference* manual.

To understand how to use nicknames, consider this DSOPEN call:

```
NAME_LIST(1) = 'COLPRT3';
   /* DEVICE-ID  FAMILY  TOKEN  PROCESSING-OPTIONS  DEVICE-NAME */
CALL DSOPEN(4,     2,     '*',   0,PROCOPT_LIST,    1,NAME_LIST);
```

When the DSOPEN is executed, GDDM searches for matching nickname statements, that is, ones that refer to the same device family and device name. If no match is found, the DSOPEN applies unchanged. Here is a nickname statement that matches this DSOPEN:

```
ADMMNICK FAM=2,NAME=COLPRT3,
         TOFAM=,TONAME=,
         DEVTOK=L87,
         PROCOPT=((INVKOPUV,YES))
```

The FAM and NAME parameters specify the device family and device name to which the statement applies. This nickname statement will add 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 L87.

- The processing option, supplied in the PROCOPT parameter, is INVKOPUV; it is equivalent to option group 1004 in DSOPEN, and it automatically invokes the function of the CMS version of the GDDM print utility, ADMOPUV.

  A list of all the nickname processing options and their DSOPEN option group equivalents is given in the *GDDM Base Programming Reference* manual.

The TOFAM and TONAME parameters are used to change the device family and device name. No values have been given, so the family and name remain as specified in the DSOPEN call.

Syntax

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:

```
ADMMNICK 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:

```
ADMMNICK 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:

```
ADMMNICK NAME=(OUT1,ADMPRINT,A),TONAME=(PRTFIL,ADMPRINT,G)
```

You can omit name-parts, as described in "Multipart names" on page 380.

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
```

Unspecified or zero device family

If you specify FAM=0 on a nickname statement, the statement will match any family. This statement will change the device name whatever the family specified on the DSOPEN:

```
ADMMNICK FAM=0,NAME=061,TONAME=063
```

Omitting the family value or omitting the FAM parameter completely are equivalent to specifying FAM=0. These statements are both equivalent to the previous one:

```
ADMMNICK FAM=,NAME=061,TONAME=063
ADMMNICK NAME=061,TONAME=063
```

Unspecified, null, *, or blank device name

If you specify a null device name on a nickname statement by coding "NAME=," then the statement will match any name. This statement will change the device family whatever the name specified on the DSOPEN:

```
ADMMNICK NAME=,FAM=2,TOFAM=1
```

Omitting the NAME parameter is equivalent to specifying a null name.

A name * will match only an explicit *. Similarly, a blank device name will match only a blank. You can specify a blank device name like this:

```
ADMMNICK FAM=0,NAME=(),TONAME=063
```

This statement would direct output originally intended for any device with a blank name, that is, any dummy device, to a real device.

Multipart names

In a multipart name specification, parts can be omitted or specified as *. Here is a rather extreme case:

```
ADMMNICK NAME=(,*,C),TONAME=063
```

This will match a name with a blank as the first part, * as the second, and C as the third.

The general rules about matching multipart names are similar to those for a single-part name. For instance,

```
 ADMMNICK NAME=PRINTER2,...
or
 ADMMNICK NAME=(PRINTER2,*),...

will match:

          PRINTER2
or        PRINTER2 *
or        PRINTER2 * *

but not:

          PRINTER2 ADMPRINT
nor       PRINTER2 ADMPRINT A1
```

Conversely, this statement would match any of them:

```
ADMMNICK NAME=(PRINTER2,?,?),...
```

Relative priorities of nickname statements and DSOPEN call

The processing options and device token in a nickname statement are, in effect, default values. Any explicit value on the DSOPEN call will override them. (* is not an explicit device token.)

The reverse is true of the device family and name: the values in the TOFAM and TONAME parameters of a nickname statement override the values in the DSOPEN call.

Defaults module and defaults file

The most common ways of passing nicknames to GDDM are by the external defaults module, which applies to all users in the system, or (under CMS and TSO only) by an external defaults file that applies only to those users with access to it. Information about these and other methods of passing nickname statements to GDDM is given in "How to pass nickname statements to GDDM" on page 384.

A defaults file has priority over the defaults module. Typically, a system programmer would set up the defaults module, and application programmers would set up their own files, where necessary, to supplement or override the module. Under CMS, you can put a defaults file on any disk you have accessed, such as your A-disk. Under TSO it can be a data set that you have allocated.

How to use nickname statements

This section contains some more examples of nickname statements. For examples of directing output to plotters, see "Using nicknames to direct and control the output" on page 433. For examples of spooling output to RSCS (the Remote Spooling Communication System of CMS), see "Chapter 22. Using printers" on page 395.

Simplifying DSOPEN

Setting up default processing options and device tokens: This statement sets up a default device token and processing option for a family-2 printer named A3287:

```
ADMMNICK FAM=2,NAME=A3287,DEVTOK=L87,
         PROCOPT=((PRINTCTL,0,1,72,0,0,0,100,0))
```

The device token and processing option can be overridden by a DSOPEN call (except that a device token of * on the DSOPEN would not override the one specified in the nickname statement).

Omitting the NAME parameter would make the defaults applicable to all family-2 devices.

Identifying devices by name alone: The next statement shows how to relieve application programmers of the need to specify a real device family on DSOPEN. Output for any printer named as PRT1, whatever family was specified, will be directed to a family-4 printer named P3800M3. The statement also supplies a device token, and processing options could be added.

```
ADMMNICK NAME=PRT1,TOFAM=4,TONAME=P3800M3,DEVTOK=IMG240
```

It is not necessary for a real device with the name PRT1 to exist. By using nickname statements such as this one, a set of unique names for all devices of all families can be set up. The DSOPEN calls can then identify the devices by these names alone. A dummy device family value (0, say) could even be coded in all DSOPEN calls in this format:

```
DCL NAME_LIST(1) CHAR(8);
NAME_LIST(1) = 'PRT1';

   /* DEVICE-ID  FAMILY  TOKEN  PROCESSING-OPTIONS  DEVICE-NAME */
CALL DSOPEN(4,     0,     '*',   0,PROCOPT_LIST,    1,NAME_LIST);
```

Defining devices at execution time

Changing device name: This statement simply converts a program to use a different family-1 device from the one specified in a DSOPEN call:

```
ADMMNICK FAM=1,NAME=061,TONAME=063
```

The processing options and device token are left unchanged. The statement would, for instance, make the "Example program: Using two primary devices" on page 372 use the device at address 063 rather than the one at 061.

Changing device family: This statement converts a program from using a family-1 device to family-2:

```
ADMMNICK FAM=1,NAME=061,TOFAM=2,TONAME=PRT3287,DEVTOK=L87
```

The device token of 'L87' will override a device token of *, but not an explicit one. Processing options can be added to this nickname statement, if required.

Adding processing options: This statement adds the local mode processing option to the definition of all family-1 devices:

```
ADMMNICK FAM=1,PROCOPT=((LCLMODE,YES))
```

Processing options that do not apply to the device being opened are ignored. For instance, the LCLMODE option in the above example applies only to 3270-PC/G and /GX work stations. It will be ignored for any other terminals.

If any option specified in this way conflicts with another specified in the DSOPEN call, the DSOPEN value will take precedence.

Two options are mergeable – PRINTCTL and STAGE2ID. They accept variable numbers of parameters, and those on a nickname statement will be merged with those on the DSOPEN. If a parameter is specified in both, the DSOPEN value takes precedence.

Adding a device token: A real (non-*) device token on a DSOPEN cannot be changed by a nickname statement, but * can be replaced by a real token. This statement changes a device token of * to an explicit one, namely IMG85, for all family-4 devices:

```
ADMMNICK FAM=4,DEVTOK=IMG85
```

Multiple nickname statements

Device token and processing options: These statements show how GDDM accumulates information from more than one nickname statement:

```
ADMMNICK NAME=PRT1,FAM=4,PROCOPT=((HRISPILL,YES))
ADMMNICK NAME=PRT1,FAM=4,PROCOPT=((HRISWATH,10),
                                  (HRIFORMT,BITMAP))
ADMMNICK NAME=PRT1,FAM=4,DEVTOK=IMG85
ADMMNICK NAME=PRT1,FAM=4,DEVTOK=IMG240
ADMMNICK NAME=PRT1,FAM=4,DEVTOK=*,
                         PROCOPT=((HRISWATH,20))
```

GDDM accumulates information from the DEVTOK and PROCOPT parameters as it scans the statements. Where there is more than one DEVTOK parameter, the latest value applies. Where two PROCOPT values conflict, the later one applies. The above statements are hence equivalent to the single statement:

```
ADMMNICK NAME=PRT1,FAM=4,DEVTOK=*,
                         PROCOPT=((HRISPILL,YES),
                                  (HRIFORMT,BITMAP),
                                  (HRISWATH,20))
```

At the end of nickname processing, any processing options that do not apply to the real device are ignored. The device token that results from nickname processing will only override a device token of * on the DSOPEN. A non-* device token cannot be overridden.

TOFAM and TONAME parameters: GDDM accumulates TOFAM and TONAME values during the scan. These statements:

```
ADMMNICK NAME=DEV99,FAM=1,TOFAM=4,TONAME=P3800M3,DEVTOK=IMG240,
                                                 PROCOPT=((HRISPILL,YES))
ADMMNICK NAME=DEV99,FAM=1,TOFAM=3,TONAME=SYSPRT2,DEVTOK=S1403N8,
ADMMNICK NAME=DEV99,FAM=1,TOFAM=2,TONAME=A3287P5,DEVTOK=L87,
                                               PROCOPT=((PRINTCTL,1,8))
```

are equivalent to:

```
ADMMNICK NAME=DEV99,FAM=1,TOFAM=2,TONAME=A3287P5,DEVTOK=L87,
                             PROCOPT=((HRISPILL,YES),(PRINTCTL,1,8))
```

At the end of the scan, GDDM updates the DSOPEN parameter list with the latest TOFAM and TONAME values (in additiion 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.

REPLACE and APPEND parameters: These parameters control the effects of second and subsequent matching nickname statements found during a single scan. REPLACE causes all earlier statements found during the current scan to be ignored. APPEND is the default and causes the parameters to be merged with those of any earlier matching statements. If a statement with APPEND specified or defaulted has a parameter value that conflicts with an earlier value within the current scan, then the later one overrides the earlier.

For example:

```
ADMMNICK NAME=D1,FAM=1,TOFAM=2,DEVTOK=TOK2,
                         PROCOPT=((LOADDSYM,YES),(LCLMODE,YES))
ADMMNICK NAME=D1,FAM=1,TONAME=D2,DEVTOK=*,
                         PROCOPT=((LCLMODE,NO),(SEGSTORE,NO)),
                         REPLACE
are equivalent to:

ADMMNICK NAME=D1,FAM=1,TONAME=D2,DEVTOK=*,
                         PROCOPT=((LCLMODE,NO),(SEGSTORE,NO))

whereas:

ADMMNICK NAME=D1,FAM=1,TOFAM=2,DEVTOK=TOK2,
                         PROCOPT=((LOADDSYM,YES),(LCLMODE,YES))
ADMMNICK NAME=D1,FAM=1,TONAME=D2,DEVTOK=*,
                         PROCOPT=((LCLMODE,NO),(SEGSTORE,NO)),
                         APPEND

are equivalent to:

ADMMNICK NAME=D1,FAM=1,TONAME=D2,TOFAM=2,DEVTOK=*,
                         PROCOPT=((LOADDSYM,YES),
                         (LCLMODE,NO),(SEGSTORE,NO))
```

Rescanning: A change of device family or name causes a rescan of all nickname statements not already matched. During the rescan, GDDM searches for nickname statements that match the new device 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.

How to pass nickname statements to GDDM

There are four ways of passing nicknames to GDDM. In order of increasing priority, they are:

1. The GDDM external defaults module. A file of nickname statements can be assembled using a set of GDDM-supplied macros. For more information, see the *GDDM Installation and System Management*, and the *GDDM Base Programming Reference* manuals.

2. A user-created external defaults file.

   All you need to do is put nickname statements like the ones shown in this chapter into a suitable file.

   Under CMS, the file must be on a currently accessed disk and have the filename PROFILE and filetype ADMDEFS.

   Under TSO the file must have a ddname of ADMDEFS.

   The format of the file is not critical, but fixed length 80-byte records are recommended. (Full details are given in the *GDDM Base Programming Reference* manual, together with a full description of the statement's syntax, including information about labeling and adding comments.)

3. The control block called the SPIB that is passed to GDDM by the system-programmer interface initialization call, SPINIT. For more information, see the *GDDM Base Programming Reference* manual.

4. Calls in the application program. You can supply one nickname statement at a time by use of the ESSUDS call. For example:

   ```
   CALL ESSUDS(56,
        'NICKNAME FAM=1,TOFAM=4,NAME=A4250,PROCOPT=((COLORMAS,100))');
   ```

   The first parameter is the length of the nickname statement.

GDDM scans first the external defaults module, then the external defaults file if there is one, then, if the system programmer interface is being used, the SPIB. In each case, the statements are scanned in the order in which they are stored.

Finally, GDDM scans the statements derived from any application program calls executed before the DSOPEN, in the order in which they were executed.

The examples in this chapter are in **source format**. There is an alternative **encoded format**. Supplying the nickname statements in encoded format saves processing resources. You can create them in encoded format, or, for the defaults module, create them in source format and assemble them into encoded format. If they are to be assembled, the source statements must conform to Assembler language syntax rules.

The defaults module must always be supplied in encoded format. Normally it is created in source format and then assembled. The defaults file must always be supplied in source format. A set of statements intended to form a defaults module can be created as a source-format defaults file for test purposes, and then assembled into a defaults module.

Nickname statements passed by the SPIB must be in encoded format.

To supply encoded format statements by means of application program calls, you must use the ESEUDS call instead of ESSUDS. They must be created directly in encoded format. The performance gain is likely to be significant only if you make extensive use of nicknames. You can pass several nickname statements in one ESEUDS call.

Full information about the encoded format is given in the *GDDM Base Programming Reference* manual.

Processing options for operator windows

Some examples of windowing programs are given in "Operator windows" on page 467. You can make operator windows available to the user, using a processing option, either on a DSOPEN call:

```
DECLARE NAME(1) CHARACTER(8);
DECLARE PROCOPT_LIST(2) FIXED BINARY(31):

PROCOPT_LIST(1) = 24;              /* Operator window mode      */
PROCOPT_LIST(2) = 1;               /* 1 = yes, 0 = no (default) */

CALL DSOPEN(0,1,'*',   2,PROCOPT_LIST,   0,NAME);
```

or in a nickname statement:

```
ADMMNICK FAM=1,PROCOPT=((WINDOW,YES))
```

GDDM does not allow real partitions when operator windows are used. Instead, GDDM gives you emulated partitions.

When you use a DSOPEN, with its own list of processing options, to open a virtual device, the following options are ignored, and taken instead from the processing options for the real device:

AUNLOCK	Always unlock keyboard mode.
CMSATTN	CMS attention handling.
CMSINTRP	CMS PA1/PA2 protocol.
CTLKEY	User control key
CTLMODE	User control
PSCNVCTL	CICS pseudoconversational mode
TSOINTRP	OS/TSO CLEAR/PA1 protocol
TSORESHW	TSO reshow protocol

Processing options for user control

User control from the terminal user's point of view is described in the *GDDM Guide for Users*. This section describes the controls that a program can use.

You can make user control available to the user, either on a DSOPEN call for the terminal:

```
DECLARE NAME(1) CHARACTER(8);
DECLARE PROCOPT_LIST(2) FIXED BINARY(31):

PROCOPT_LIST(1) = 28;              /* User control              */
PROCOPT_LIST(2) = 1;               /* 1 = yes, 2 = no           */

CALL DSOPEN(0,1,'*',   2,PROCOPT_LIST,   0,NAME);
```

or in a nickname statement:

```
ADMMNICK FAM=1,PROCOPT=((CTLMODE,YES))
```

(CTLMODE,YES) is the default for non-partitioned devices.

Whenever your program is waiting for input, as a result of ASREAD, GSREAD, MSREAD, or WSIO, the terminal user can invoke user control. The user must exit from user control before responding to the read.

The PA3 key is the default key for invoking user control. If the terminal does not have a PA3 key, you can specify, for example, the PF1 key, using the control-key processing option, either on a DSOPEN call for the terminal:

```
DECLARE NAME(1) CHARACTER(8);
DECLARE PROCOPT_LIST(3) FIXED BINARY(31):

PROCOPT_LIST(1) = 29;      /* Control-mode key processing option */
PROCOPT_LIST(2) = 1;       /* 1 = PF key, 4 = PA key             */
PROCOPT_LIST(3) = 1;       /* PF1 key                            */

CALL DSOPEN(0,1,'*',   3,PROCOPT_LIST,   0,NAME);
```

or in a nickname statement:

```
ADMMNICK FAM=1,PROCOPT=((CTLKEY,1,1))
```

As an further example, the following nickname statement specifies the PA2 key:

```
ADMMNICK FAM=1,PROCOPT=((CTLKEY,4,2))
```

The first number (1 and 4 in the above examples) specifies whether it is a PF or PA key. It uses the same codes as the second parameter of ASREAD. The second number (1 and 2 in the examples) is the key number.

On 5550-family multistations, if you do not make user control available, or if you make user control available but you specify a key other than PA3 for user control, the PA3 key can be used to refresh the screen. If you make user control available but do not specify a key other than PA3 for user control, PA3 activates user control, and the screen will be refreshed on exit, at least. PA3 is not passed to your application under any circumstance.

You can use the CTLFAST processing option to specify that your program should use fast path mode for User Control functions that require pointings (MOVE, SIZE, POINT, CENTER, ZOOM-IN, ZOOM-OUT). Here is the nickname statement:

```
ADMMNICK FAM=1,PROCOPT=((CTLFAST,YES))
```

When (CTLFAST,YES) is specified and a User Control function that requires pointing is selected by a PF key, it is assumed that the user has already positioned the cursor at the first pointing.

The default is (CTLFAST,NO).

You can use two other processing options, CTLPRINT and CTLSAVE, to specify whether print/plot and save functions are available in user control. Here are the nickname statements:

```
ADMMNICK FAM=1,PROCOPT=((CTLSAVE,YES))
ADMMNICK FAM=1,PROCOPT=((CTLPRINT,YES))
```

By default, on all terminals, printing and plotting are allowed; saving is subsystem-dependent. See the *GDDM Base Programming Reference* manual for details.

Putting the terminal into user control, using call DSCMF

Instead of allowing the terminal user to press a key to enter user control, you can use the DSCMF call in your program:

```
CALL DSCMF(1);                /* Invoke user control at read call */
```

Following the above call, any GDDM read call (ASREAD, GSREAD, MSREAD, or WSIO) puts the terminal into user control. The user still has to exit from user control before responding to the read call.

The call has no effect if user control has been disabled by processing option.

A value of 0 in DSCMF's single parameter resets your program so that read calls no longer automatically invoke user control. Your program can query the current state using the DSQCMF call. DSQCMF returns a 1 value for on, and a 0 for off.

Processing options for the 3270-PC/G and /GX

Retained and non-retained modes

The work station can either retain graphics orders in its storage after the picture has been generated or the picture can be generated without the orders being retained. You can control the mode of operation with a processing option, either on a DSOPEN call or a nickname statement. More information is given in "Retained and non-retained modes" on page 508.

Panning and zooming

Panning and zooming are available on all graphics displays, as part of the user control function. By default, new picture definitions are sent to the terminals when required.

For 3270-PC/G and /GX work stations, you can specify that, rather than sending new picture definitions, GDDM should perform panning and zooming by sending new transforms to the work station. The transforms are then processed at the work station, rather than by GDDM.

To enable this local function, you must set the local mode processing option on, either on a DSOPEN statement for the terminal:

```
DECLARE NAME(1) CHARACTER(8);
DECLARE PROCOPT_LIST(2) FIXED BINARY(31);

PROCOPT_LIST(1) = 21; /*Local mode pan and zoom processing option*/
PROCOPT_LIST(2) = 1;  /* 1 = on; 0 = off (default)                */

CALL DSOPEN(0,1,'*',   2,PROCOPT_LIST,   0,NAME);
```

or in a nickname statement:

```
 ADMMNICK FAM=1,PROCOPT=((LCLMODE,YES))
```

Enabling local mode results in less host processing and shorter data streams for panning and zooming. Transforms for zooming operations are only sent to the device when the zooming is witih certain device- and page-related limits (usually allowing the picture to be zoomed by a factor of up to 16). Above these limits,

panning and zooming is performed in the host, and continues to be until the user selects "reset" from the graphics menu. Although using the LCLMODE procopt reduces the host processing overheads while panning and zooming take place, it may introduce overheads during the normal output processing. See *GDDM Performance Guide* for details.

Default symbol sets for graphics text

As explained in "Differences on IBM 3270-PC/G and /GX work stations" on page 233, the default symbol sets for mode-2 and -3 graphics text are the image and vector sets held by the work station. You can specify that GDDM symbol sets should be used instead, by means of an option. This can be coded either on a DSOPEN call:

```
DECLARE NAME(1) CHARACTER(8);
DECLARE PROCOPT_LIST(2) FIXED BINARY(31);

PROCOPT_LIST(1) = 19;/* Default symbol set processing option    */
PROCOPT_LIST(2) = 1; /* 0 = hardware set (default); 1 = GDDM set*/

CALL DSOPEN(1,1,'*',   2,PROCOPT_LIST,   0,NAME);
```

or in a nickname statement:

```
ADMMNICK FAM=1,PROCOPT=((LOADDSYM,YES))
```

Processing options for 3270-PC/G and /GX, 3179-G, and 5550 family displays

There are two modes of picture update on 3270-PC/G and /GX, 3179-G, and 5550 displays:

- **Full draw** mode gives an accurate picture. This is the default.
- **Draft draw** mode, gives you the option of avoiding redraws when graphics data is changed. Faster updates can be obtained in many cases, because GDDM only updates the changed segments. There is a cost in possible drawing inaccuracies, where primitives overlap either before or after the update, because GDDM degrades the color-mixing. Overlapped sections might therefore be missing or in the wrong colors. Also, when overlapping partitions are deleted, the data in the partition that was overlapped may have an inaccuracy.

The modes can be set in user control, or in your program. Your program could, for example, set draft-draw mode while the user is editing the picture, but set full-draw mode, with no consequent inaccuracies, for displaying or plotting the final version. In your program, it can be coded either on a DSOPEN call:

```
DECLARE NAME(1) CHARACTER(8);
DECLARE PROCOPT_LIST(2) FIXED BINARY(31);

PROCOPT_LIST(1) = 26;/* Update mode processing option          */
PROCOPT_LIST(2) = 1; /* 1 = draft draw, 0 = full draw (default) */

CALL DSOPEN(1,1,'*',   2,PROCOPT_LIST,   0,NAME);
```

or in a nickname statement:

```
ADMMNICK FAM=1,PROCOPT=((FASTUPD,YES))
```

or with the call FSUPDM. Here is an example:

```
CALL FSUPDM(1);                    /* Set update mode to draft draw */
```

Each setting of the update mode overrides any previous setting, regardless of the method used to set it.

Processing option for the 5080 graphics system

The 5080 is supported under TSO and VM, but not under CICS or IMS. The 5080 is used as a family-1 device. Under VM, it must be attached to your virtual machine as a virtual device. Under TSO, the 5080 must be online to the MVS system.

The 5080 uses two logically separate screens for graphics and alphanumerics.

If the 5080 has the 3270 feature, the 5080 physical screen is used for both logically separate screens, but only one can be seen at a time.

If the 5080 does not have the 3270 feature, the 5080 is associated with a physically separate 3270 terminal (screen and keyboard). In this case, the 5080 displays graphics and the 3270 displays alphanumerics.

The 5080 is either associated with the 3270 feature or the physically separate 3270 by a processing option. The 3270 is the primary device.

With the 3270 feature, the 5080 keyboard is used in either graphics or alphanumerics mode; without this feature, both the 5080 and 3270 keyboards are used. The protocol for the use of the 5080 and 3270 keyboards together is described in *GDDM Guide for Users*.

These are the steps required to run GDDM applications on a particular 5081 display unit:

1. Log on to your VM or TSO subsystem using either the 5080 in 3270 mode or a separate 3270 terminal. For VM systems, also IPL CMS. Under VM, ensure that the 5081 is attached to your virtual machine. You may need to ask the system operator to attach it. Then, either using the 5081 in 3270 mode, or the separate 3270 terminal:

2. To define a ddname for the 5081:

   Under VM, enter the following command:

   ```
   FILEDEF ddname GRAF cuu
   ```

   Under TSO, enter the following:

   ```
   GABALOC FILE(ddname) UNIT(cuu)
   ```

   In both cases, cuu is the address of the 5085 control unit for the 5081 display unit

3. The processing option for the 5080 must either be set in your program:

   or using a nickname:

   ```
   ADMMNICK FAM=1 PROCOPT=((SPECDEV,IBM5080,ddname))
   ```

4. Invoke your GDDM application program.

Querying the device

You can query the effects of a DSOPEN call with a DSQDEV call. The information returned includes the device family, the device name, the device token, and the processing options actually used for the DSOPEN, that is, the values after any nickname processing.

You can query the characteristics of the primary device (but not an alternate device) with an FSQURY call. This returns the device family, and some information about the device hardware, such as cell size, pixel density, and number of programmed symbol stores. It does not return DSOPEN information – the device name, the device token, or the processing options.

For 5080, it returns information about the 5080 and the associated 3270.

For full details of the DSQDEV and FSQURY calls, see the *GDDM Base Programming Reference* manual.

Other device calls

- DSRNIT - reinitializes a device. Any usage of the device is discontinued, resources (for example symbol sets) allocated to it are released, and the device is returned to the state it was in when first opened.
- DSQUID(device-id) - returns a valid, currently unused device identifier.

For full details of the above calls, see the *GDDM Base Programming Reference* manual.

Pseudoconversational programming under CICS

Applications that are initiated from a terminal and consist of a continuous dialogue with the terminal user (for example, a system editor such as ISPF on TSO, or XEDIT on VM) 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. **Converse** with the terminal user (typically, display the first panel and wait for input).
4. Do while finish not requested.
   a. Process the terminal input.
   b. **Converse** with the terminal user (typically, display the requested panel and wait for input).
5. End.
6. End the application.

You can see from the above summary that the program conducts a conversation with the terminal user.

GDDM provides three calls to perform the conversation:

ASREAD Output the current page and await alphanumeric input
GSREAD Output the current page and await graphics input
MSREAD Output the current map and await mapped input.

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 for the duration of the wait for terminal input. If the application is widely used, this could have adverse implications on the 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 above application is as follows:

1. Start the application.
2. Perform initialization.
3. **Send** data to the terminal user (typically, display the first panel).
4. End

   Return to CICS requesting a following transaction.
5. Start transaction requested by previous return.
6. **Receive** the terminal input.
7. Process the input.
8. **Send** the data to the terminal user (typically, display next panel).
9. End

   Return to CICS either requesting a following transaction or not.
10. Repeat steps 5 through 9 while a following transaction has been requested.

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 Primer* manual.

GDDM provides the ability for a GDDM application to use CICS pseudoconversational programming using the following calls:

DSOPEN The PSCNVCTL processing option specifies to GDDM whether pseudoconversational mode is in use. PSCNVCTL,START specifies that the use is starting. PSCNVCTL,CONTINUE specifies that it is continuing.

The procopt tells GDDM to obtain device query information from the device itself, or from previously saved data in temporary storage.

ASREAD When the application is in CONTINUE pseudoconversational mode, the first ASREAD issued by the application causes no output to be sent to the terminal, and only the input part of the ASREAD is performed.

DSCLS If pseudoconversational mode is in use, DSCLS unlocks the device keyboard. In addition, two options are provided that are used by pseudoconversational applications to end the pseudoconversational mode.

These options tell GDDM to either save device query information in temporary storage, or to erase the temporary storage queue.

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 that there is no pseudoconversational support for the GSREAD and MSREAD calls.

The following list illustrates the order of the GDDM calls for pseudoconversational mode in a mapping application:

- On the initial invocation of the transaction:
  1. FSINIT
  2. DSOPEN (Start pseudoconversational mode)
  3. Create mapped alphanumeric data for the first screen using MSPCRT, MSDFLD,and MSPUT
  4. Create any graphics output
  5. FSFRCE
  6. DSCLS (Option 1 - do not erase the screen)
  7. FSTERM
  8. EXEC CICS RETURN TRANSID(Tname) COMMAREA(Carea)

     'Carea' should contain any information required to continue the transaction processing; in particular, it should contain the Application Data Structures used for output of the mapped data.
- On subsequent invocations of the transaction:
  1. FSINIT
  2. DSOPEN (Continue pseudoconversational Mode)

3. Create mapped alphanumeric data for 'previous' screen, using the identical set of MSPCRT, MSDFLD, and MSPUT calls used the last time, and also using the same Application Data Structures (as saved in 'Carea')
4. **Do not** issue any graphics calls
5. ASREAD
6. Process mapped input using MSGET as normal
7. Create mapped alphanumeric data for the next screen using MSPCRT, MSDFLD,and MSPUT
8. Create any graphics output
9. FSFRCE
10. DSCLS (Option 1 - do not erase the screen)
11. FSTERM
12. EXEC CICS RETURN TRANSID(Tname) COMMAREA(Carea) LENGTH(Clen)

    'Carea' should contain any information required to continue the transaction processing; in particular, it should contain the ADSs used for output of the Mapped data.

- Use DSCLS with option 2 or 3 to terminate the pseudoconversation.

There is an example of the application outlined above in "Example 5. CICS pseudoconversational example" on page 499.

Chapter 22. Using printers

Overview

There are four distinct methods of sending output to a printer. They are summarized in Figure 97.

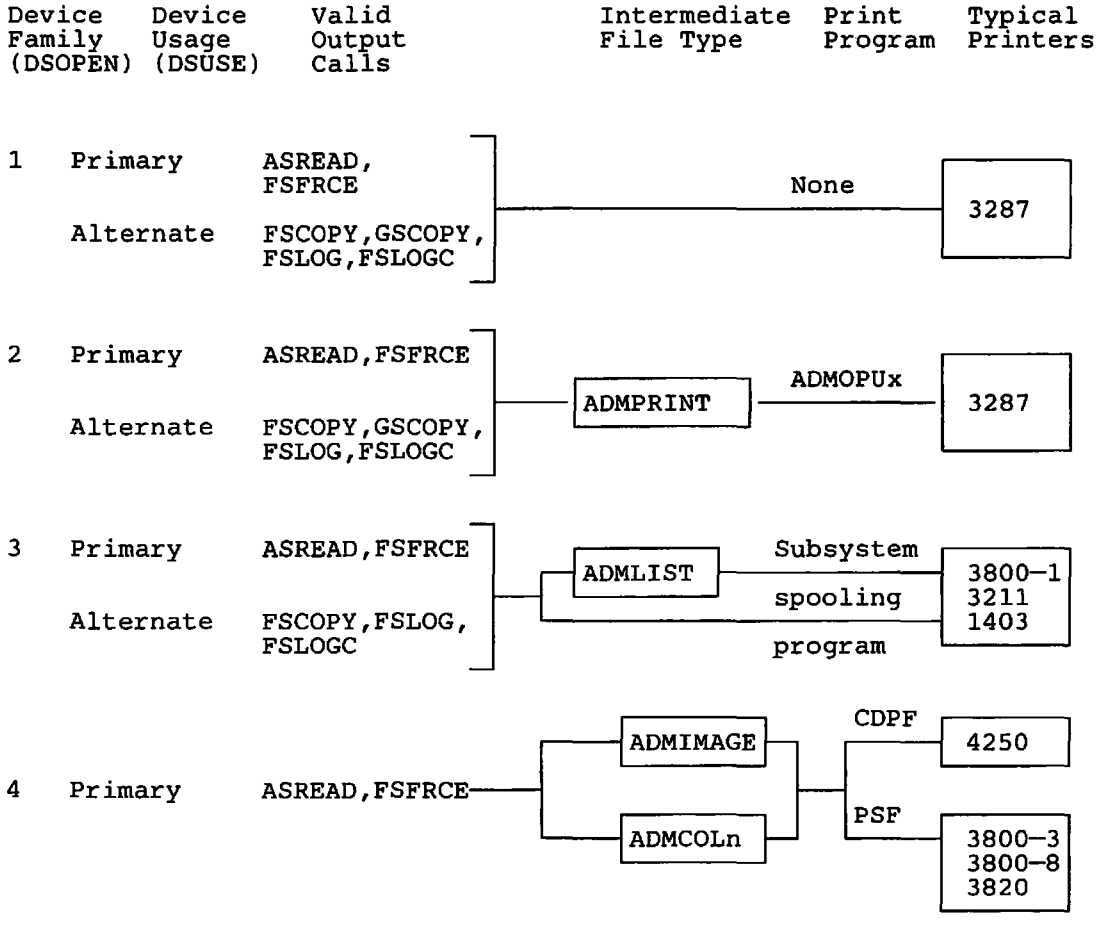


Figure 97. Overview of GDDM support for printers

The device-family parameter of the DSOPEN call defines the method, as follows:

- Family-1 means a printer attached to your program. This is possible only under the CMS, CICS/VS, and IMS/VS subsystems.
- Family-2 means a **queued printer**, that is, a printer belonging to the subsystem and shared between several users. 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 407.

- Family-3 means a **system printer** driven by a subsystem spooling program. Your program's output can be stored on a GDDM-created file before being passed to the spooling program.

- Family-4 means a **composed-page printer** (sometimes called a **high-resolution printer**) capable of printing both text and graphics. The quality of their output is generally high enough to be used for the masters from which publications are printed. Composed-page printers are not attached directly to application programs. The data from a program is saved in a GDDM-created file which is then printed using one of the IBM utility programs: Composed Document Printing Facility (CDPF), Print Services Facility (PSF), or Document Composition Facility (DCF). Before printing, a number of text and graphics files may be combined in a page composition process.

A family-1, -2, or -3 printer can be a primary or an alternate device. A family-4 printer must be a primary device.

More information about using each device family is given in the following sections.

Attached 3270 printer as a family-1 primary device

You can treat a printer as an ordinary family-1 device, if it is directly attached to your program. You can use exactly the same source code to create your graphics and alphanumerics as for a display device. The DSOPEN will determine the destination of the ASREAD and FSFRCE output. In a CMS environment, the program would be like the example below.

```
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    */
                                    /* 3287 4-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 printer as primary   */

CALL GSFLD(1,1,30,70);              /* Define graphics field    */
                                    /* 30 rows by 70 columns    */
CALL GSSEG(0);
CALL GSMOVE(24.0,70.0);             /* Start to draw graphics   */
     and so on...
CALL FSFRCE;                      /* Send 1st output to printer */
```

Before the program is executed, the printer will be attached to the user's VM machine at virtual address '061'. If, instead, a display is attached at address '061', the program will work equally well, if it has at least 30 rows and 70 columns.

When the primary device is a printer, the page may be any size such that:

number of rows x number of columns ≤ 16000

The default page size is determined by the device token. The default token for a printer gives a default page size of 80 rows by 132 columns. Only the leftmost 120 columns will appear in color on the IBM 3287 printer. As always, the graphics field will default to the page size.

Under CMS, you will need exclusive control of a directly attached family-1 device. You can avoid this by spooling the output to RSCS (the Remote Spooling Communication System) Networking. You will 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=L87,
         PROCOPT=((CPSPOOL,TO,RSCS),(CPTAG,REMPRT7,PRT=GRAF))
```

The TONAME parameter sends the output to the virtual punch. The two processing options first spool the punch file to RSCS, and then tag 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 21. Device support" on page 367 and in *GDDM Installation and System Management.*

Queued printer as a family-2 primary device

The output to a queued printer is first passed to the GDDM Print Utility and then to a printer. The exact mechanism varies according to the subsystem (see "Printing GDDM family-2 print files" on page 407).

When a queued printer is opened, various parameters may be set. They form a **print-control option group** within the processing options list (see "Device processing options" on page 370).

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 */
                                   /* (See note 3 for discussion*/
                                   /* of the following options) */
PROCOPT_LIST(2)=8;                 /* No. of fullwords following*/
                                   /* in  this  option  group   */
PROCOPT_LIST(3)=0;                 /* Do not print heading page */
PROCOPT_LIST(4)=2;                 /* No. of copies required    */
PROCOPT_LIST(5)=10;                /* Maximum depth for FSLOG   */
                                   /* (FSLOG is described later)*/
PROCOPT_LIST(6)=3;                 /* Depth of top margin       */
PROCOPT_LIST(7)=5;                 /* Width of left margin      */
PROCOPT_LIST(8)=0;                 /* Depth of bottom margin    */
PROCOPT_LIST(9)=80;                /* Maximum width for FSLOG   */
PROCOPT_LIST(10)=0;                /* Default translation       */
```

```
NAME_LIST(1)='PRINT65';              /* CMS file name (See note 1)*/

/*    DEVICE-ID  FAMILY  DEV_TOKEN   OPTIONS        WHICH DEVICE */
CALL DSOPEN(31,    2,      '*',   10,PROCOPT_LIST,  1,NAME_LIST );
                                   /* Open queued-printer device*/
```

The call requires some explanation:

1. The name of the queued printer device is subsystem-dependent. Briefly, it is a terminal name on CICS/VS, an LTERM name on IMS/VS, a VTAM LUname on TSO, or a file name on CMS (or, of course, a nickname on any of these subsystems).

2. On CMS, you can specify a file type and file mode as the second and third elements of the name list. The example supplies only the file name, in which case GDDM supplies a default file type of ADMPRINT and file mode of A1.

3. The most commonly used print control option is probably the number of copies.

   Some other parameters in the list require further mention:

   - 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 **heading 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 heading page (the default), or to 0 to suppress it.

   - The fifth, eighth, and ninth words are applicable 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 404 and "Sending a character string with control character to printer using call FSLOGC" on page 405.

   - The tenth word is a rarely-used parameter affecting device-type translation.

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            */
```

Then, any further statements (such as page-create or create graphics field) will refer to the queued printer.

System printer as a family-3 primary device

GDDM supports alphanumerics-only output to these IBM system printers: 1403, 3211, and 3800. All the alphanumerics calls may be used with these devices with the exception of symbol-set functions.

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,    'S1403N8',  0,PROCOPT_LIST,  0,NAME_LIST);
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 1982');
     and so on ...
CALL FSFRCE;                    /* Send output to system printer */
```

- 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).

  The name parameter is subsystem-dependent. Briefly, it is a transient data destination on CICS/VS, an LTERM name on IMS/VS, a SYSOUT DDNAME on TSO, and a filename on CMS (the file type and file mode defaulting to ADMLIST and A1).

- The device-token in the example, 'S1403N8', specifies a 1403 printer with page size of 88 lines by 85 columns and line spacing of 8 lines to the inch.

Composed-page printer as a family-4 primary device

GDDM supports the 4250, 3800 Models 3 and 8, and 3820 composed-page printers. They will print monochrome versions of any graphics that could be displayed on a screen. In addition, they will produce sets of color-separation masters, from which printing plates for color illustrations in publications can be prepared (see "Color masters for publications" on page 415 for further information).

They do not support alphanumerics: only the graphics will be printed. Other restrictions are listed in "Restrictions with composed-page printers" on page 419.

The output goes to a high-resolution, binary-image file, which may be:

- Spooled as a printer data stream directly to a printer. It is then called a **primary data stream.**

- Included as part of some other printer data stream. It is then called a **secondary data stream.**

- A bit-for-pixel image array suitable for processing by another program. This is called an **unformatted** or **canonical data stream.**

The following code shows a typical DSOPEN for an IBM 4250, together with its parameter values and the necessary DSUSE call. The only change required to use a different composed-page printer would be to select a suitable device token. (For a list, see the *GDDM Installation and System Management* for your system.)

```
DCL PROCOPT(6) FIXED BIN(31);

    PROCOPT(1) = 7;                          /* Swathing           */ /*A*/
    PROCOPT(2) = 10;                         /* 10  swathes        */ /*A*/

    PROCOPT(3) = 8;                          /* Page size          */ /*B*/
    PROCOPT(4) = 85;                         /* 8.5 inches wide    */ /*B*/
    PROCOPT(5) = 110;                        /* 11 inches deep     */ /*B*/
    PROCOPT(6) = 0;                          /* 1/10 inch units    */ /*B*/

DCL NAMELIST(1) CHAR(8);

    NAMELIST(1) = 'SALES83';                 /* File name          */ /*C*/

     /* DEVICE_ID   FAMILY   TOKEN     PROC_OPTIONS   NAME_LIST */
CALL DSOPEN (12,      4,   'FINE600',  6,PROCOPT,     1,NAMELIST);

CALL DSUSE (1,12);                  /* Make it the primary device */

CALL FSPCRT(1,110,85,1);            /* Create 1/10 inch grid      */ /*D*/
CALL GSFLD(11,11,90,65);            /* Put top left corner of     */ /*E*/
                                    /* graphics field at 11,11    */
                                    /* and make it 90 rows deep   */
                                    /* by 65 columns wide         */
```

High-resolution image file: For printing, your application must write its output to a high-resolution image file, which later becomes the input to another IBM program product - either the Composed Document Printing Facility (CDPF) for the 4250 or the Print Services Facility (PSF) for the 3800 Models 3 and 8 and the 3820. These pass the output on to the printer.

You specify the name of the file in the name-list parameter of the DSOPEN call, as shown at `/*C*/` in the example. If there is no file with the specified name, GDDM creates one. An ASREAD or FSFRCE call sends the output to the file, rather than to a terminal device.

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. More information about names under CMS and other subsystems is given in the *GDDM Base Programming Reference* manual.

Paper size: You can specify on the DSOPEN call the physical size (in tenths of an inch or in millimeters), of the paper area on which the page is to be printed. Or you can omit the specification and include a device token from which GDDM can obtain it. Device tokens are described in *GDDM Installation and System Management.*

The size is specified in a processing option. It should not exceed the values in the device token. In the example, the size is specified at `/*B*/`. The option code is 8. The next fullword after the code specifies the width, and the one after that the depth. The last fullword of the group specifies the units: 0 means 1/10 inches, 1 means millimeters. The example specifies a size of 8.5 inches by 11 inches.

Rows and columns: The composed-page printers are not cell-based devices. However, GDDM still uses a specified or defaulted set of rows and columns on the current page; they form a conceptual grid on which your program can specify the position and size of the graphics field. The horizontal spacing of the grid is:

```
(width of paper area) / (number of columns)
```

and the vertical:

```
(depth of paper area) / (number of rows).
```

The example creates a page with 110 rows and 85 columns, at `/*D*/`. Because the page is 8.5 inches wide and 11 inches deep, its conceptual grid will have elements 1/10 inch square.

Positioning the graphics field: The graphics field created at `/*E*/` will be 65 columns wide by 90 rows deep, that is, 6.5 inches by 9 inches. Its top left-hand corner will be 10 rows down from the top of the paper and 10 columns in from the edge. The graphics area will therefore be surrounded by a one-inch margin, assuming that the printer is physically loaded with paper of the size specified in the DSOPEN call.

If you omit the GSFLD call, the graphics field will cover the whole page, and will therefore fill the whole of the physical area of paper defined implicitly by the device token, or explicitly by the FSPCRT.

Spill file: GDDM keeps a record of the graphics created by your API calls in graphics data format (GDF) (see "Chapter 13. Picture handling in graphics data format" on page 171). GDF is an intermediate form between the API and the rastered images required by terminal devices.

When the primary device is a composed-page printer, all lines have to 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 will, by default, hold the GDF for composed-page printers on external storage, in a **spill file**. You can specify, in a processing option, that the GDF is to be held in main storage instead. The option code is 6, and the fullword containing this code must be followed by one other containing the value 1, meaning no spill file should be used.

Using a spill file saves main storage, but increases processing time because of the additional external storage I/O.

Swathing: The rastered image for a composed-page printer may contain a very large amount of data, because of the high pixel density. To avoid keeping it all in main storage, GDDM will write it to the high-resolution image file in sections or swathes. The swathes are equal-sized horizontal slices of the picture, and GDDM processes each one completely and writes it to the file before starting on the next one.

Swathing saves main storage but increases processing time, because the whole picture (as held in GDF) must be scanned for each swathe.

You specify the number of swathes with option code 7. The example specifies 10 swathes at `/*A*/`. The default is 1, which means no swathing - the picture is processed in a single GDF scan.

Primary and secondary data stream

Input to CDPF or PSF that comprises a complete document is said to be a primary data stream. GDDM creates a primary data stream by default.

You can specify in a processing option (option group 5) that GDDM is to create only part of a document. In this case, the GDDM output is known as a secondary data stream. A file containing a secondary data stream must be merged by CDPF or PSF with one or more other files to create a complete document.

You would need to create a secondary data stream if your application prepares the illustrations for a publication, while the text is prepared by another means, such as the IBM Document Composition Facility. CDPF or PSF would then merge the illustrations and text to create a complete document. During preparation of an illustration, you might need to print it without the text, for checking. In this case, you would specify that GDDM is to create a primary data stream for the illustration.

If a picture contains text that uses the 4250 fonts (see "Using typographic fonts on a family-4 4250 printer" on page 411) in addition to graphics, you would normally need to create a secondary data stream. This is to avoid exhausting CDPF program storage.

The option code for the data-stream type is 5. Its associated value can be either 0, meaning a primary data stream (the default), or 1, meaning a secondary data stream. An example is included in Figure 104 on page 419.

Unformatted (canonical) output

Instead of output in a form suitable for CDPF or PSF, you can specify that GDDM is to create an unformatted data stream. This is simply an uncompressed bit pattern representing the image. Its format is device-independent.

You select formatted or unformatted output with option code 9. An associated value of 0 means unformatted, and 1 means formatted (the default).

Printer as an alternate device

GDDM allows you to send copies of the primary device's output to an alternate device. You can use a family-1, -2, or -3 printer as an alternate device and so, for instance, obtain a hard copy of the output to a display terminal. A program for doing this is shown in "Example program: Copying screen output to a printer" on page 405.

The DSOPEN calls described in the earlier sections of this chapter apply equally to alternate devices and primary ones.

The print control processing options described in "Queued printer as a family-2 primary device" on page 397 can be applied to family-1, -2 or -3 printers when they are being used as alternate devices. Their main use is to set the margins around the printed area. The number-of-copies option (the fourth one in the list) will be honored for family-2 devices only.

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.

Four calls, FSCOPY, GSCOPY, FSLOG, and FSLOGC, send output to the alternate device. They will now be described.

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 a family-1 or -2 printer, the alphanumerics, graphics, and image data are copied. If the alternate device is a family-3 printer, only the alphanumerics are copied.

The output to a family-1 or -2 printer is subject to the considerations outlined in "Mixed graphics and alphanumerics" on page 409. Unsatisfactory results may occur when you try to copy mixtures of alphanumeric and graphics output, because the relative positions of the two types of data will be subject to change. See also "Printing and plotting images" on page 358.

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 will be 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 will occupy a different portion of the page (compared with that on the primary device), and consequently will be positioned differently in relation to any alphanumeric fields. More information is given in "Mixed graphics and alphanumerics" on page 409.
- Alphanumeric field and character attributes are retained on the printed copy whenever possible. Underscore is retained, for example, but blinking is not.
- Wherever symbol sets were used to create the original picture, they will be 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 "Symbol sets for alphanumerics" on page 221), GDDM will load 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:

If your program uses a different symbol set for printing (by, for instance, employing a substitution character), this 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 will be 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 will **not** get an elongated version of the picture stretching right across the page. You will 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 will make 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 "Queued printer as a family-2 primary device" on page 397).

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 Figure 98. 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.

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'

Figure 98. Carriage-control codes for FSLOGC

Example program: Copying screen output to a printer

The example program in Figure 99 on page 406 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 heading 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             */

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 onlymap 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 99. 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 will close it (and proceed 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 will create 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 that must be processed by the GDDM Print Utility. This section gives an overview of the utility. Full details are given in the *GDDM Base Programming Reference* manual.

On subsytems 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 will be separated by header pages (unless suppressed on the DSOPEN).

There is a different version of the utility for each subsystem.

On CICS/VS, IMS/VS, and OS/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=4224SE,
         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 "Attached 3270 printer as a family-1 primary device" on page 396. 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 will invoke it whenever your program completes the creation of a print file. For more information about this technique, see *GDDM Installation and System Management.*

Printing non-GDDM sequential files

Under CMS and TSO, you can use GDDM utilities to print ordinary sequential files on printers belonging to GDDM family-1, such as the 3287 and the 4224.

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.

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 Figure 98 on page 405. You can specify a device token (see "Opening a device using call DSOPEN" on page 367) 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 treats the first byte according to the carriage-control specification in the DCB for the file.

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 Programming Reference* manual 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; a typical alternate device, the 3287 printer, has cells of 10 by 8. All the graphics, therefore, has 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 "Chapter 13. Picture handling in graphics data format" on page 171).

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 3279 when 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 will change from device to device. On a printer, 32 rows by 80 columns, for example, gives a different aspect ratio from that on a 3279 display unit.
- If the aspect ratio is explicitly set (by calling GSPS), the **position** of the picture space within the graphics field will vary from device to device. This is no problem unless alphanumeric fields are present. The relative position of alphanumerics and graphics will then be affected.

- The default page size varies from device to device. On a 3279 it is 32 by 80; on a printer it is 80 by 132. The output from programs that use the default page-size will therefore differ 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 special steps are taken.

  These are the necessary steps:

  - The same graphics field must be (explicitly) specified for each device.

  - No GSPS or GSVIEW call may be made.

  Provided these precautions are taken, the alphanumerics and graphics will maintain their relative positioning. The aspect ratio of the graphics will change, though. It is not possible to maintain both factors.

- If the program uses mode-3 graphics text rather than alphanumerics, there will be no problem over relative positioning, when the character box (the character size) is explicitly set.

  For example, you may have a section of graphics calls that draws a geographical map. These calls will be a mixture of GSLINEs, GSAREAs, and mode-3 GSCHARs. Assume that the map has been produced and tested using a 3279. When you are satisfied with the output, you may decide to run the same section of code 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 will be 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 will 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 will be preserved, but the aspect ratio of the graphics will change. A value of 0 (the default) means the reverse. The call must be executed for each page being copied before the FSCOPY call.

Colors and shading patterns on the IBM 3268 and 3287 printers

The 3268 and 3287 printers have only four colors (blue, red, green, and black), instead of the seven supported by the 3279 display unit. Graphics destined for the printer should therefore avoid using pink, turquoise, and yellow – they will all default to black. Remember, also, that neutral (color 7) is white on the display screen, and black on the printer. Color −2 is explicit white, which means background on a printer. Color −1 is explicit black, which means background on a screen.

Strange effects will occur with shaded areas if the cell-size of the pattern image symbol set does not match that of the printer. Only part of each cell will be

shaded. To avoid this, you must use either a symbol set of the correct cell size (see "Symbol sets for alphanumerics" on page 221), or the system-defined patterns (see "Setting the current pattern, using call GSPAT" on page 38).

When using multicolored pattern sets, remember that every pink, turquoise, yellow, or white pixel in a pattern will appear as black on the printer.

Using loadable symbol sets on family-3 3800 printer

The 3800 printers permit 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 alphanumeric text" on page 223).

These are typical calls:

```
CALL ASFPSS(22,3); /* Field 22 will be 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 will 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. PL/I does not support hexadecimal constants, so the value X'02' had to be assigned into a temporary variable CHAR1.
- 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, using attributes" on page 58. They also differ in character and line spacing. In mode-1, the spacings will follow the width and depth definitions contained within the fonts. This means the text will be proportionally spaced along the lines. In mode-2, the characters will be spaced at the width and depth of the current character box, which means the spacing along the lines will be 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.*[1]

3. *Font AFT06008 is 10 point Monotype Times New Roman.*[2]

The fonts are illustrated in *IBM 4250 printer type font Catalog*. You can obtain 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, order number SH35-0070).

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 will be correct.

---

1 Univers is a trademark licensed to IBM by Allied Corporation.

2 Monotype Times New Roman is a trademark licensed to IBM by The Monotype Corporation, Limited.

Code pages

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). However, your installation may override this, and make another code page the default. Instructions for overriding the GDDM default are given in the *GDDM Installation and System Management* manual.

Example program: Using 4250 fonts

An example of how to use 4250 fonts is given in Figure 101 on page 415, with the output in Figure 100 on page 414.

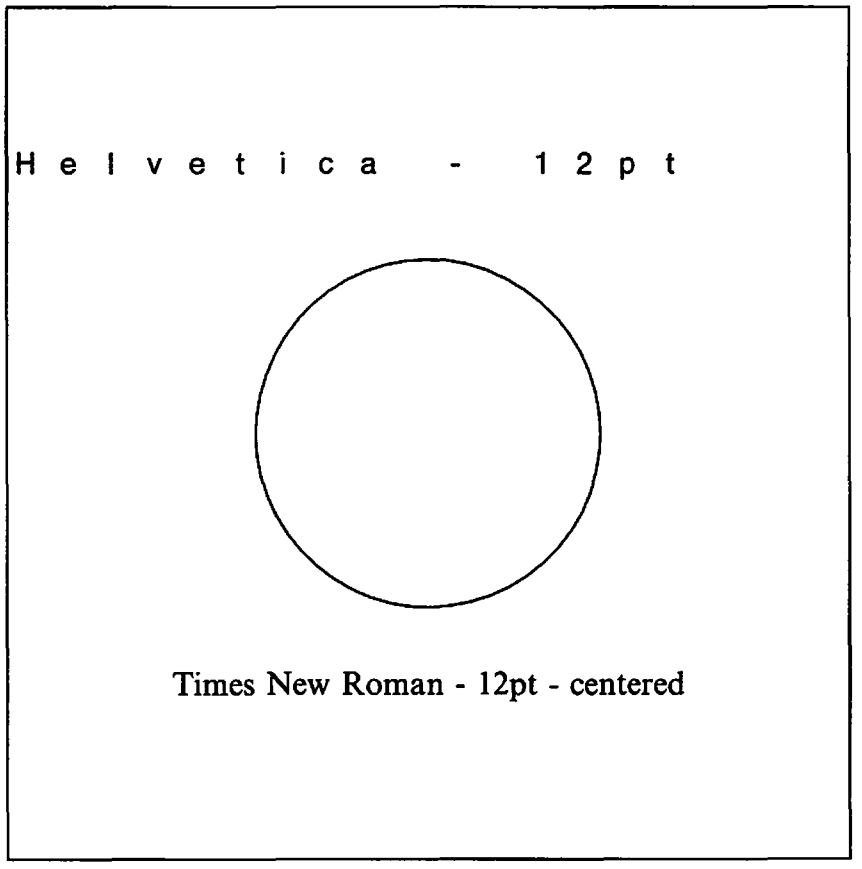


Figure 100. Output of 4250 font example

```
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 101. Example of using 4250 fonts

Color masters for publications

You can use the family-4 (composed-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 102 on page 416. The images on the masters have to be transferred photographically to the printing plates.

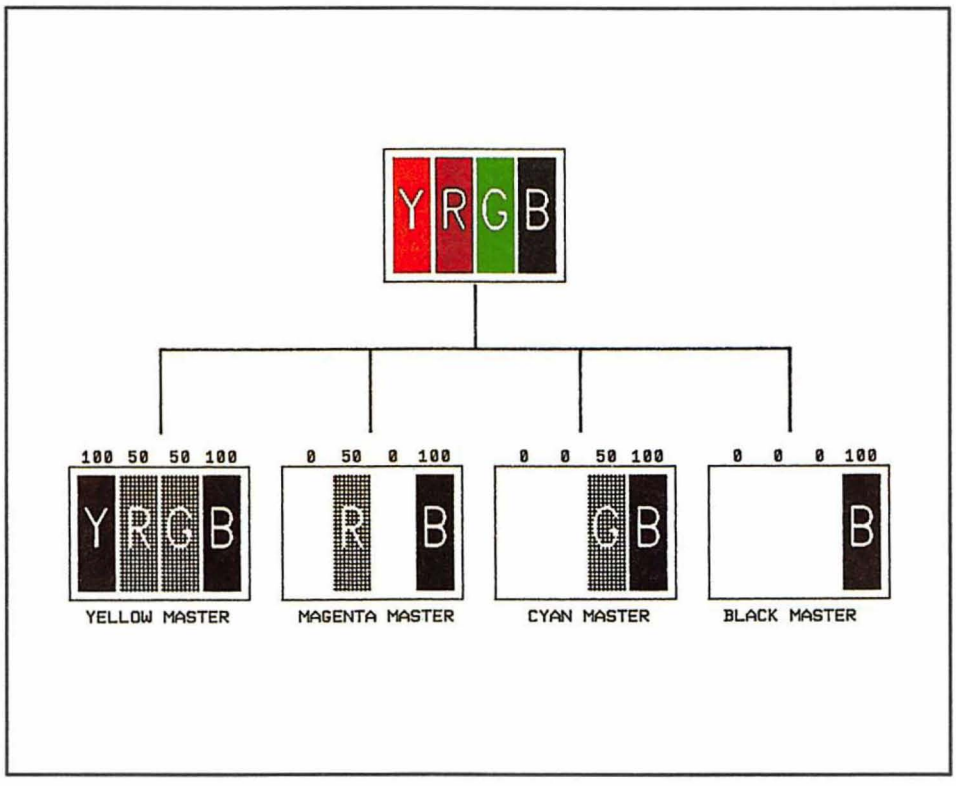


Figure 102. How a picture is changed into a number of color masters

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 a new current color, using call 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 will appear 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 certain density will be 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 will be 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 103 on page 417), gives an indication of what such a pattern set might be like.

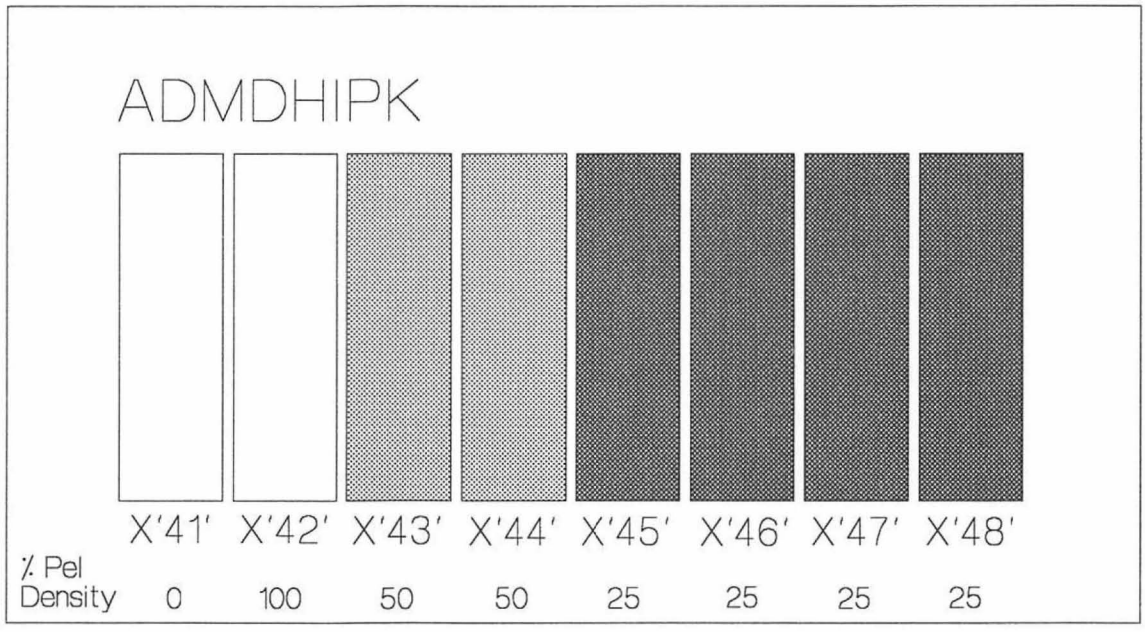


Figure 103. ADMDHIPK, the GDDM sample symbol set for color masters

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 Programming Reference* manual. Here is an example:

```
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 to 239 (X'41' to X'EF') in

the same way as for user-defined patterns in the GSPAT call, which is described in "Setting the current pattern, using call GSPAT" on page 38.

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 will be 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 will vary from one printing establishment to another, because of variations in inks, papers, printing technology, and so on. To obtain 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 1 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 104 on page 419. 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.

Each color master is created as a high-resolution image file of its own. Under CMS, GDDM will use the specified file name, and assign 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 Programming Reference* manual 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;              /* Formatting          */
    PROCOPT(4) = 1;              /* Yes                 */

    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 104. Creating color-separation masters

Restrictions with composed-page printers

This is a summary of the restrictions that apply when you use a composed-page printer:

- The composed-page printer must be the primary device.
- A family-4 printer cannot be an alternate device. If you want to view a picture before printing it, you can either use the technique described in "How to use more than one primary device" on page 372, or make the printer the primary device and copy to the screen.
- The GDDM pages must not be mapped; in other words, the MSPCRT call is not allowed.
- Only graphics can be printed. This implies that:
  - Alphanumerics calls are ignored.
  - Calls that refer to PS stores are invalid.
  - The only part of the current page that goes to the printer (by way of the image file) is the graphics field.

Using the IBM 4224 printer

This is a summary of restrictions that apply when you use a 4224:

- GDDM requires the printer pitch to be set to 10 characters per inch, and the number of lines per inch to 8. You can set these parameters using the 4224 operator panel.
- Alphanumeric data are printed in Near Letter Quality (NLQ) mode, except for APL characters, which are printed in Data Processing (DP) mode.
- GDDM cannot load user-pattern sets to the 4224 printer, and will issue a warning message if a pattern from such a set is referenced by the application. The 4224 uses its internal default shading pattern (solid) instead.
- The 4224 will not print a marker selected from the system marker set if any part of the enclosing marker box falls outside the graphics field. Markers selected from a user-defined marker set are not subject to this restriction.
- The 4224 supports a graphics mix mode of overpaint only. GDDM issues a warning message if the application requests any other form of mix mode.
- Symbol sets referenced by alphanumeric fields will only be loaded into extended storage models of the 4224. GDDM issues an error message if the application uses loadable symbol sets when the printer does not have extended storage. Affected characters will be printed using the 4224 resident default font.
- Using the 4224 extended storage (512K bytes) model considerably reduces the possibility of graphics output causing printer storage overflow. If overflow is unavoidable, GDDM issues a warning message and sends only that amount of picture data that will fit in the available printer storage.

Chapter 23. Using plotters

You can send graphics output to a plotter attached to a 3179-G, any of the 3270-PC/G or /GX family (except under IMS), and any of the 5550 family. A typical use is for making hard copies of screen graphics.

Only the graphics field is sent to the plotter. Alphanumerics are not supported. When a plotter is the current device, alphanumeric calls are invalid. Graphics primitives outside segments are not plotted.

Plotters are family-1 devices. 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 (say a family-2 printer) to a plotter.

DSOPEN for plotters

A DSOPEN call for a plotter requires a two-part name, identifying the work station 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 work station from which the program was invoked. On CMS, you can send the output to a plotter on a different work station by specifying the work-station's address.
- A work station can have more than one plotter attached to it. All the plotters are given names when the work station 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, "first" meaning the top one in the IEEE488 Channel Customization display panel.

GDDM can query the plotter, so you can specify a device token of * rather than an explicit token name.

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 can be set by the application program using processing options on the DSOPEN, some by the operator, and some by both.

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);
```

Here is a summary of all the processing options for plotters:

- Pen velocity

  Specifies the speed of the pens (see "Optimum pen speed and force" on page 440).

  1st fullword - option group code:
  11

  2nd fullword - the pen velocity option
  0 - use hardware setting (the default)
  1 through 255 - the velocity in centimeters per second.

  If 0 is specified or allowed to default, the pen velocity that is set on the plotter takes effect. If a value in the range 1 through 255 is specified, this overrides the setting on the plotter. If the value is more than the plotter's maximum, the maximum is used.

- Pen width

  Specifies the width of the pens in tenths of a millimeter. The width actually used for plotting depends on which pens have been loaded into the plotter's pen holders, and 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.

  1st fullword - option group code:
  12

2nd fullword - the pen width option:
0-0.3 millimeters (the default)
1 through 10 - pen width of 0.1 through 1.0 millimeters.

- Pen pressure

  Specifies the force of the pen onto the paper (see "Optimum pen speed and force" on page 440). Some types of plotter do not have variable pen pressure, in which case the processing option is ignored.

  1st fullword - option group code:
  13

  2nd fullword - the pen pressure option:
  0 - use the hardware setting (the default)
  1 through 255 - pen pressure in grams.

  If 0 is specified or allowed to default, the pen pressure that is set on the plotter takes effect. If a value in the range 1 through 255 is specified, this overrides the setting on the plotter. If the value is more than the plotter's maximum, then the maximum is used.

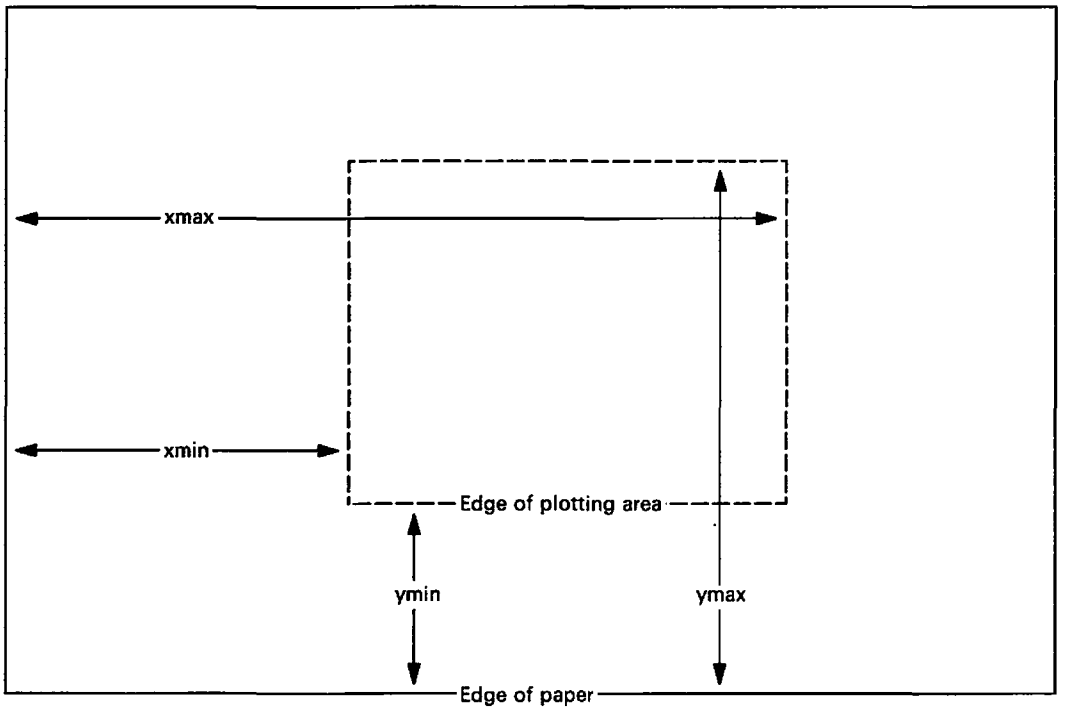


Figure 105. Plotting area

- Plotting area

  Specifies on which part of the paper the picture is to be plotted, as shown in Figure 105. 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. The DSOPEN example earlier in this section shows how to specify a plotting area.

1st fullword - option group code:
14

2nd fullword - the minimum x value (XMIN in Figure 105)
0 through 100

3rd fullword - the maximum x value (XMAX in Figure 105)
0 through 100

4th fullword - the minimum y value (YMIN in Figure 105)
0 through 100

5th fullword - the maximum y value (YMAX in Figure 105)
0 through 100.

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.

Either the maximum x or maximum y values, or both, can be less than the corresponding minimum, in which case the picture will be reflected about the x or the y axis, or both.

- Paper size

Specifies 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.

1st fullword - option group code:
15
2nd fullword - the paper size code:
0 - actual paper size in plotter (the default)
1 - A4 or A
2 - A3 or B
3 - A2 or C
4 - A1 or D
5 - A0 or E
3rd fullword - the size type code:
0 - same as 1 (the default)
1 - ISO sizes
2 - ANSI sizes.

GDDM derives the plotting area it will use 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 or size type 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.

- Picture orientation

  Allows you to rotate the picture by 90 degrees.

  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). Using the orientation option, you can make the x axis parallel to the shorter side (portrait format).

  If a plotting area of 0,0,0,0 has been specified, picture orientation, as also the plotting area, depend on the plotter's hardware settings. The default orientation is the hardware default. Whether this means that the x axis is parallel to the longer or the shorter side of the paper depends on the type of plotter and the paper size. The plotter operator's manuals give details. Pressing the rotate button will make the x axis parallel to the alternative side.

  So the rule is: the plotting area and the picture orientation are set using either processing options for both or buttons on the plotter for both.

  1st fullword - option group code:
  16
  2nd fullword - the orientation option:
  0 - same as 1 (the default)
  1 - no rotation
  2 - rotates the picture through 90 degrees.

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 434).

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 work station. 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 work station 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 Programming Reference* manual. Because the rows and columns are defined as fitting the plotting area, changing the area's dimensions will change 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 106 on page 428 uses the plotter as the primary device. It plots a picture created by another program and stored on a segment library. The picture, called SAVEP, 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 will 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 will be the whole paper. The pen velocity and pen pressure will be 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 will 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 426.

The program displays setup instructions for the IBM 7375 plotter, which can detect the size of paper loaded, and has variable 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 106 (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( 'SAVEP',     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 106 (Part 2 of 2). Program using plotter as primary device

Copying screen output to plotter

The program in Figure 107 on page 430 first creates a picture on the screen, then copies it to the plotter.

The picture is actually created by the operator using a stroke device, in the block of code at /*A*/. It could be created in any of the other usual ways, including primitive calls like GSLINE and GSCOL, or loading with a GSLOAD.

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 /*F*/ for the GSCOPY call, /*G*/, to send the picture to the plotter. This call requires the number of notional cells in each row and column of the plot area. These are queried at /*E*/.

Unless you specify otherwise with a GSARCC call, GDDM maintains the aspect ratio of the graphics when copying to the plotter: the picture fills as much of the area specified on the GSCOPY 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 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);
DCL (DEVTYPE,DEVID) FIXED BIN(31);
DCL DFLAGS(100) FIXED BIN(31);
DCL (XARRAY,YARRAY)(100) FLOAT DEC(6);
DCL NUM FIXED BIN(31);

CALL FSINIT;

/*************************************************************/
/*                  CREATE A PICTURE                         */ /*A*/
/*************************************************************/

CALL GSENAB(5,1,1);              /* Enable tablet or mouse as    */
                                 /* polyline stroke device       */
CALL GSREAD(1,DEVTYPE,DEVID);    /* Read and wait                */
CALL GSENAB(5,1,0);              /* Disable stroke device        */

CALL GSQSTK(100,DFLAGS,XARRAY,YARRAY,NUM);/* Obtain stroke data*/
IF NUM=0 THEN GO TO FIN;         /* if nothing to plot           */

/* Now draw the polyline from the returned arrays of points    */

CALL GSSEG(1);                   /* Begin new segment            */
CALL GSMOVE(XARRAY(1),YARRAY(1)); /* Move to start of line       */
DO I=2 TO NUM;                   /* Draw the polyline            */
    CALL GSLINE(XARRAY(I),YARRAY(I));
END;
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 secondary device */ /*D*/
```

Figure 107 (Part 1 of 2). Program using plotter as secondary device

```
/*************************************************************/
/*                  QUERY ROWS AND COLUMNS                   */
/*************************************************************/

CALL DSQDEV(202,DEV_TOKEN,0,PROCOPTS,0,NAME_LIST,4,QDEV);          /*E*/
ROWS    = QDEV(3);
COLUMNS = QDEV(4);

/*************************************************************/
/*                  SEND PICTURE TO PLOTTER                  */
/*************************************************************/

CALL FSPSEL(0);                        /* Select page with picture */ /*F*/
CALL GSCOPY(ROWS,COLUMNS);                                            /*G*/

FIN:

CALL FSTERM;

%INCLUDE ADMUPINA;
%INCLUDE ADMUPIND;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

END PLOT2;
```

Figure 107 (Part 2 of 2). Program using plotter as secondary device

Plotting to scale

Plotter output often needs to be of a particular physical size. You can do this by making each window (world-coordinate) unit represent a particular physical measurement in the plotted output. The example program in Figure 108 on page 432 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 - the 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). These statements would replace /*D*/ and /*E*/:

```
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 at /*G*/, it will be 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 108 (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 108 (Part 2 of 2). Scale plotting program

Using nicknames to direct and control the output

If you have a program that currently sends graphics to a printer, you can have the output sent to a plotter instead by creating a suitable nickname file (see "Nicknames" on page 378). Here are three nickname statements that will 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 will redirect any output for family-1, -2, and -4 printers named as 061, PR2, and PR4, respectively. The output will go 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.

You can also make the reverse change (diverting output from a plotter to a printer) using nicknames. For instance, this statement will send 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 will send 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)
```

Such a statement could be used to plot charts created by the GDDM Interactive Chart Utility (ICU). The ICU allows the user to send charts to a printer, and to specify the name of the printer. The ICU issues a DSOPEN call for a family-2 device with that name. If the user specified a name of PLOTTER, the above nickname statement would send the output to the terminal's first (or only) plotter.

In all cases, you can supply 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 Programming Reference* manual. Here is an example of a nickname statement that specifies two processing options. The first is a pen pressure of 20 grams, and the second a zero plot area, which allows the operator to define the area using the plotter controls:

```
ADMMNICK FAM=1,NAME=(*,PLOTTER5),
         PROCOPT=((PLTPENP,20),(PLTAREA,0,0,0,0))
```

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 Figure 109.

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

Figure 109. Suggested color scheme for plotter pens

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 Figure 110 on page 436. 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 422).

  To use pen 8, you can specify color 0 or 16, or allow the color to default.

- 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 will be 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, will use pen 6 (color 8 is exceptional - 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 will use 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.

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)

Figure 110. Color and pen numbers on plotters

* On 3270-PC/GX work station. For a 3179-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.

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.

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 -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 434.

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 work station that is also being plotted, the results will 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. The following actions will minimize the processing:

- 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 111 on page 438. 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 non-solid 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.

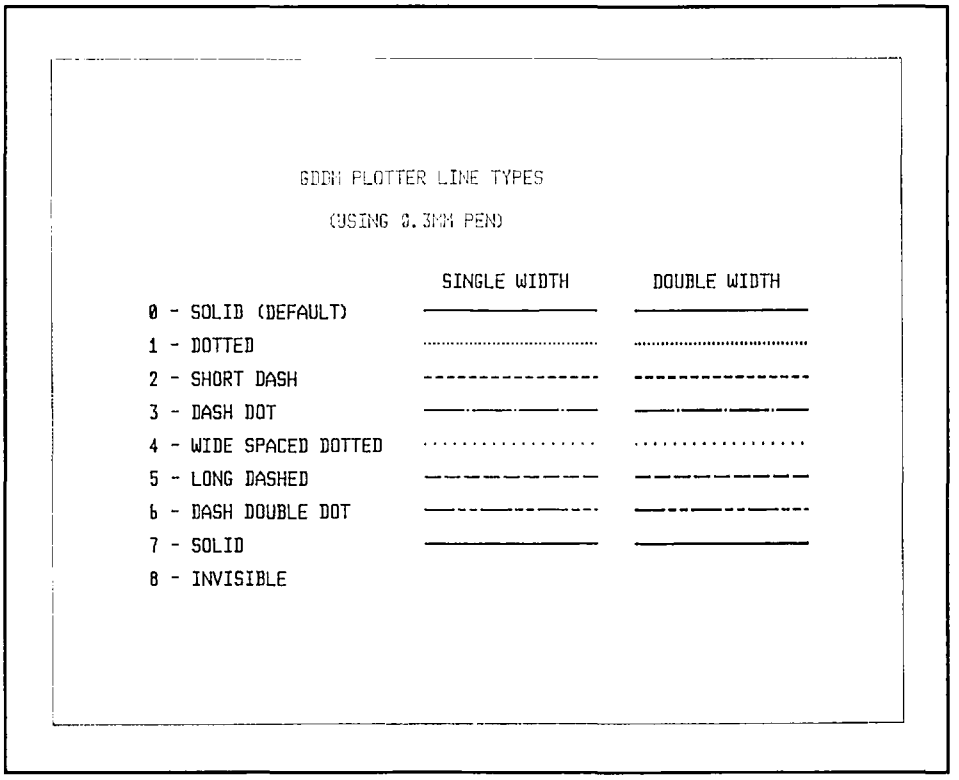


Figure 111. The eight GDDM line types for plotters

Shading patterns

There are sixteen special GDDM-defined shading patterns for plotters. They are illustrated in Figure 112 on page 439. 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 422). 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 will not be solid: there will be gaps between the shading lines.

Figure 112. The 16 GDDM shading patterns for plotters

Symbol sets

The functions described in "Chapter 15. Symbol sets" on page 219 apply to plotters as well as terminals. The symbols are drawn using the calls described in "Chapter 7. Basic graphics text" on page 55.

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 426. 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 force

The most suitable speed and force 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 Programming Reference* manual and in the plotter operating manuals.

Chapter 24. Windowing

This chapter tells you about the GDDM calls that organize the screen of a display device into rectangular areas, using the following different types of presentation structure:

- **Partitions** (application windows)
- **Operator windows**

Partitions and operator windows were first introduced in "Chapter 9. Hierarchy of GDDM concepts" on page 89. 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 "Operator windows" on page 467 in the second part of this chapter.

Partitions

This section tells you how to create and use partitions. Partitions can be real or emulated. The IBM 3290 Information Panel, 8775 Display Terminal, and 3193 Display Station have hardware facilities that allow application programs to create **real** partitions. Alternatively, GDDM will 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 will let 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 – say, a source file in one partition and a compiler listing in another.

  Real 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, say, 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.

A simple partitioning example

Here is 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 113 on page 448.

```
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');

/* 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;

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;
```

The program illustrates some of the concepts of partitioning:

Partition sets

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 is used, rather than a grid defined by the hardware rows and columns, because the cell size has not yet, in general, been determined.

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 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:

0 Use hardware partitioning if the device has it, otherwise use GDDM software emulation

1 Use emulated partitioning in any case

2 The program will not create any partitions.

The fourth element defines whether the partitions can overlap:

0 They cannot (as in the example)

1 They can.

The sample 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.

The sample program 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 Programming Reference* manual. 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 non-partition 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 */
```

There is an example in the program 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 ENTER), 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 ENTER, 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. The call to ASREAD at `/*G*/` will create 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 will change only the data altered by the program. The rest of the screen will remain unchanged. In the sample program, 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 was in partition 2 when the terminal user pressed ENTER, this will be 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 will allow 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 will become 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 will make 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 would be lost if your program overwrote it.

Handling terminal-user errors

After the ASREAD at /*G*/, the program 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 will cause 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.

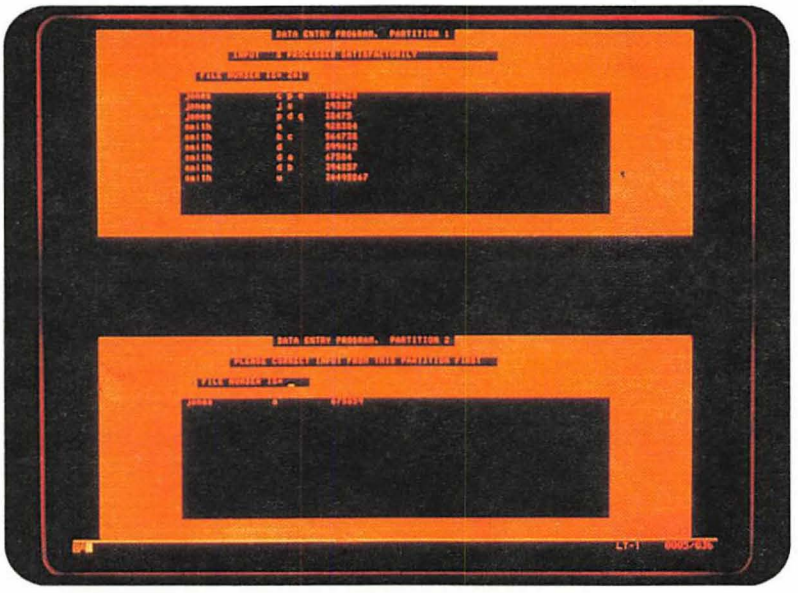


Figure 113. Screen formatted by simple partitioning program

Some other things you can do with partitions

The following 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 skeleton program 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);

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*/

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;
```

A program based on the above skeleton program produced the panels in Figure 114 on page 452 and Figure 115 on page 452. All we added to the skeleton program to produce the screen layouts in the two figures was the code to label each partition, and to draw a line around its border. In practice, for an ICU-like program, the partitions would hold procedural or mapped alphanumerics.

The skeleton program creates nine partitions to each hold 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 will not see any overlapping partitions in the output displayed by the program. This is because, where partitions overlap, we have specified only one of them 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 because partitions 1 through 6 are the only ones that we want to be seen in the first panel that we display using the ASREAD at /*C*/.

In the second panel, we want to display partitions 1, 2, 4, 7, 8, and 9. We do not have to do anything to partitions 1 and 2 for them to be displayed again.

At /*D*/, as we no longer want partition 3 to be shown, we must first make it current, using the PTNSEL call, and set its visibility parameter to 0 (invisible). We 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. In the example, we have used the array that we originally used to create the partition.

Using PTNSEL and PTNMOD, we 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 we set their visibility attribute 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*/ we alter their positions to 5, 9, and 13 respectively, and make them 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.

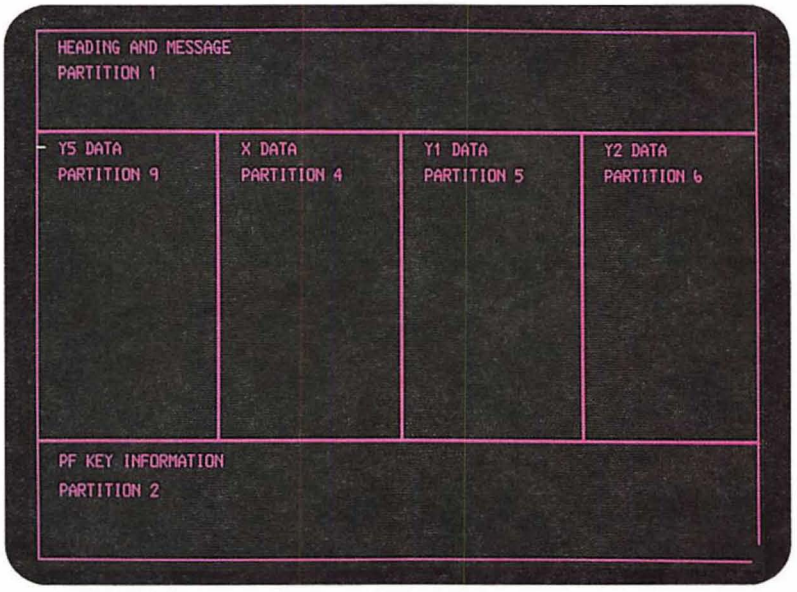


Figure 114. First panel using visible and invisible partitions

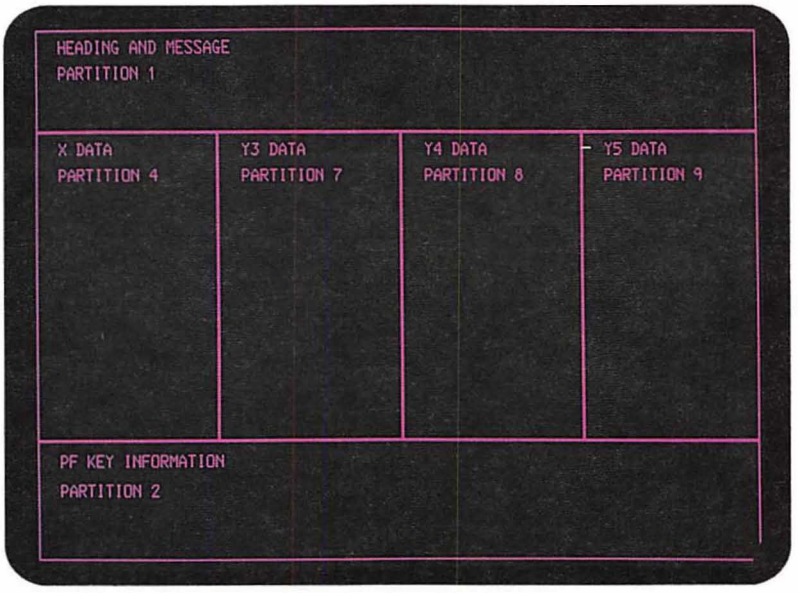


Figure 115. Second panel using visible and invisible partitions

Overlapping partitions

You can overlap partitions. Partitions are opaque, so the part of a partition that is overlapped by another partition will be completely obscured by the top partition.

The next sample program 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);
        .
        .
        .

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 116 on page 454. All that we added to the program was 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 will 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.

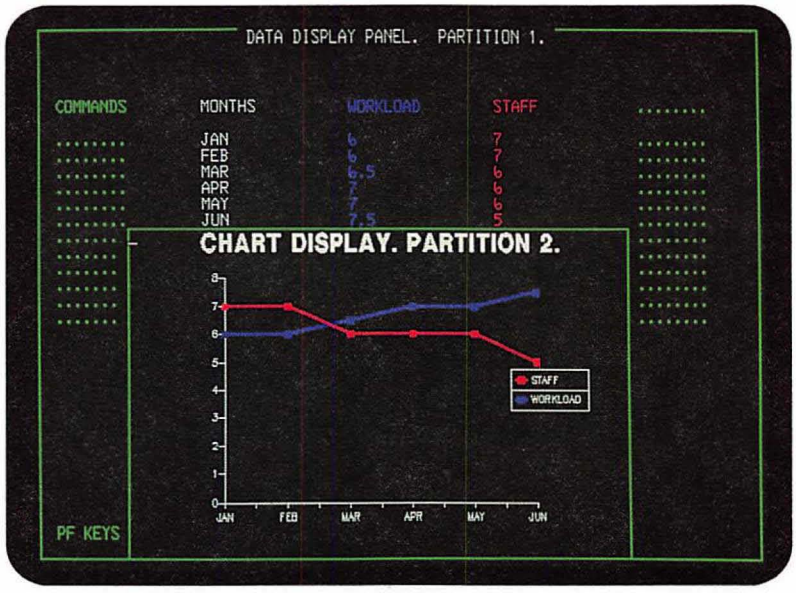


Figure 116. Overlapping partitions

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, say 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 will become 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 will become 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 sample program 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 117 on page 457. 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 ENTER (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 ENTER, 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');

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;

ENDIT:;
CALL FSTERM;
%INCLUDE ADMUPINA;
%INCLUDE ADMUPINP;
%INCLUDE ADMUPINF;
%INCLUDE ADMUPING;

END FOLDERS;
```

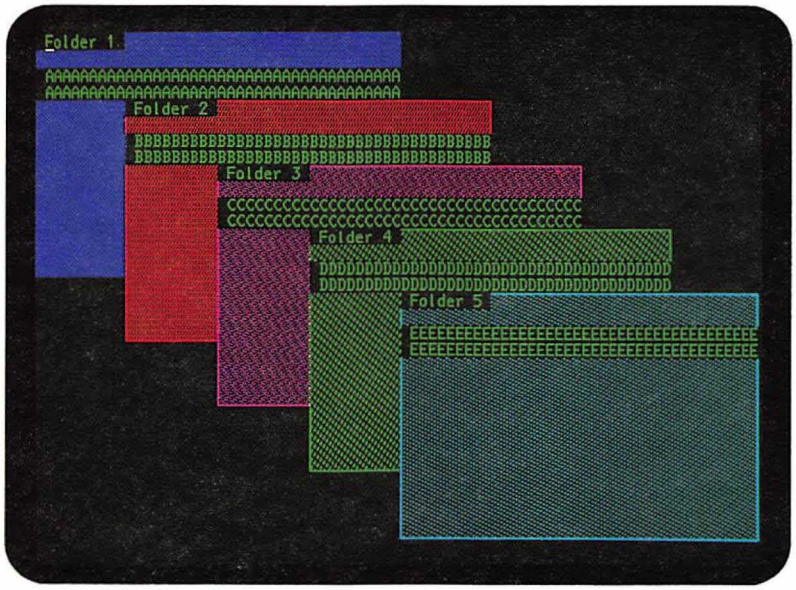


Figure 117. Output from sample partition prioritizing program

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 will be the top partition in the viewing order, and the rest of the partitions in the array will be those that are behind it.

  Ascending The first partition in the array will be the bottom partition in the viewing order, and the rest of the partitions in the array will be 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 will return 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 Programming Reference* manual.

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 461. 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 will change 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 will appear 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 ENTER to send the page back to your program. Because the program is not notified of the change, to it the position of the page window is still line 1. If it resends the page to the terminal, line 1 and not line 20 will be 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 will then be 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 will actually position 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 will move the cursor to within the window. Conversely, if you move the cursor to outside the window, GDDM will reposition 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 will vary the cell size on the IBM 3290 Information Panel according to the number of rows and columns to be displayed. Within limits, it will select 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 93. For example:

```
CALL FSPCRT(1,60,80,0);
```

creates a page 60 rows deep by 80 columns wide. For MSPCRT, GDDM will take the page size from the mapgroup if you do not specify it explicitly.

GDDM will try 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 will choose 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);
```

will cause 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, or both, GDDM will assume 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, will be 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 in Figure 118. The table also shows the loadable cell size, that is, the size that the terminal uses for programmed symbols.

	Cell Size in Pels	
	Width	Depth
Minimum	6	12
Default	6	12
Loadable	9	16
Maximum	16	31

Figure 118. 3290 cell sizes

The terminal will scale its own hardware characters to fit cells whose width and depth 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 will 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 will be visible.

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 will 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 Programming Reference* manual. 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, any graphics will be 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 119 on page 464 combines partitioning with some of the functions described in "Large and small pages." 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 461 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 will handle 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 120 on page 466.

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 will select 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 will again select 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');

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*/
```

Figure 119 (Part 1 of 2). Program using scrollable partitions and two cell sizes

```
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*/

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 119 (Part 2 of 2). Program using scrollable partitions and two cell sizes

Figure 120. Screen with two cell sizes

Operator windows

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, a task manager can use a GDDM instance to allow the terminal 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 terminal user accesses each application, and the end-user dialog with the task manager, through rectangular subdivisions of the display device screen called operator windows. The terminal 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 will run 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 non-GDDM function for terminal input/output. Such non-GDDM 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 in "Task management" on page 481.

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 terminal 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 "Sample program using one operator window" on page 469.

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.) See "Processing options for operator windows" on page 386 for a list of the processing options for a virtual device that are overidden 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 121.

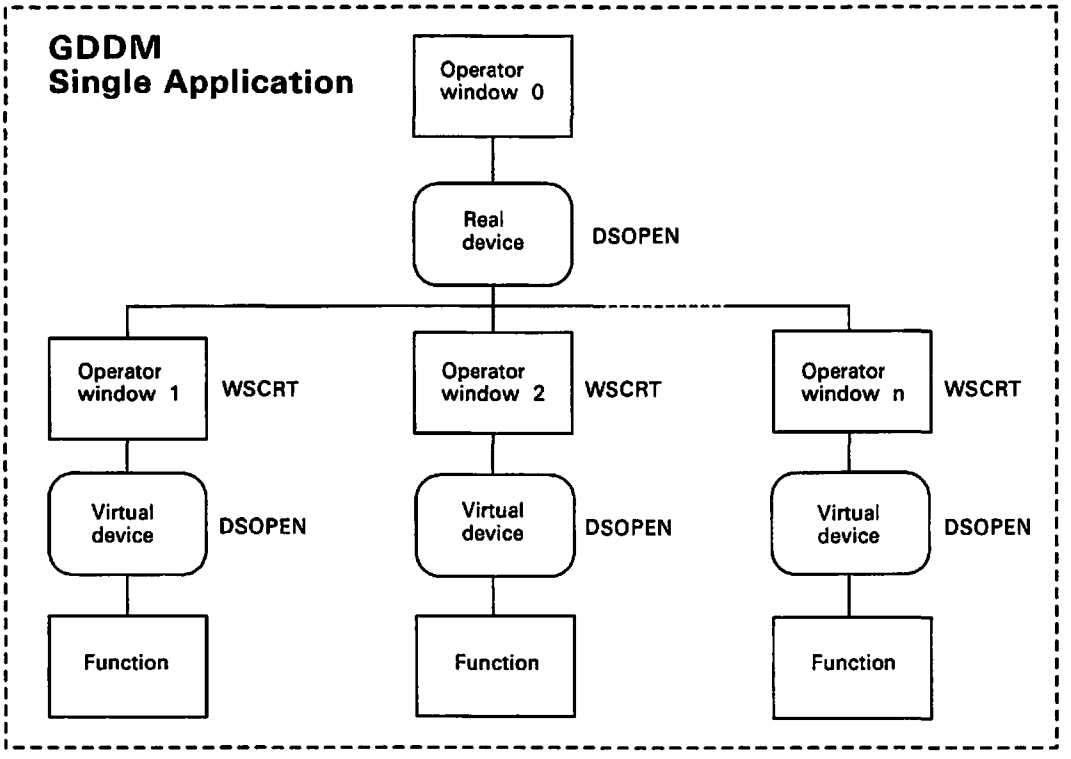


Figure 121. Hierarchy of devices and windows in a single application

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 *GDDM Guide for Users*.

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.

Sample program using one operator window

The following sample program is an example of 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);

/*****************************************************************/
/* 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;
```

The sample program, 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 12. Storing graphics" on page 157 and "Chapter 13. Picture handling in graphics data format" on page 171 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 "Processing options for operator windows" on page 386 and "Processing options for user control" on page 386.

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 sample program.

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 477. 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 will be 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 sample program, 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 481.

  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.

Sample 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
- 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 ENTER. 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,30,'GDDM Sample Program - Window 2');     /*B*/
                                   /* Create window 2           */

 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,30,'GDDM Sample Program - 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);

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;
```

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 478.

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 will be 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 will be 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 will be 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) will be 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 will therefore initially be 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 478.

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 will return 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 will return 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 sample program, 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 sample program, 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 481.

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 will be placed behind the reference operator window.

  `1` Ascending order (as in the example). The operator windows in the array will be 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 will become 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 will become 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 will be the top operator window in the viewing order, and the rest of the operator windows in the array will be those that are behind it.

  Ascending The first operator window in the array will be the bottom operator window in the viewing order, and the rest of the operator windows in the array will be 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 will return 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 Programming Reference* manual 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 "Sample program using two operator windows" on page 473 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 will 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 sample task manager (ADMUTMT for MVS/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 DSOPENs **in each application program** will open one or more virtual devices, which will be associated with the operator windows created by the task manager. This is illustrated in Figure 122 on page 482.

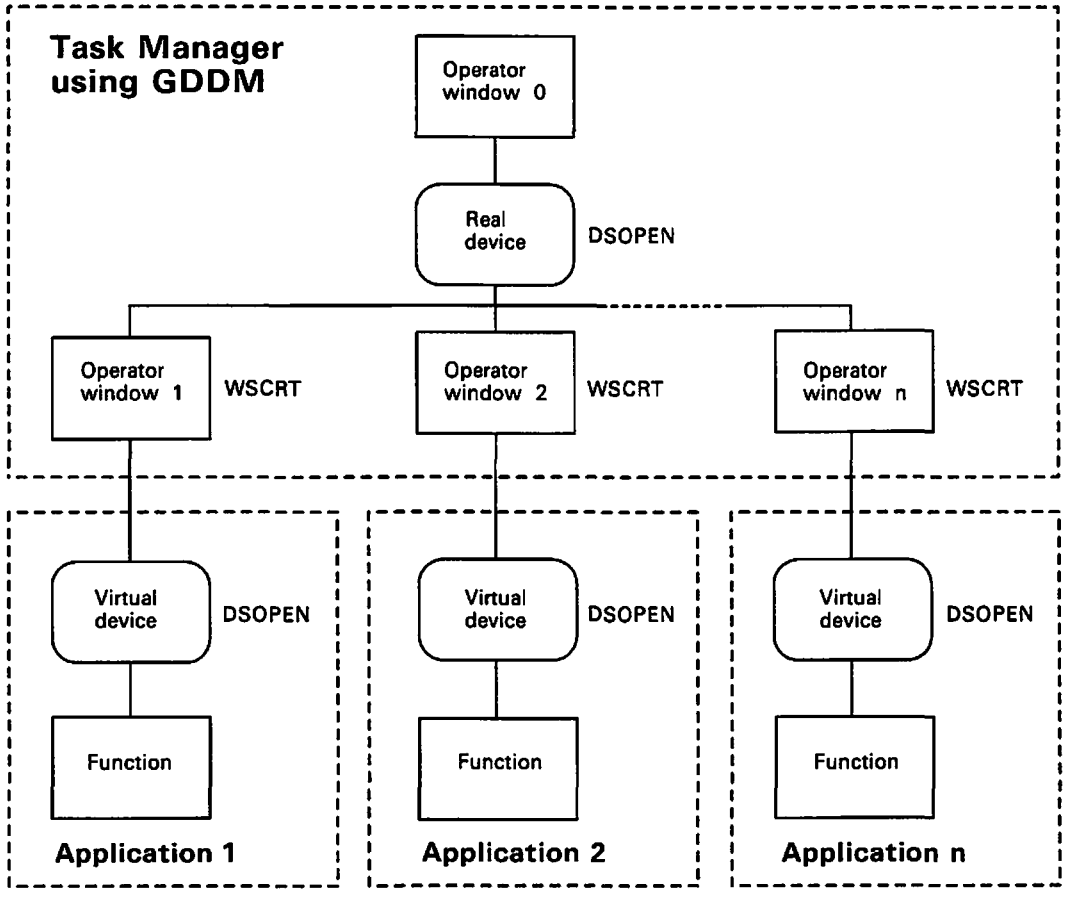


Figure 122. Task manager with several applications

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 pseudo-tasking facilities (TSO has full task-management facilities but CMS does not). The system tasking or pseudo-tasking executes each program in a separate sub-task.

The way that GDDM makes it possible for several application programs to share the screen is 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 123 on page 483.

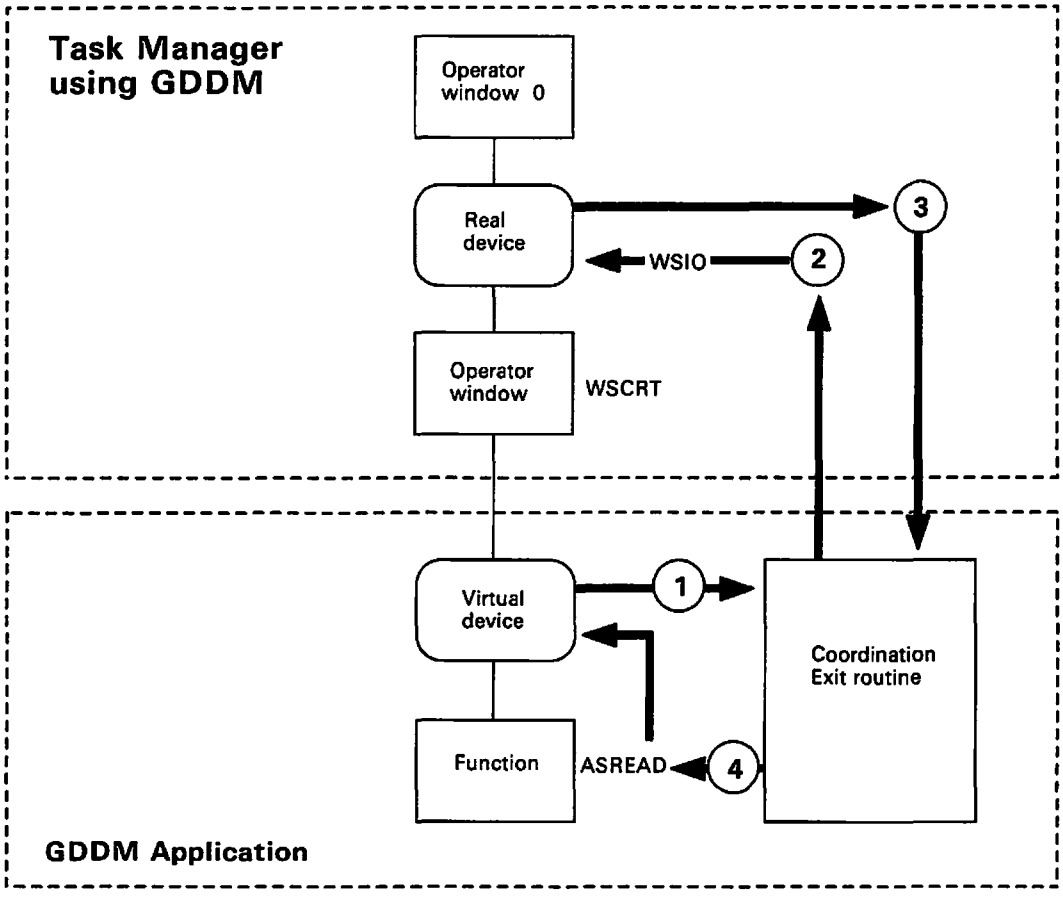


Figure 123. The coordination exit routine

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 will complete the processing of this call, and pass control back to the application program. Any input data entered by the terminal user will then be 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 will, when link-edited, attempt 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 will ensure 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 will be 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 Programming Reference* manual for a full description of each call.

Part 6. Example programs

Example 1. The ADMUSP4 graphics editor sample program

The ADMUSP4 sample program provides an example of a graphics editor program. It uses function and devices that are supported in GDDM Release 2.1 and is designed to be run on a 3270-PC/G or 3270-PC/GX work station. Its purpose is to let the user create pictures that are made up of one or more objects (for example, lines, shaded boxes, and strings of text).

The program is an example of the programming techniques that GDDM Version 2 Release 1 provides. Although it is written in PL/I, it can be used as the basis for user-written programs of a similar type in other programming languages.

Because of its length, the program listing is not given in this section. However, the remainder of this section summarizes what the program does and what is displayed when it is called.

Note: ADMUSP4 can be run on a 3279 display; however, it provides less function than when run on a 3270-PC work station.

What ADMUSP4 provides

Figure 124 shows the format of the display that the sample program provides.

Figure 124. The menu displayed by the ADMUSP4 sample program

The screen is divided into:

- An information area, that will contain 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 restore already saved graphics.
- A menu area, where 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 stylus tip-switch on the tablet, or by pressing enter, depending on the device.

The menu offers the following functions:

Global actions

Load	The user is prompted to enter the name of an ADMGDF file, which is loaded and displayed in the drawing area.
Save	The user is prompted to enter the name under which the contents of the drawing area is to be saved as an ADMGDF file.
Clear	Erases everything in the drawing area.
Plot	Sends the contents of the drawing area to an attached plotter.
Print	Sends the contents of the drawing area to a graphics printer (using a nickname).
View	Allows you to look at the picture in the drawing area, without the menu and border lines.
Finish	Terminates ADMUSP4. If there are unsaved changes, the picture is preserved in a file called ADMUSP4 ADMGDF.
Cancel	Cancels a drawing action (for example, box).

Drawing actions

Sketch	Freehand sketching using a mouse or puck (not available on the 3279).
Line	Draws straight lines in the selected color, line type, and line width.
Box	Draws rectangular boxes in the selected color, line type, and line width.
Point	The user defines a series of points, using a mouse or puck. The points are joined by straight lines in the selected color, line type, and line width (not available on the 3279).
Arc	The user defines the start, end, and midpoints through which an arc is to pass, in the selected color, line type, and line width.
Circle	The user defines the center of, and a point on the circumference of a circle.
Rubout	Not supported.
Text	Writes text in the current color and style. (A warning is issued on the 3279).

Actions on drawn objects

Move	Repositions objects or groups of objects.
Copy	Copies objects or groups of objects.
Delete	Deletes objects or groups of objects.
Scale	Enlarges or reduces objects or groups of objects.
Rotate	Rotates objects or groups of objects.
Group	The user specifies a set of objects to be treated as one entity.
Undo	Not supported.

Style selection

Text	Four fonts.
Line width	Two line widths
Line type	Seven line types
Pattern	Sixteen shading patterns in any of the colors listed below.
Color	Blue, red, pink, green, turquoise, yellow, neutral, and background.

Invoking ADMUSP4

There are no special considerations for compiling, link editing, and running this sample program; if necessary, see "Chapter 2. Drawing a simple picture" on page 7 or the appropriate chapter in the *GDDM Base Programming Reference* that describes how to run GDDM under the subsystem in use.

Example 2. Assembler language 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
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
```

Example 3. An APL2 example

There is a function called GDMX supplied with APL2 Release 2. 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 GDMX function supplied with APL2 Release 2
[2]   'GSSEG' GDMX 1              ⍝ Open segment 1
[3]   'GSMOVE' GDMX 50 50         ⍝ Initialize current position
[4]  LOOP:'ASREAD' GDMX ''        ⍝ Wait for operator action
[5]   →(1 3∧.=2↑5↓1⊃RET_G)/END    ⍝ Check for PF3
[6]   'GSQCUR' GDMX ''            ⍝ Find where cursor is
[7]   'GSLINE' GDMX ¯2↑1⊃RET_G    ⍝ Draw line from previous point
[8]   →LOOP                       ⍝ Loop back
[9]  END:'GSSDEL' GDMX 1          ⍝ Delete segment 1
    ∇
```

Example 4. BASIC 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 will 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 ADMGPLIB
0110 REM  Also, if your default CMS LDRTBLS is less than 5,
0120 REM  you will 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, we have equated these
0160 REM  numbers 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
```

Example 5. CICS pseudoconversational example

The following example 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 will be displayed first, and PF Keys 3 or 4 entered from */
/* this panel will cause the end of the application with either  */
/* an erased screen or not respectively.                         */
/* Entering options '1', '2' or '3' from MENU00 will cause 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 will be 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 WE DO NOT HAVE A COMMAREA, 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                                       /* WE HAVE A COMMAREA         */
  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

Appendix A. Major types of supported device

Although GDDM programs are largely device-independent, the functions provided are inevitably influenced to some extent by the hardware. Some GDDM functions are not supported on some devices. In other cases, the results vary somewhat from one device to another. The purpose of this appendix is to help you understand the GDDM support for the particular devices used by your application programs.

Note: With all devices, some models only may be supported, and specific features may be prerequisite. Full details are given in the *GDDM Installation and System Management* manual.

3179-G display station

The 3179-G accepts graphics orders such as "draw a line" or "set color to red." The vector-to-raster process is performed in the terminal.

The 3179-G is a family-1 device.

3270-PC/G and /GX work stations

After the 5080 Graphics System, these are the most powerful terminals supported by GDDM. Of all display devices, they have the widest range of available function.

Like the 3179-G, they accept graphics orders. In addition, they accept alphanumerics data formatted into 3270 fields.

The 3270-PC/GX can be configured as a dual-screen version, on which the alphanumerics are displayed on a separate screen from the graphics. Application programs need not be concerned with this unless they depend on the layout of alphanumerics and graphics together. Otherwise, a program that executes correctly on the single-screen work station will also do so on the dual-screen configuration, and conversely.

Not only do the 3270-PC/G and /GX work stations support the most function, but for many applications they give the lowest usage of the host processor and the shortest data streams.

The work stations are GDDM family-1 devices.

Retained and non-retained modes

In the normal mode of operation, GDDM sends graphic orders to the work station 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 work station 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 work station with a request for immediate execution, and hence immediate display of the picture. The work station retains no graphics orders. This is non-retained mode.

If the program amends non-retained graphics, GDDM may have to retransmit the whole picture, whereas in retained mode it could transmit just the updates. The former results in longer data streams. Furthermore, some work-station functions that require retained graphics will be wholly or partially unavailable in non-retained mode. These include segment dragging – the default cursor will be used as the locator echo instead of the segment. Some work-station functions such as Jump or Change Screen, may result in graphics being missing from the screen until the host retransmits the picture.

Retained mode is the default. GDDM will degrade from retained to non-retained mode if necessary, with no action by you. Or, instead, you can specify non-retained mode in a processing option. The terminal can also be configured as "output only", which has the same effect as the non-retained-mode processing option.

When GDDM degrades, it initially attempts to use non-retained mode for the graphics segments, but to keep retained mode for the symbol-set definitions. If there is not even enough storage for just the symbol sets, the symbols are expanded into primitives, and these are transmitted in non-retained mode. When sufficient storage becomes available at a later stage, GDDM upgrades back to retained mode.

No attempt is made to use retained mode if the total segment storage available to GDDM is less than 1024 bytes.

Non-retained mode is used when more than one graphics field is displayed on the screen. It is also used for window borders. Local functions such as Jump Screen should be avoided when operator windows are used with multiple applications.

Data from stroke devices is held in segment storage. The complexity of the picture may therefore affect the number of stroke-data points that can be held.

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 non-retained mode. And applications that create complex pictures and have little graphics-input function may be most efficient in non-retained mode.

You can specify non-retained 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/NON-RETAINED MODE PROC. OPTION   */
PROCOPT_LIST(2) = 1;      /* 0 = RETAINED (DEFAULT); 1 = NON-RETAINED  */

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, as it allows the operator to select the mode to suit the terminal being used.

5550 family multistations with 3270 PC/G program

The 5550 Multistation supports DBCS characters (used in some Asian languages) in alphanumeric fields. It can include a 5550-family printer in addition to the display unit. The 5550 is roughly equivalent to a 3270-PC/G, but supports non-retained mode only. It accepts graphics orders. It supports alphanumeric data formatted into 3270 fields.

The 5550 is a family-1 device.

5080 Graphics System

The 5080 is the most powerful terminal supported by GDDM. However, it is primarily intended for polyline CAD/CAM applications. GDDM applications will run on the 5080 but will not take advantage of its full capabilities.

The 5080 is supported under TSO and VM, but not under CICS or IMS. GDDM/MVS, GDDM/VM, and GDDM-PGF communicate with the 5080 through GDDM/graPHIGS, a separate IBM licensed program.

The 5080 is either associated with a physically separate 3270, or with the 3270 feature that resides within the 5080 hardware, to give a dual-screen configuration. In the case of the physically-separate 3270, alphanumerics are displayed on the 3270 screen, while graphics are displayed on the 5080 screen at the same time. In the case of the 3270 feature, either graphics or alphanumerics are displayed on the 5080 screen. You cannot see them at the same time.

3270-family terminals that use programmed symbols for graphics

These include the 3278, 3279, 3290, 8775, and the 3270 PC.

They have less graphics capability than the 3270-PC /G and /GX work stations. The main differences are:

- These 3270 terminals do not accept graphics orders. GDDM displays graphics using programmed-symbol (PS) stores instead.
- They are capable of fewer processes than the 3270-PC/G and /GX work stations. Some operations, such as clipping, have to be done in the host rather than the terminal. Others, such as segment dragging, are not supported at all.
- They do not have a graphics cursor, nor any associated graphics input device (that is, no mouse, stylus, or puck).

The terminals are GDDM family-1 devices.

How graphics are created using programmed symbols

The display area of these terminals is divided into cells. The PS feature permits any desired pattern of dots, or pixels, to be sent to any of the cells on the device. Graphics are created by displaying appropriate patterns in appropriate cells.

When you request a thick blue line for example, GDDM will determine which hardware cells the line will traverse. It will then determine what dot patterns are required in these cells to produce a thick blue line. The process of translating graphical items into dot patterns is known as **rastering**. When the time comes to send out the graphics, GDDM will construct a data stream to transmit the dot patterns (that is, the programmed symbols) to the device and to display them in the requisite cells.

Multicolor devices have two different types of PS-stores. Those that may contain multicolored programmed symbols are known as **triple-plane** PS-stores. Those that permit only monochrome programmed symbols are called **single-plane** PS-stores. Monochrome symbols may be displayed in any one of the seven colors: the term means that only one color is used within the symbol.

GDDM uses both types of store to hold the PS that make up the graphics. For example, if you draw a red line crossing a blue line, the dot pattern for the cell containing the crossing point would need to be loaded into a triple-plane PS-store. The dot patterns representing the remainder of the two lines could be loaded into a single-plane PS-store.

A 3279 terminal has up to six PS-stores, depending on its features. Numbers 2, 3, and 6 are single-plane. Numbers 4, 5, and 7 are triple-plane.

For the allocation of PS stores to a number of operator windows, see "Allocation of resources to operator windows" on page 484.

PS overflow – corruption of the display output

Under some conditions there may be insufficient PS-store locations available to hold all the dot patterns needed to represent your graphics. This causes a phenomenon known as **PS overflow**. The symptoms of this may be twofold:

- If the device has exhausted its triple-plane locations, but has some single-plane locations available, some graphics cells will appear with the correct dot patterns but an incorrect color.
- If the device has exhausted both types of PS location, some graphics cells will contain an asterisk instead of the requisite dot pattern.

PS locations are likely to become exhausted only when there is a large number of **different** dot patterns in use, because GDDM shares locations when a pattern is repeated. These are the conditions which might cause overflow:

- An unusually high number of multicolored dot patterns are required to represent the graphics. The triple-plane PS-stores may become exhausted.
- One or more of the PS-stores have been reserved for font usage as described in "Chapter 15. Symbol sets" on page 219. They are therefore unavailable for GDDM's use.
- The picture is too large and too complex.

You can check whether a picture would cause PS overflow before sending it to the terminal by executing an FSCHEK call (see "Checking picture complexity using call FSCHEK" on page 15).

You can always reduce the required PS-storage by making your picture smaller. In other words, you reduce the size of your graphics field (see "The graphics field" on page 96).

3270-family terminals without programmed symbols

These may include the 3104, 3178, 3275, 3276, 3277, 3278, 3279, 3290, 5550 with Japanese 3270 PC program or 3270 emulation program, 8775, and 3270 PC. When programmed symbols are not available, only alphanumeric functions are supported.

All these devices are family-1.

3117 and 3118 scanners, and the 3193 display station

The 3193 is the primary display for images. As well as displaying image data, it can perform some image processing.

The 3193 also supports alphanumeric input and output, using an alphanumeric cursor.

Although GDDM graphics are valid on the same page as image, graphics cannot be shown on a 3193.

The 3117 and 3118 scanners attach to the 3193. Both devices scan a document and convert it into electronic image data.

3270-family graphics printers

These are the 3268 and 3287. They support graphics using programmed symbols and alphanumerics.

Some models print in monochrome only. Others will print in four colors. On these, black and the primary colors (red, blue, and green) print as specified by your program, and all other colors print as black.

It is rare to get PS overflow on a printer. The output is sent in bands (known as **swathes**). If ever the PS becomes exhausted, GDDM discards it and starts a new swathe. This technique is not available on a display, where removal of the PS for the first part of the picture would cause its disappearance from the screen.

These printers can be treated as GDDM family-1 or family-2 devices.

3270-family alphanumeric printers

These include the 3230, 3232, 3262, 3283, 3284, 3286, 3288, 3289, and 5550-family printer, such as the 5553, 5557, and 5577. They will print alphanumeric data only. They can be GDDM family-1 or family-2 devices.

The 3262 can also be used as a system (family-3) printer.

System printers

These include the 1403, 3203, 3211, and 3800. They support alphanumerics only. In GDDM terms, they are family-3 printers.

Some models of the 3800 can also be used as composed-page (family-4) printers.

Composed-page printers

These are the 3800 Models 3 and 8, the 3820, and the 4250. They are high-resolution devices that accept coded bit images of graphics output. Alphanumerics are not supported.

They are GDDM family-4 devices. The output from your program goes first to an intermediate image file. This may be merged with other output (typically text from the IBM Document Composition Utility program product) when it is sent to the printer.

The printer's output is monochrome. However, GDDM supports the creation of color-separation masters for publications printing.

Plotters

GDDM will send graphics output to the following IBM plotters:

- 7371, 7372, 7374, 7375, or 6180, attached to a 3270-PC/G or /GX family work station
- 7371, 7372, or 6180, attached to a 3179-G
- 7371, or 7372, attached to a 5550-family multistation.

In each case, alphanumerics are not supported.

The plotters are GDDM family-1 auxiliary devices.

IPDS printer

This is the 4224 printer. It supports alphanumerics, graphics, and image data. The graphics data are sent as vectors and the printer performs the vector-to-raster conversion. It can print in monochrome, 4 colors (blue, red, green, and black) or 7 colors (blue, red, magenta, green, cyan, yellow, and black) depending on the type of ribbon installed.

The extended storage model contains 512K bytes of RAM for holding picture data sent by GDDM. The base model contains 64K bytes for this purpose. If the picture data exceeds the storage capacity of the target printer, a warning message is issued, and transmission of data is truncated at that point.

It can be used as a GDDM family-1 or family-2 device.

Appendix B. Device-independent programming tips

Introduction

GDDM provides much of its function in a device-independent way. However, there are two kinds of device-dependent operation of some Base API calls:

1. The 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 device capabilities, so it follows that only some devices allow the full range of some call parameter values to take effect.

   For example, color in the GSCOL call.

2. The call is not valid for the current device, and a GDDM error message will be displayed.

   For example, use of alphanumerics on a plotter.

The points below are aimed to make you aware of potential situations in the first category, and to help you avoid the second.

See Appendix A, "Major types of supported device" on page 507 for further device-specific information.

Points to help you minimize device dependency in your programs

Before going into the detail below, remember two general tips:

1. Use GDDM **defaults** as much as possible, and

2. Make good use of the Base API **query calls**. Not all of them are mentioned below – see the *GDDM Base Programming Reference* for a full list. Also see that manual for the considerable scope of the FSQURY call, which can be invoked so as to return general, graphics, partitions, or plotter-related information.

Specific points are now discussed:

Graphics primitives

- Do not use primitives outside segments if you want them plotted. You should also note that the lifetime of primitives outside segments is device-dependent.
- Note that GSIMG image objects will vary in size because of different resolution (pixels per inch) on different devices.

Graphics attributes

- When using GSCOL, bear in mind that different numbers of colors are supported on different types of display (for example, the 3270-PC/GX supports 16), and frequently less are supported on plotters and printers (for example, the 3287 supports 4).

  See the *GDDM Base Programming Reference* for a full description of supported values.

- When using GSFLW, note that only the 3800 and 4250 support line width values less than 1 or greater than 2.

  See the *GDDM Base Programming Reference* for a full description of supported values.

- When using GSMIX, use only overpaint (the default).

  Mix mode is not supported on the 5080.

  Underpaint is not supported on the 3270-PC/G or /GX, or the 5550.

  Mix mode should not be used on plotters, unless you genuinely want the undefined color resulting from mixing of the pen inks.

- Note that plotter patterns are a fixed set different from the GDDM- supplied patterns appearing on displays.
- Multi-colored shading is not supported on the 5080.

Displaying text

- When using procedural alphanumerics, avoid field positioning which may exceed the page limits on the current device (see "Graphics hierarchy" on page 515).
- Avoid mixing alphanumeric text accurately spaced with respect to graphics. Instead use graphics text, mode 3, especially if the output could appear on a plotter, in addition to a display.

  Use of GSARCC can help – it allows either graphics aspect ratio or graphics/alphanumeric positioning to be maintained.

- When using GSCB to specify character box sizes, use mode 3 graphics text.
- Note that image symbols will vary in size due to different resolution (pixels per inch) on different devices.

Text input

When using the 3279 as well as 3270-PC/G or /GX, do not use string input.

Graphics hierarchy

- Limit page size to row and column limits of the current device – use FSQURY to determine these dynamically.

  In general avoid enforcing specific rows and columns when specifying the hierarchy objects.

- Note that graphics field position and size, in rows and columns, may be obtained by using the GSQFLD call.

- Use the GSUWIN call to enforce uniform world coordinates in x and y directions. The resulting world coordinate ranges for the current window may then be queried by use of GSQWIN, and for a graphics cell by use of GSQCEL.

Storing and loading graphics

- Use GSSAVE and GSLOAD rather than FSSAVE and FSSHOR.

- When using the GSSAVE call, use the default for coordinate data type (floating point).

Interactive graphics

Note that:

- The 3179-G, 3279, and 5550 do not support segment dragging.

- The 3179-G, 3279, and 5550 do not support string and stroke devices.

- The 3179-G, 3279, 5550, and 5080 do not support choice device data keys.

- Only the 3270-PC/G and /GX, and 5080 support the tablet input device. The 3179-G and 5550 support only the mouse. The 3279 does not support either tablet or mouse.

Symbol sets

- Note that image symbols, being defined in actual pixels, will change in size and aspect ratio when displayed on different devices. This is the case whether the symbol set is loaded with a PSLSS or GSLSS call, and whether or not the default symbol set is used.

- When using supplied symbol set names, use the substitution character facility.

- Before specifying specific PS stores in the PSLSS call, query the number, type, usage and availability of PS stores by use of FSQURY and PSQSS. Otherwise code PSLSS(0,.....) so as to be non-specific.

Device support

- Specify the minimum in DSOPEN – use nickname files to set and change processing options.
- Use the DSQDEV call to find out which processing options are in effect, including the effects of any nickname file processing.
- Use the DSQUSE call to find out the identifier of the current primary or secondary device.

Windowing

Hardware partitions are supported on a limited number of displays. Emulated partitions are supported on all displays. You should therefore avoid any dependency on hardware partitions, if possible.

GDDM glossary

This glossary defines various terms used in the documentation of GDDM.

This glossary includes terms and definitions from the *IBM Vocabulary for Data Processing, Telecommunications, and Office Products*, GC20-1699.

A

AAB. Application anchor 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**.

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.

ADS. Application data structure.

AIC. Application interface component.

AID. Attention identifier.

alphanumeric character attributes. In GDDM, comprise the highlighting, color, and symbol set to be used.

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**, and **image field**.

alphanumeric field attributes. In GDDM, comprise intensity, highlighting, color, symbol set to be used, field type, field end output conversion, input conversion, translate table assignment, transparency, field outlining, and mixed-string fields.

alternate device. In GDDM, a device to which copies are sent of the primary device's output. 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.

APAR. Authorized program analysis report. A report of a problem caused by a suspected defect in a current unaltered release of a program.

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 programming-language interface between an IBM system control program or licensed program and its user.

area. In GDDM, a graphics area is 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.**

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.

B

background color. Black on a display, white on a printer. The initial color of the display medium. Contrast with **neutral color.**

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.**

blank character. An empty character represented by X'40' in the EBCDIC code. Such a character occupies one position and can be used for positioning purposes. Contrast with **null character.**

BMS. Basic Mapping Support (CICS/VS).

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

CCW. Channel command word.

CDPF. Composed Document Print Facility.

candidate operator window. The operator window with which a subsequently opened virtual device is associated.

cell. See **character cell.**

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 IBM 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 **chart area.** The size of the grid determines the basic size of the characters in all text constructed by PG 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/VS. Customer Information Control System/Virtual Storage. 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 IBM 4250 printer.

code point. Synonym for **character code**.

Composed Document Print Facility. An IBM licensed program for processing documents destined for the 4250 composed-page printer.

composed-page image file. An intermediate form, residing on disk, of a picture destined for a composed-page printer.

composed-page printer. A printer, such as the IBM 3800 Model 3 or 4250, to which the host computer transmits data in the form of a succession of formatted pages. Such devices can print pictorial data and text, and will position all output to pixel accuracy. The pixel density and the general print quality both often suffice as camera-ready copy for publications.

composed-page printer format. A general term describing the format of print data destined for output by using either **CDPF** or **PSF**.

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.

coordinating device. In GDDM, a real or virtual device, opened for operator windowing. It coordinates the sharing of the real device.

correlation. The translation (by GDDM) of a screen position into a part of the user's picture. The action following a pick operation.

current operator window. In GDDM, the operator window whose attributes can be modified.

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 will also be the start of the next primitive.

cursor. A physical indicator that can be moved around a display screen. See **alphanumeric cursor** and **graphics cursor**.

D

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.

DCT. Destination control table (CICS/VS).

default value. A value chosen by GDDM when no value is explicitly specified by the user. For example, the default line type is a solid line.

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 options**. For example:

- Family 1: 3270 display or printer
- Family 2: queued printer
- Family 3: system printer (alphanumerics only)
- Family 4: composed-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).

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. Usually includes a keyboard and always provides a visual presentation of data. For example, an IBM 3179 display.

DL/1. Data language 1. A language for data-base 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. In GDDM, characters that each occupy two bytes in internal storage and in display buffers. They are used to display **Kanji** or **Hangeul** symbols.

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. Used for oriental languages; for example, **Kanji** or **Hangeul**.

DPCX. Distributed Processing Control Executive. An 8100 system control program.

DPPX. Distributed Processing Programming Executive. An 8100 system control program.

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

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.

extended data stream. For 3179, 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

Farsi. Pertaining to the character set for the Persian language.

FCT. File control table (CICS/VS).

field. An area on the screen or the printed or plotted page. See **alphanumeric field**, **graphics field**, **image field**, and **mapped field**.

field attributes. See **alphanumeric field attributes**.

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.

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**.

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.

GDDM/graPHIGS. A member of the GDDM family used for creating hierarchical three-dimensional structures on the IBM 5080 Graphics System. It is based on the proposed ANSI standard for the Programmer's Hierarchical Interactive Graphics System (PHIGS).

GDDM-IMD. GDDM-Interactive Map Definition. See **Interactive Map Definition**.

GDDM-PGF. GDDM-Presentation Graphics Facility. See **Presentation Graphics Facility**.

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, 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.

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, comprise 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**, and **image 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, comprise 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.**

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.**

host. See **host computer.**

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 symbol. A character or symbol defined as a dot pattern.

Image Symbol Editor (ISE). A GDDM-supplied interactive editor that lets users 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).**

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/VS.

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.

IPDS. Intelligent printer data stream.

ISE. Image Symbol Editor.

ISPF. Interactive System Productivity Facility.

ISS. Image symbol set.

J

JCL. Job Control Language.

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.

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-attached. Pertaining to devices that are connected to a controlling unit by a data link. Synonymous with **remote.** Contrast with **channel-attached.**

link edit. To create a loadable computer program by means of a linkage editor.

load module. A program unit that is suitable for loading into main storage for execution; it is usually the output of a linkage editor.

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.

LTERM. In IMS/VS, logical terminal.

M

map. A predefined format of alphanumeric fields on a screen. Usually constructed outside of the application program. See **Interactive Map Definition.**

mapgroup. A data item that contains a number of maps and information about the device on which those maps will 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.**

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 operator can make a selection. Sometimes called a menu panel.

menu-driven. Describes a program that is driven by operator response to one or more displayed menus.

MFS. Message format service.

mixed character string. A string containing a mixture of **Latin** (one-byte) and **Kanji** or **Hangeul** (two-byte) characters.

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 hand-held device (the IBM 5277 Mouse) that is moved around a locator pad to position the graphics cursor on the screen.

MSHP. Maintain system history program.

MSL. Map specification library.

MVS/XA. Multiple Virtual Storage/Extended Architecture.

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) feature. The translations of the ICU panels and some of the GDDM messages into a variety of languages other than American-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.**

nickname. In GDDM, a quick and easy means of referring to a device, the characteristics and identity of which have been predefined.

NLS. National Language Support.

non-paired data. See **tied data.**

null character. An empty character represented by X'00' in the EBCDIC code. Such a character does not occupy a screen position. Contrast with **blank character.**

O

object code. Output from a compiler or assembler that is in itself executable machine code or is suitable for processing to produce executable machine code.

object deck. Synonym for **object module.**

object libraries. An area on a direct access storage device used to store object programs and routines.

object module. A module that is the output of an assembler or a compiler and is input to a linkage editor. Synonymous with **object deck.**

off-point. A pixel that has been turned off by the user of the Image Symbol Editor.

on-point. A pixel that has been turned on by the user of the Image Symbol Editor.

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 having to examine every byte.

overlapping bar chart. A form of business chart where adjacent bars partly overlap each other. Overlapping bars are sometimes called **hidden bars**.

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.

pageable (main storage). In a virtual storage system, fixed-length blocks that have virtual addresses and can be transferred between real (main) storage and auxiliary storage.

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. Synonymous with **program library**.

PCB. Program communication block (IMS/VS).

PCT. Program control table (CICS/VS).

PDS. In TSO, a partitioned data set.

pel. Synonym for **pixel**.

PGF. Presentation Graphics Facility.

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 element. Synonym for **pixel**.

picture interchange format (PIF) file. In graphics systems, the type of file, containing picture data, that can be transferred between GDDM and a 3270-PC/G or 3270-PC/GX work station.

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; when this happens, the list or sequence of data is said to be "popped."

ppi. Pixels per inch.

PPT. Processing program control table (CICS/VS).

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.

Presentation Graphics Facility (PGF). A member of the GDDM family of licensed programs. It is concerned with business graphics, rather than general graphics.

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 utility. A subsystem-dependent utility that sends print files from various origins to a queued printer.

Print Services Facility. An IBM licensed program for processing documents destined for the 3800-3 composed-page printer.

procedural alphanumerics. The creation of alphanumeric displays using the GDDM alphanumeric API. Contrast with **mapped alphanumerics.**

processing options. Describe how a device's I/O will be processed. These are device-family-dependent and subsystem-dependent options that are specified when the device is opened (initialized). An example is the choice between CMS attention-handling protocols.

procopt. Processing option.

program library. (1) A collection of available computer programs and routines. (2) An organized collection of computer programs. (3) Synonym for **partitioned data set.**

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. In IMS/VS, a program specification block.

PSF. Print Services Facility.

PTF. Program temporary fix.

puck. Synonym for **four-button cursor.**

Q

QSAM. Queued sequential access method.

QTAM. Queued telecommunications access method.

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

RAS. Reliability, availability, serviceability.

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).

RCP. Request control parameter.

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.

roman. Relating to the **Latin** typestyle, with upright characters.

S

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 will be drawn, also the order in which they will be 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.

SMF. System management facilities.

SMP. System management program.

SNA. Systems network architecture.

source image. An image that is the data input to image processing or transfer.

SPI. System programmer interface.

SPIB. System programmer interface block.

stand-alone (mode). Operation that is independent of another device, program, or system.

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. Composed-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.**

systems programmer interface (SPI). The formally defined systems-level interface between an IBM system control program or licensed program, and its user.

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 which 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.

TCT. Terminal control table (CICS/VS).

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.**

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 OS/VS 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 default specification (UDS). The means of changing a GDDM default value. The 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.

VCNA. VTAM communications network application.

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.

Venn diagram. A form of business chart in which, in GDDM, two populations and their intersection are represented by two overlapping circles.

viewport. A subdivision of the picture space, most often used when two separate pictures are to be displayed together.

virtual device. In GDDM, a functional simulation of a real display device, associated with an operator window.

virtual screen. In GDDM, the screen of a virtual device. The presentation space viewed through an **operator window.** Contrast with **real device.**

VM/SP CMS. IBM Virtual Machine/System Product Conversational Monitor System. A system under which GDDM can be used.

VSE. Virtual storage extended. An operating system consisting of VSE/Advanced Functions and other IBM programs. 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.

VTAM. Virtual Telecommunications Access Method.

W

window. (1) In GDDM, a defined section of world coordinates. The window can be regarded as a set of coordinates that are overlaid on the viewport, and used for defining the primitives that make up a graphics display. By default, both x and y coordinates run from 0 through 100. (2) In GDDM, 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) In GDDM, the "page window" defines which part of a page which is deeper or wider than its partition should currently be displayed.

work station. 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.
_control parameter""""PGFPR"
_control parameter""""PGFPR"

Index

Special Characters

A

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

B

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

D

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

E

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

F

G

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

H

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

J

K

L

M

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

N

O

P

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Q

R

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

T

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

U

V

W

X

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Y

Z

Numerics

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

Page numbers refer to this book; further information can be found in other GDDM books:
APG2 *Application Programming Guide Vol 2* PGFPR *PGF Programming Reference*
BPR1 *Base Programming Reference Vol 1* BPR2 *Base Programming Reference Vol 2*

GDDM Application Programming Guide Volume 1

READER'S COMMENT FORM

Order No. SC33-0337-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Number of your latest Technical Newsletter for this publication . . .

Note: *Staples can cause problems with automated mail-sorting equipment. Please use pressure-sensitive or other gummed tape to seal this form.*

If you want an acknowledgement, give your name and address below.

Name . . . . . .

Job Title . . . . . Company . . . . . .

Address. . . . . .

. . . . . Zip . . . . . .

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC33-0337-0

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 6R1H,
180 Kost Road,
Mechanicsburg, PA 17055, USA

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

SC33-0337-0
Version 2 Release 1

GDDM Application Programming Guide Volume 1 Printed in USA SC33-0337-0

IBM®