SERIES 60 (LEVEL 68)

MULTICS GRAPHICS SYSTEM ADDENDUM A

SUBJECT

Additions and Changes to the Detailed Description of the Multics Graphics System, Including Details of the Commands and Subroutines Used to Create, Edit, Store, Display, and Animate Graphic Constructs

SPECIAL INSTRUCTIONS

This is the first addendum to AS40, Revision 1, dated December 1979.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Throughout the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of this manual.

Note:

Insert this cover after the manual cover to indicate the updating of the document with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 9.0

ORDER NUMBER

AS40-01A

August 1981

32498
5C981
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
title page, preface	title page, preface
iii through ix, blank	iii through x
2-1, 2-2	2-1, 2-2 2-2.1, blank
2-35 through 2-38	2-35 through 2-38
3-25, 3-26	3-25, blank 3-25.1, 3-26
3-29 through 3-38	3-29 through 3-38
3-43 through 3-46	3-43 through 3-46
4-7, 4-8	4-7, 4-8
4-19 through 4-22	4-19 through 4-22
5-21 through 5-24	5-21, 5-22 5-23, blank 5-23.1, 5-24
5-35 through 5-38	5-35 through 5-38
5-41, 5-42	5-41, 5-42
5-49, 5-50	5-49, 5-50 5-50.1, 5-50.2 5-50.3, 5-50.4
5-51, 5-52	5-51, 5-52
5-57 through 5-60	5-57 through 5-60
5-79, 5-80	5-79, blank 5-79.1, 5-80

File No.: 1L13

8/81

AS40-01A

5-83, 5-84	5-83, 5-84
5-95, 5-96	5-95, 5-96 5-96.1, blank
5-103, 5-104	5-103, 5-104
5-107, 5-108	5-107, blank 5-107.1, 5-108
6-1 through 6-4	6-1, 6-2 6-3, blank 6-3.1, 6-4
6-5, 6-6	6-5, 6-6
6-7, blank	6-7, 6-8
7-1, 7-2	7-1, 7-2
8-3, blank	8-3, blank
i-1 through i-4 i-5, blank	i-1 through i-4 i-5, blank

SERIES 60 (LEVEL 68)

MULTICS GRAPHICS SYSTEM

SUBJECT

Detailed Description of the Multics Graphics System, Including Details of the Commands and Subroutines Used to Create, Edit, Store, Display, and Animate Graphic Constructs

SPECIAL INSTRUCTIONS

This revision supersedes Revision 0 of the manual dated October 1977. Change bars indicate new and changed information; asterisks denote deletions.

Section 7 is completely new and does not contain change bars.

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

AS40-01

December 1979

Honeywell

PREFACE

This manual is a combined graphics primer and reference manual for the Multics Graphics System. The manual is intended for users familiar with the general characteristics of the Multics system, including the mechanics of terminal usage, and assumes the user has a basic understanding of the simpler features of the PL/I or FORTRAN Languages.

Section 2 of this manual is devoted to the primer. It is suggested that users unfamiliar with graphics read Section 2 to acquaint themselves with the Multics Graphics System before becoming familiar with the reference portion of the manual (basics of the Multics Graphics System such as structure, commands, subroutines, etc.) contained in Section 3 through 8. Periodic return to Section 2 examples while studying or referencing later sections will prove beneficial in clarifying actual use.

Other manuals that provide additional information and that are referenced in this manual include:

Document	Referred To In Text As
Multics Programmers' Manual (MPM)-- Reference Guide (Order No. AG91)	MPM Reference Guide
MPM Commands and Active Functions (Order No. AG92)	MPM Commands
MPM Subroutines (Order No. AG93)	MPM Subroutines
Multics PL/I Reference Manual (Order No. AM83)	PL/I Reference Manual
Multics FORTRAN (Order No. AT58)	FORTRAN Manual

The MPM Reference Guide contains general information about the Multics command and programming environments. It also defines items used throughout the rest of the MPM and, in addition, describes such subjects as the command language, the storage system, and the input/output system.

File No.: 1L13

8/81

AS40-01A

The MPM Commands is organized into four sections. Section 1 contains a list of the Multics command repertoire, arranged functionally. Section 2 describes the active functions. Section 3 contains descriptions of standard Multics commands, including the calling sequence and usage of each command. Section 4 describes the requests used to gain access to the system.

The MPM Subroutines is organized into three sections. Section 1 contains a list of the subroutine repertoire, arranged functionally. Section 2 contains descriptions of the standard Multics subroutines, including the declare statement, the calling sequence, and usage of each. Section 3 contains descriptions of the I/O modules.

The Multics PL/I Reference Manual, referred to in this book as the PL/I Reference Manual, is a combined tutorial and reference manual for Multics PL/I. It is very detailed and provides many examples of Multics PL/I language usage.

The Multics FORTRAN manual describes the Multics FORTRAN language, including fundamental concepts and definitions of the syntactic form and meaning of each language construct.

Changes to Multics Graphics System, Revision 1

In Section 2, new text describes "Programming Considerations."

In Section 3, "Arrays" describes treatment of arrays by the graphic compiler. Also, the description of "Graphic Device Table" is expanded to include GDT format, major and minor keywords and values. Although new to Section 3, "Format of a GDT" and "Graphic-Support Procedures" contain change bars only to denote new or changed information. "Graphic Character Tables" are now described in this section.

In Section 4, there is a new command, "compile_gct", and the "tmgc" command has been deleted.

In Section 5, three subroutines have been deleted:

graphic_char_table_
graphic_matrix_util_
make_graphic_array_

Section 6, "Graphic Device Table," has a description of one new GDT, "tek_4662".

Section 7 now contains the descriptions and displays of the "Graphic Character Tables."

Changes to Multics Graphics System, Revision 1, Addendum A

In Section 2, the description of "Programming Considerations" now includes a caveat for FORTRAN users. At the end of the section, new text describes the "Search List."

In Section 3, the "Specification of the Virtual Graphic Terminal" has been expanded. Changes have been incorporated into "Graphic-Support Procedures." The "Graphic Character Table (GCT)" now describes the handling of special format characters.

In Section 5, the graphic_gsp_utility_ subroutine is new and does not contain change bars. Changes have been made in the following subroutines: graphic_chars_, graphic_compiler_, graphic_error_table_, graphic_manipulator_, and gui_.

Section 6, "Graphic Device Table," describes a new GDT, rg512, which is the RetroGraphics 512 enhancement for the ADM3A terminal. The rg512 description is new and does not contain change bars. Changed GDTs are: tek_4002, tek_4012, tek_4014, and tek_4662.

CONTENTS

ILLUSTRATIONS

TABLES

SECTION 1

INTRODUCTION

The Multics Graphics System (MGS) provides a general purpose terminal-independent interface through which user or application programs can create, edit, store, display, and animate graphic constructs.

Primitives are provided for manipulating a structured picture description composed of lines, points, screen modes, rotations, translations (position shifts), and scalings, in three dimensions. Primitives are also provided for displaying parts of such a graphic structure, for animating an already displayed structure, for obtaining graphic input, and for controlling special terminal functions (such as screen erase). These primitives are suitable for direct use by a knowledgeable programmer.

The structured picture description interface primitives, in addition to being well-suited for a wide variety of graphic programming tasks, are also well-suited for use as building blocks by application modules that provide simpler or more application-oriented interfaces. Efficiency is enhanced by providing several alternate forms for storing graphic information that promote efficient editing of frequently changing graphic constructs and efficient storage and "play-back" of background scenes and standard display pictures.

The MGS is terminal-independent; that is, a graphic program written for one type of graphic terminal is operable on another graphic terminal of similar capabilities without modification. Users are not isolated by the particular type of graphic terminals used and can use graphic programs developed on different terminals by other users. They also are not restricted by their programs to particular terminal types, but can use any available graphic terminal. Graphics subsystems written for specific terminals can be easily transferred to new and better terminal types.

Terminal-independence is achieved in the MGS in the following way. The programming interface incorporates the most useful features of existing terminals and allows the addition of new features as graphic terminal capabilities evolve. Users tailor their programs to use the features of the terminal types intended for use. When the program is run, the use of any unavailable feature can be mapped by the system into the most reasonable compromise feature of the terminal being operated. Thus, users have a reasonable guarantee that their graphic programs will produce a recognizable picture on almost any type of graphic terminal connected to Multics. Of course, not all graphic programs will operate equally well on any type of graphic terminal (e.g., animation is not possible on a storage-tube terminal).

Although the MGS is discussed in this manual largely in terms of PL/I, the MGS is designed to be usable from FORTRAN, and for the most part, from many other Multics programming languages.

The following list identifies all abbreviations used in this document.

abbreviation	descriptive name
DPI	double precision integer (format)
GDT	graphic device table
GSP	graphic support procedure
MGS	Multics graphics system
MSGC	Multics standard graphics code
PGS	permanent graphic segment
SCL	scaled fixed point (format)
SPI	single precision integer (format)
UID	unique identifier (format)
VGT	virtual graphic terminal
WGS	working graphic segment

SECTION 2

GRAPHICS USERS' GUIDE

This section serves as a primer for the MGS. It contains supplementary explanations of the major graphics commands and subroutines, along with examples of their use.

The text closely follows the actual sequence of events in a typical session with the MGS. Although the exact command invocations and examples shown in this section may be duplicated for training purposes by a terminal user, they should not be interpreted as representing a rigid and necessary sequence of operations. Rather, each serves to outline the general function and typical usage of a command or subroutine. The user should examine the detailed description of the module in question in order to tailor the example to fit a particular requirement.

Likewise, the extent of the discussion of any particular module should not be taken to be an exhaustive description of its capabilities. Only those features that are deemed essential or instructive to the novice user of graphics are discussed in this section. Information necessary to exploit more advanced features of the MGS may be gained from the appropriate module description in Section 5, "Subroutines".

In most examples, the longest and most descriptive name by which a command or subroutine can be referenced is used for clarity.

At times, a command or subroutine is mentioned in the text without benefit of an example. In these cases, a reading of the appropriate module description in the Commands or Subroutines sections should be sufficient explanation of its use in that context.

Not all parameters to entries used in examples are explained in the text. Users may refer to the appropriate module description in Section 5, "Subroutines".

TYPICAL USE OF THE GRAPHICS SYSTEM

To use the MGS, a user must first log in on the Multics system from an available terminal. Of course, to produce graphics on a terminal, rather than to an offline device such as a plotter, it must be determined that the terminal chosen is capable of graphics.

The user must set up a graphic I/O environment, using the setup_graphics command as described later. This allows the graphics system to perform graphic I/O to the device, and informs it of the type of device being used.

At this point, the user may construct a program to perform graphics, use an already existing program, or use the graphics_editor.

Graphic programs may be written in most languages supported by Multics. These programs perform graphic operations by issuing calls to subroutines provided by the graphics system. The typical graphic program first constructs a graphic structure (builds the internal representation of the sequence of lines, points, etc., which are later to be displayed). When this has been done, the program calls the graphic_compiler_, which interprets this internal representation and causes it to be displayed on the graphic device. The user then typically goes through several iterations of correcting and re-executing the program until satisfied that it is correct. If the program is designed to construct a "canned" display, the user may save the result in any of several forms that allow recreation of the desired display without rerunning the program that created it.

The higher-level subroutines of the MGS such as gui_, calcomp_compatible_subrs_, and plot_, as well as the interactive graphics_editor, are simply graphic programs, much like any that a user would write. They use the primitive graphic subroutine calls provided by the central graphics system (which is also directly available to users) to perform their functions.

PROGRAMMING CONSIDERATIONS

When writing graphics programs, it is the responsibility of the user to understand the implications of choosing a particular programming language to perform the desired task. Although the MGS is designed to be usable by programs written in any Multics programming language, there are sometimes minor incompatibilities between other languages and PL/I that must be compensated for before correct results can be obtained from the MGS. Users are referred to the relevant reference manual and/or users' guide for the programming language of their choice. Two very common incompatibilities merit description here.

Many subroutine entrypoints (for example, in graphic_manipulator_ and graphic_operator_) return fixed bin (18). Since the prefix "graphic_" begins with the letter "g," the FORTRAN compiler (unless otherwise informed) assumes all these entrypoints return floating (real) quantities. It then tries to convert the return value from "float" to "fixed," and the result is usually zero. Thus, the FORTRAN user making this common error usually builds up a complete graphic structure containing nothing but null nodes, even though the error codes are zero. When displayed, this results in a screen erase and no picture, or an error message about "Node out of bounds" or "Node not a graphic datum."

To anticipate and eliminate this problem, entries that return fixed bin of any precision must be delcared in FORTRAN programs via:

```
integer graphic_manipulator_$create_list
```

Many entries in the MGS have declarations containing star-extents (e.g., "char(*)", or "dimension(*)"). FORTRAN users should be aware that the correct operation of these entrypoints depends on the calling program creating descriptors to accompany the arguments, which provide information on the length of the character string supplied or the bounds of the array supplied. The FORTRAN compiler does not provide descriptors by default. The user must specifically request the generation of descriptors via the external statement. (See the Multics Fortran manual, Order No. AT58 for further information.) For example, a FORTRAN program that calls graphic_compiler_$display_name must contain a statement of the form:

```
external graphic_compiler_$display_name (descriptors)
```

Failure to use this statement can result in obscure errors in the operation of the program.

BASIC GRAPHIC PREMISES

Before any graphic programming is attempted, the user must be aware of certain conventions of the MGS concerning screen size, terminal capability, available graphic elements, and allowable operations on graphic elements.

Parameters of the Graphic Display

The various graphic terminals and plotters that may be connected are defined in relation to a hypothetical graphic device called the Virtual Graphic Terminal (VGT). Although these devices may differ widely as far as origin positioning, screen or bed size, character codes required to perform graphic operations, and so on, the MGS defines each of these properties for the user in terms of the VGT. In this way, any graphic program may use any graphic device interchangeably, within reason. (Obvious exceptions include performing animation on a plotter, and so on.)

The VGT possesses a square screen, whose area is defined in units of "points". Points have no particular relationship to any real unit of measurement, but are defined solely by the size of the screen, which is by convention 1024x1024 points. This two-dimensional screen really represents a three-dimensional viewing space of 1024x1024x1024 points. The coordinate origin, (0,0,0), is defined to be in the center of the screen. Thus the screen coordinates extend from -512 to +511 in each dimension. Since many terminals actually have rectangular screen shapes, the largest (central) square area on the screen is used as the 1024x1024 area. Although the MGS may allow the user to display graphics outside this area, it does not guarantee device-independence in this case. If a user whose programs take advantage of leftover vertical space on the screen of one particular terminal type were to move to a terminal whose screen was oriented in the other direction, those parts of the display occupying the leftover space would be clipped off.

The axes used are standard right-handed cartesian axes. They are oriented so that the positive x direction extends to the user's right; the positive y direction upwards; and the positive z direction out of the screen towards the user.

Nomenclature of Graphic Elements

For the purpose of this preliminary discussion, graphic effectors[1] are divided into six classes: positional elements, mode elements, mapping elements, dynamic elements, structural elements, and miscellaneous elements. Dynamic effectors are not within the scope of this introductory material, and are described in Section 3.

The following short descriptions of graphic elements[1] are provided so that the reader may have a basic understanding of the terms used in this section. Formal definitions and full explanations of each element may be found in Section 3. In most cases, an example is given of the use of each of these elements later in this section.

---

[1] As used in this manual, the term "graphic effector" includes every basic item defined by the graphics system that can be used for graphic effect. "Graphic elements" are that subset of graphic effectors which can be used as building blocks of graphic structures to represent objects or pictures in a graphic fashion (e.g., lines, points, and character strings), as opposed to graphic effectors which cannot be included in structures, such as requests for graphic input and screen erasure.

The MGS defines a "graphic cursor" called the current graphic position. At the start of any graphic operation, the current graphic position (analogous to the "pen" or "beam position") is defined to be at the center of the screen (0,0,0). It may be modified in two ways:

- it can be set to a known position on the screen by the use of an absolute graphic element, or
- it can be moved a specified distance in a specified direction by a relative graphic element.

Absolute Elements

The MGS allows two absolute elements:

1. setposition: sets the current graphic position to a specified location on the screen. For example, a setposition of (-20,10,0) causes the current graphic position to be set to that coordinate location (see Figure 2-1, "Step 1").
2. setpoint: sets the current graphic position to a specified location on the screen, and in addition displays a visible point.

Relative Elements

Unlike absolute elements, relative elements do not start or end at any particular point on the screen. They start at the current graphic position and proceed for a specified distance. This distance, also expressed in coordinate notation, represents the dimensions of the element.

Three relative graphic positional elements are defined:

1. vector: draws a visible line of specified dimensions and moves the current graphic position to the end of that line. For example, a vector of dimensions (32,-40,0) draws a line from the current graphic position (of slope -5/4 and length 51.2) in the x-y plane (see Figure 2-1, "Step 2").

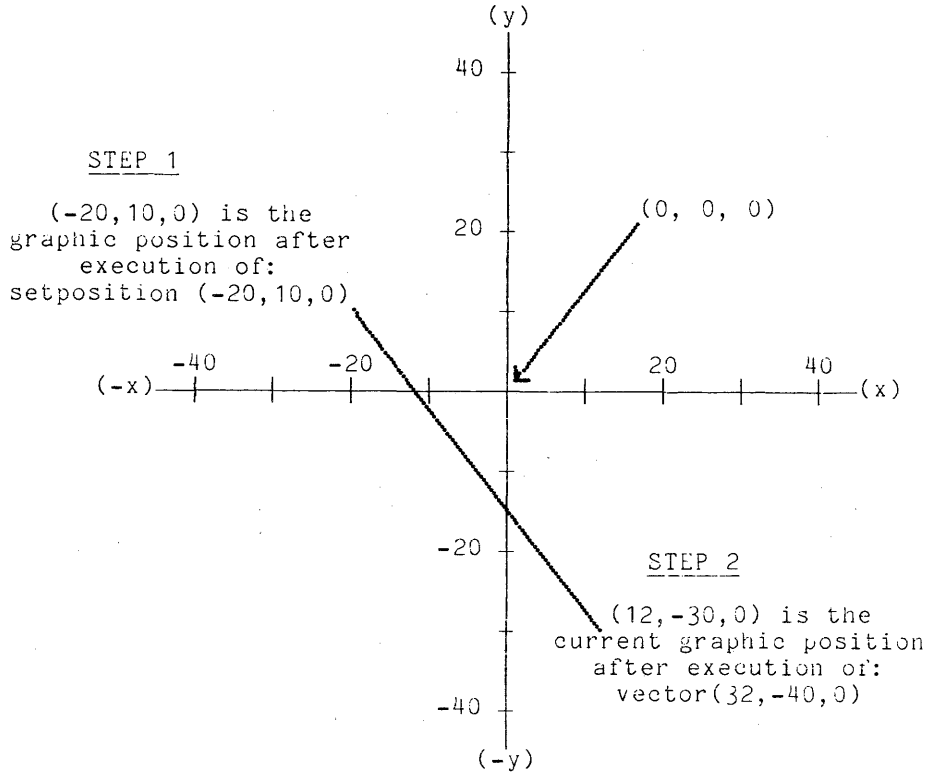


Figure 2-1. Current Graphic Position Modification

2. shift: is similar to the vector element, except that no visible line is drawn.

3. point: is similar to the shift element, except that a visible point is displayed at the new current graphic position (at the end of the movement).

Graphic structures made up entirely of relative elements possess a great deal of flexibility. Since all such elements are relocatable, they can be positioned anywhere on the screen with no loss of generality, as well as scaled or rotated without unexpected side effects.

MODE ELEMENTS

Mode elements may be used to alter the appearance of an object in certain defined ways without altering the object itself. They affect fundamental properties of objects such as intensity, linetype (e.g., dotted or solid), and color:

1. intensity: affects the brightness of an object.

2. blinking: affects whether an object blinks.

3. linetype: affects whether vectors in an object are drawn as solid, dotted or dashed.

4. sensitivity: affects whether an object is sensitive to "hits" from a light pen.

5. color: affects the color that a displayed object possesses.

MAPPING ELEMENTS

Mapping elements may be used to alter the appearance of an object in certain defined ways without altering the object itself. They affect the appearance of the object on the display screen with respect to orientation, size, and extents. They affect absolute elements as well as relative elements. The mapping elements are:

1. scaling: affects the size and proportions (independently in three dimensions) of a displayed object.

2. rotation: affects the orientation of a displayed object in three dimensions.

3. extent: affects the boundaries at which a displayed object ceases to be visible. The extent element performs clipping (causing parts of objects outside a given boundary not to appear) and masking (causing parts of objects inside a given boundary not to appear).

STRUCTURAL ELEMENTS

Structural elements are used to form groups of other elements. Both the array and list structural elements can be used to perform this function. The difference between them need not concern the user at this point, except to say that the computational overhead associated with the use of the array element is less than that associated with the list element. However, the list element is more useful when performing animation and dynamic graphics.

MISCELLANEOUS ELEMENTS

1. text: is used in a graphic structure to include a text string to be displayed.

2. symbol: is used to assign a mnemonic name to a graphic structure.

3. datablock: is used to include, in a graphic structure, user-defined data that is to be stored for later examination by the user, but which has no direct bearing on the graphical representation of the object to which it is attached.

SETTING UP THE GRAPHIC I/O ENVIRONMENT

Graphic input and output are not performed over the default Multics switches "user_input" and "user_output". Rather, they are performed over a pair of switches named "graphic_input" and "graphic_output". These switches are not attached by the normal Multics process initialization, but must be attached by the user.

Graphic I/O is routed through a special I/O module named "graphic_dim_". The I/O module is responsible for performing the proper code conversion and graphic device control for graphic operations to a graphic device. Multics cannot determine automatically what type of graphic terminal is being utilized and so the system must be given this information by the user.

Both of these functions are performed by the setup_graphics command. The most common usage of this command is:

```
setup_graphics -table gdt_name
```

This command attaches the necessary graphic I/O switches through the graphic I/O module. It also informs the I/O module of the name of the graphic device table (GDT) to use in conjunction with graphic I/O to the specific type of graphic terminal being used.

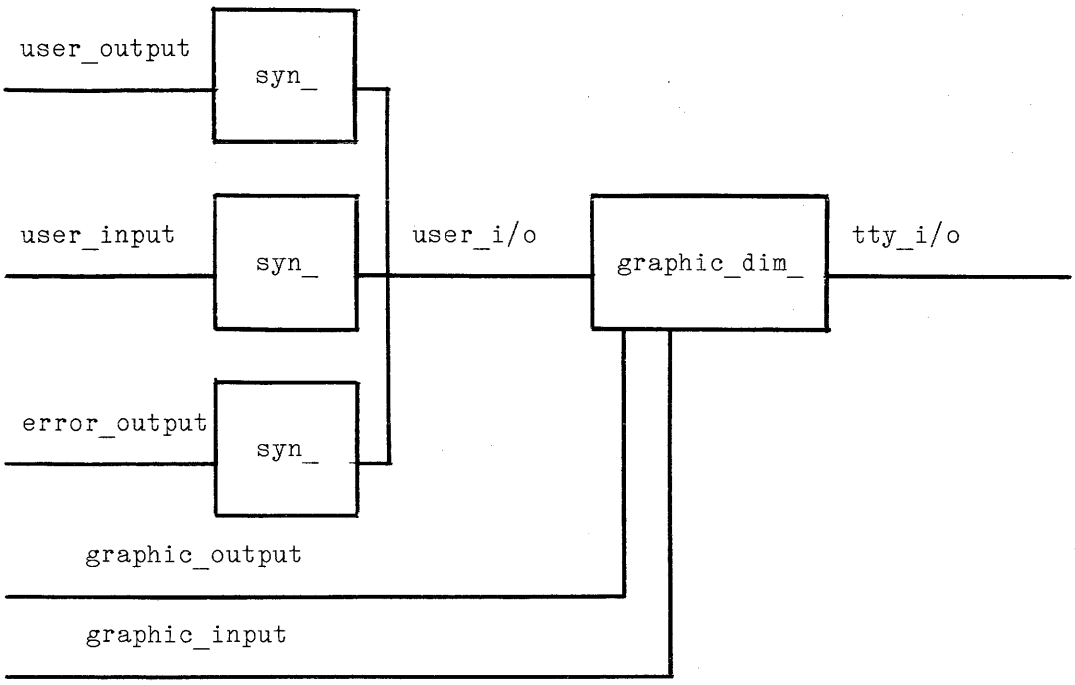
Briefly, a GDT is simply a table describing one particular type of graphic terminal. When given a certain GDT, the graphic I/O module performs terminal control and code conversion in the native language of that specific device. The use of these tables frees the user from dependencies on any particular features of one graphic device. Several GDTs are provided by the system and are listed in Section 6. The user may also provide GDTs for terminal types for which the system does not provide GDTs.

Effects on the Process' Other I/O Attachments

Users who do not take explicit action to perform unusual Multics I/O are generally safe from side effects caused by the interaction between nongraphic and graphic I/O. However, a brief explanation of the unusual nature of MGS I/O is presented below for those users who would benefit from it.

When graphics is being used in the online mode to a terminal (the normal case; as opposed to the offline mode, e.g., creating a plotter tape) all the I/O of a process, graphic and nongraphic, is routed through the graphic I/O module. This keeps unexpected nongraphic I/O such as error diagnostics or inter-user messages from being sent to the terminal without the knowledge of the graphic I/O module. Reception of such a message by the terminal when it is in a state where it expects all output to be graphic commands (i.e., in "graphic mode") would cause the message to be lost, and cause confusion to the terminal. Therefore the graphic I/O module intercepts all the output of a process and switches the terminal between graphic mode and text mode as appropriate.

In order for correct interleaving of graphic and nongraphic I/O to work, the I/O switches of the process are restructured in a fashion similar to that shown in Figure 2-2.



Sample output from an invocation of the print_attach_table command.

```
user_i/o            graphic_dim_ tty_i/o
                                       stream_input_output
user_input          syn_ user_i/o
user_output         syn_ user_i/o
error_output        syn_ user_i/o
graphic_output      graphic_dim_ tty_i/o graphic
                                       stream_output
graphic_input       graphic_dim_ tty_i/o graphic
                                       stream_input
tty_i/o             tty_  tty715       stream_input_output
```

Figure 2-2. I/O Attachments While Using Graphics

The user should never perform explicit I/O over the direct terminal switch (here named tty_i/o.)

Routing Multics Standard Graphics Code to a File

The Multics Standard Graphics Code (MSGC) produced by a graphic program may be routed to a file instead of to a terminal. This file then contains the device-independent graphic code representation of that picture. Later, when the graphic switches are again routed to the graphic terminal, the contents of this file may be written to the graphic_output switch to produce the same picture as many times as desired, without the necessity of rerunning the program that generated the picture.

To route the graphic_output switch to a file named "pic_1.graphics" the following commands would be used[1]:

```
io_call attach graphic_output vfile_ pic_1.graphics

io_call open graphic_output stream_output
```

The program is then run to produce the graphic output. When program execution is completed, the user can close and detach the file with the commands:

```
io_call close graphic_output

io_call detach graphic_output
```

The file "pic_1.graphics" contains the graphic code for the desired picture. When the graphic I/O streams are routed to a graphic terminal, the following command could be used to display the contents of the file:

```
io_call put_chars graphic_output -segment pic_1.graphics -nnl
```

USING THE CENTRAL GRAPHICS SYSTEM

The central graphics system is the lowest level and most powerful set of graphics system subroutines for creating, manipulating, editing, and displaying graphical objects. These routines form a basis for the higher level subroutines and for the utility and application packages.

Users looking for convenience and speed of programming for simple picture generation (as opposed to fine control over their graphic representation, speed of execution, and the ability to perform dynamic graphics) may find it advantageous to go directly to the discussion "Using the Higher Level Graphics Subroutines," later in this section.

Using the Graphic Manipulator

Graphic objects are created and manipulated in a temporary segment known as the working graphic segment (WGS) by the graphic_manipulator_ subroutine. The WGS must be initialized by a call to graphic_manipulator_$init before any graphic operations are attempted. This call destroys the contents of the WGS, so it is the user's responsibility to decide when he is done with the previous contents and wishes to delete them and build something new.

NODE VALUES

Each graphic item created is identified by a node value. A node value is a receipt which is returned to the user each time an item is created. Further references to that item in subsequent calls to the central graphics system are performed by supplying this node value in the call. Node values are PL/I fixed binary precision (18) quantities (FORTRAN integers). Although node values are represented as numbers they are not meant to be added, subtracted, or otherwise operated upon arithmetically. The zero node value is a special value that represents the "null element" (graphic no-op).

---

[1] By graphic system convention, files containing MSGC are named with the suffix "graphics". See the io_call command in Section 3 of the MPM Commands.

For example, a vector of dimensions (100,200,0) would be created via the call[1]:

```
declare node fixed bin (18);

node = graphic_manipulator_$create_position (Vector, 100, 200, 0, code);
```

The returned variable "node" holds the node value representing that particular vector. This node value may be used in subsequent calls to the central graphics system. For example, if a user wished to learn what graphic item was represented by this node, the following call could be issued:

```
call graphic_manipulator_$examine_type (node, non_terminal, type, code);
```

The "type" returned would be 2 (Vector). Since a vector is a terminal positional element, the dimensions of the vector could be ascertained by issuing:

```
call graphic_manipulator_$examine_position (node, type, x_dim, y_dim,
     z_dim, code);
```

The x_dim, y_dim, and z_dim variables would contain 100, 200, and 0, respectively.

BUILDING COMPOUND ELEMENTS

Since there are only fifteen atomic graphic elements (e.g., setposition, setpoint, vector, etc.), none of which are very complex, it is not surprising that most node values find their way into higher level graphic structures. These structures may be viewed as arrays of other elements. They serve the purpose of:

- associating their elements into one united entity that may itself be referenced by a single node value and treated like a single graphic element, and
- arranging their elements with respect to the order in which they are to be drawn at display time.

Suppose that a user has created four vectors of dimensions (25,0,0), (0,25,0), (-25,0,0), and (0,-25,0), and wishes to construct from these a "box" of dimension 25x25. First, the node values must be arranged in order in a PL/I array of sufficient size. (The following example uses an array of arbitrary dimension 100 which is of more than sufficient size.) This array would have a declaration similar to:

```
declare box_array (100) fixed binary (18);
```

---

[1] In this example and all other examples in this section, the variable names from the PL/I include-file "graphic_etypes.incl.pl1" have been used wherever possible, for clarity. These variables may be distinguished by the fact that they begin with capital letters (e.g., the mnemonic variable "Vector" is used instead of the less meaningful constant "2" which it represents). Users are encouraged to adopt the same policy in their programming by using the PL/I "%include" facility, as outlined in Section 8.

Rather than assigning separate node variables to this array one by one, the array elements themselves are usually supplied as the left-hand side of a function reference to graphic_manipulator_$create_position or a similar entry, for example:

```
box_array (1) = graphic_manipulator_$create_position (Vector, 25, 0, 0,
     code);

box_array (2) = graphic_manipulator_$create_position (Vector, 0, 25, 0,
     code);
```

and so on for the remainder of the box.

This PL/I array of node values may then be turned into a graphic array by the statement:

```
box_node = graphic_manipulator_$create_array (box_array, 4, code);
```

Now the variable "box_node" contains a node value that represents a graphic item made up of four vectors. This node value may be treated as an atomic element which represents a box, and may in turn be utilized in even higher level graphic structures by using the same method (see Figure 2-3).

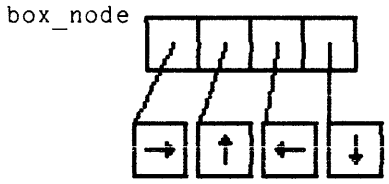


Figure 2-3. Simple Graphic Structure

The graphic_macros_ subroutine, which creates primitive elements representing boxes, polygons, and curves, simply creates a number of primitive vectors and other elements that are used to make up the desired figure with the desired dimensions. It then makes an array of these elements and returns the node value representing that array, which is used by the calling program in the same manner as any atomic graphic element.

Programming Hint

When assigning node values to PL/I arrays by supplying them as the left-hand side of a function reference to graphic_manipulator_ entries, it is useful to code the references in the following form:

```
box_array (next_free ()) = graphic_manipulator_$...
```

and include a small internal subroutine of the form:

```
next_free:
     procedure returns (fixed bin);
          free_index = free_index + 1;
          if free_index > hbound (box_array, 1) then {some error action}
          return (free_index);
     end next_free;
```

If the routine is coded in this manner, it is easy to insert a call to create a graphic element that was mistakenly omitted or delete one that was mistakenly included, without having to renumber each subsequent array index. Also, the call to graphic_manipulator_$create_array may then be coded as:

```
box_node = graphic_manipulator_$create_array
     (box_array, free_index, code);
```

GRAPHIC SYMBOLS

Graphic symbols provide a means of associating a name with a graphic object. Aside from the normal mnemonic advantages of this function, there are two other benefits:

- graphic items may refer to other graphic items by symbol names instead of by node values;
- they also provide a name by which graphic items may be stored into and retrieved from permanent storage[1].

The ability to reference graphic items by symbol name instead of by node value (number) increases the flexibility of the graphic structure. For example, suppose an architect has a graphic structure that represents a trial design of a colonnade. In various places in the structure, another graphic structure named "column" is referred to by name. To explore the effect of using differently styled columns in the overall design, the architect can simply retrieve prefabricated graphic structures representing variously styled columns from permanent graphic storage segments, rename each of them "column" in turn, and display the master structure.

The entry graphic_manipulator_$assign_name is used to create symbols. Assuming that the variable "box_node" still contains the node value of the simple box figure that was placed there in a previous example, the following statement would be used to assign it to a symbol named "box_example":

```
symbol_node = graphic_manipulator_$assign_name ("box_example", box_node,
     code);
```

---

[1] This facility is explained later in this section.

The node value returned in "symbol_node" is the node value of the graphic symbol "box_example". This value is not equal to the value of "box_node". Even though "symbol_node" is a node value, it represents a reference by name to the graphic symbol "box_example".

This is more easily understood if the operation of creating a graphic symbol is thought of not as assigning a name to a structure, but as creating a separate name-structure that references a structure. A symbol is not a name tacked onto a structure that gives rise to some arbitrary distinction in node values, but is a separate structure in its own right, that references another structure (its contents). The node value of this new name-structure is the node value of the symbol (see Figure 2-4).

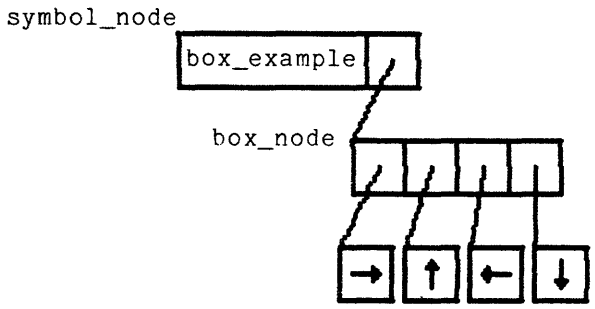


Figure 2-4. Graphic Symbol Structure

It is important to understand the distinction between the node value of a symbol (e.g., "symbol_node") and the node value of the contents of that symbol (e.g., "box_node"). The effects of using either as an element of an array is quite different.

For example, if "box_node" were to be used as an element of a graphic array, that element would remain the box figure indefinitely; or until the user specifically replaced that particular element of that particular array, using a graphic editing primitive such as graphic_manipulator_$replace_element. In the latter case, only that particular reference to the box figure would be altered. Any other references in the same array (or in other arrays that were created in the same manner) would remain unchanged. The fact that there exists a symbol named "box_example" that refers to that node would, in this case, be irrelevant.

On the other hand, if "symbol_node" were to be used as an element of all the graphic arrays, that element would represent whatever graphic structure happened to be named "box_example" at any given time. Simply by naming various graphic items "box_example", one could change the contents of all the graphic arrays to include the new figure automatically.

Historically, primitive graphic systems have required the user to make a subroutine call to control each movement of the pen, or to create one type of canned figure. This design most often resulted in the user having to write applications packages containing many subroutines, each of which was responsible for knowing how to draw one particular object. Each subroutine was called whenever the design that it knew how to draw was desired. These subroutines were actually nothing but unwieldy executable picture descriptions (graphic items).

The great advantage of working with structured picture descriptions to create an entire picture before any actual display operation takes place is that such graphic items may be created once and then referenced in numerous places through simple assignment operations.

As an example, let us assume we have a graphic object that is a stick-figure representation of a boy scout, whose size is roughly 30x100, and whose node value is contained in the variable "boy_scout". To create a row of four boy scouts, only the following program fragment would be needed[1]:

```
side_shift = graphic_manipulator_$create_position (Shift, 40, 0, 0, code);

do i = 1 to 8 by 2;
     node_array (i)   =  boy_scout;
     node_array (i+1) = side_shift;
end;

row_node = graphic_manipulator_$create_array (node_array, 8, code);
```

The node value in "row_node" represents a graphic object containing four references to the boy scout and four references to a shift of (40, 0, 0), alternately (see Figure 2-5).

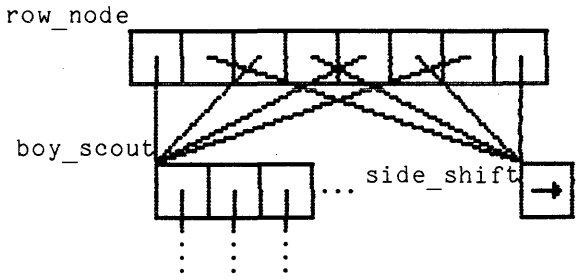


Figure 2-5. Shared Graphic Structure

1 All checks for "code = 0" in examples have been neglected for brevity. In an actual program, these checks should always be made.

This object can also be shared. To create three rows of boy scouts in parade formation, we could continue with:

```
back_shift = graphic_manipulator_$create_positon (Shift,
     -(4 * 40), 0, -60, code);
          /* go back by 60 and left by displacement of four boy scouts */

do i = 1 to 6 by 2;
     node_array (i)   =  row_node;
     node_array (i+1) = back_shift;
end;

troop_node = graphic_manipulator_$create_array (node_array, 6, code);
```

The variable "troop_node" now contains the node value of the desired structure. An example of the display that would be produced by displaying "troop_node" is shown in Figure 2-6 (the figure has been rotated for effect).

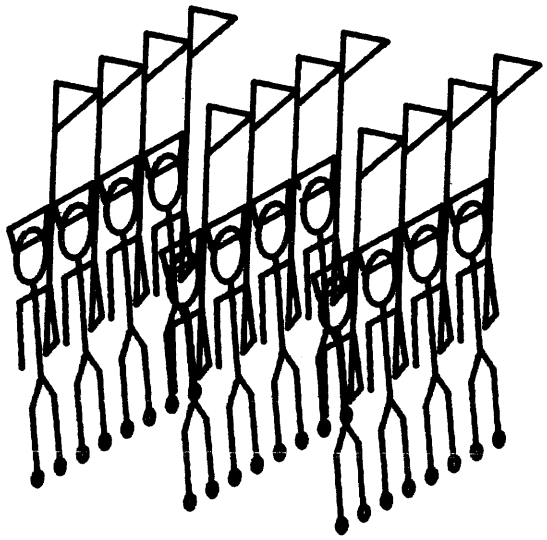


Figure 2-6. Resultant Display of Multiply-Shared Substructures

Notice that in the previous program fragment, 40 is used as the "displacement of one boy scout"; not 30, which was given as the width of one boy scout; nor 30 + 40, if you add the shift. This difference stems from the distinction between the dimensions of an object and the net relative displacement of an object. The dimensions of the boy scout are simply its width, height, and depth. The assumption here is that the net relative displacement of the boy scout (defined as the difference between the current graphic position at the point where the boy scout is drawn, and where the current graphic position is left after drawing the boy scout) is zero. That is, that the person who previously constructed the boy scout was careful to end up at the same point at which the figure began. If we had drawn four boy scouts without shifts between them, all four would have been superimposed exactly on each other. The net relative displacement between the boy scouts in the row is due entirely to the intervening shift, therefore the number 40 was used; 30 for the width of one boy scout; and a space of 10 before the next boy scout.

Since it is very useful to be able to make this assumption, it is always considered good practice to design shared graphic substructures to have a net relative displacement of zero. If this convention is ignored, it becomes very difficult to use some of the more powerful editing features of the Multics Graphics System. For example, if all the columns did not have the same net relative displacement, the architect's colonnade mentioned previously would not only be distorted, but would be distorted in different ways depending on the column in use at any moment.

For more obvious reasons, graphic structures that are meant to be shared should not contain any absolute positional elements. In fact, any graphic structure can be given a great deal of flexibility by following this rule, at the price of a small increase in complexity. For example, the elimination of absolute positioning elements in a full-screen display allows a user to later scale four such full-screen pictures to one-quarter size and display one in each corner of the screen for a composite picture. A graphic object without any absolute positional elements may be properly positioned by taking into account the convention defining the current graphic position at the beginning of every display operation as (0,0,0).

USING MODES AND MAPPINGS

Modes provide a way to alter the appearance of a graphic item without altering the item itself. By applying modes to an object, it can be:

- caused to blink
- made dimmer or brighter
- made dotted or dashed rather than solid
- made sensitive or insensitive to a light pen
- changed in color

Mappings provide a means of altering the orientation, dimensions, or extent of a graphic item without altering the item itself. By applying mappings to an object, it can be:

- scaled in size independently in three dimensions
- rotated to any orientation relative to the screen
- clipped so that parts of the item outside a given area disappear
- masked so that parts of the item inside a given area disappear

The application of modes and mappings follow a set of rules which state how and when they combine with each other, how and when they supersede each other, and when their effect is to be removed.

First, we must define the local graphic environment with respect to modes and mappings. The local graphic environment at any level in a graphic structure is simply all the modes and mappings that are in effect at that level. The graphic environment at the top level of any graphic structure is always:

- blinking off
- full intensity
- solid vectors
- insensitive to a light pen
- color white[1]
- unity scaling
- zero rotation
- no clipping
- no masking

Any modes or mappings encountered from this point on change the graphic environment.

Each level of the structure carries with it its own local graphic environment. When a level is entered, it inherits the graphic environment of its parent structure (the list or array that referenced it.) When that level returns to its parent, its own local graphic environment is discarded. This means that no matter what modes or mappings occur in any substructure, the parent structure can never be affected. For example, a user can feel free to make use of various canned graphic structures in a permanent library no matter what modes or mappings they may use to achieve their operation. Nothing any substructure does could possibly cause the rest of that user's structure to suddenly become invisible, blinking, or distorted. (Note that this feature applies to modes and mappings only. For instance, a substructure still has the ability to modify the user's current graphic position in some unforeseen way.)

When two modes or mappings of the same type occur at the same level in a piece of graphic structure, the most recent overrides the other. This makes it possible for a user to create an array in which the first few items blink and the rest do not, by assembling the array from nodes representing the graphic items:

```
blink on
(vectors which are to blink)
blink off
(vectors which are not to blink)
```

Items of "the same type" are two color items, two scaling items, and so on. For instance, a blinking item and a color item have no overriding effects on each other.

When two modes or mappings which are not at the same level of the graphic structure interact, the rules describing what occurs are more complex.

---

[1] Equal intensities of all primary colors of light.

If two modes interact on different levels of a graphic structure, the more recent overrides that of the parent. As stated above, when a level returns to its parent, the parent's modes again take precedence. For example, if a certain substructure explicitly contains a red color effector, no manipulation of color items in the parent can change the color of the red part of the substructure. However, a different type of mode can sometimes be used to have a predictable effect on the item in question. For example, although a user could not change the color of this substructure from the parent level, the substructure can be "turned off" with an "intensity zero" item (provided that it did not also force its own intensity.)

If two rotations or scalings interact on different levels of a graphic structure, they combine. For example, if a substructure that scales parts of itself by a factor of two is incorporated into a larger structure that scales itself by a factor of three, the originally scaled items in the substructure have an effective scaling factor of six. Although the mathematics of rotation cannot be as easily explained, the concept is the same. A substructure that rotates parts of itself to achieve an effect may be incorporated into a larger structure that is also rotated. The effect of rotating the parent structure is just as it would be if the substructure had been initially created to look as it does without the aid of rotations. Put another way, regardless of what rotations are used in a substructure, when a parent structure is rotated the substructure follows it around in a natural manner. Again, the substructure's contribution to the combined rotation is discarded upon return to the parent structure.

If two clippings interact on different levels of a graphic structure, they intersect. Only those items which reside within the intersection of both clipping areas appear. If two maskings interact on different levels of a graphic structure, they unite. Only those items which reside outside the union of both masking areas appear[1].

Mode and mapping items, like positional items, are created by calls to graphic_manipulator_. They, too, are represented by node values.

To create a picture containing a blinking box on the left side of the screen and a steady box on the right, we could use the following program fragment. (Assume that the variable "box_node" still contains our box.)

```
node_array (1) = graphic_manipulator_$create_position
                   (Shift, -80, 0, 0, code);
node_array (2) = graphic_manipulator_$create_mode
                   (Blink, Blinking, code);
node_array (3) = box_node;

node_array (4) = graphic_manipulator_$create_mode
                   (Blink, Steady, code);
node_array (5) = graphic_manipulator_$create_position
                   (Shift, 160, 0, 0, code);
node_array (6) = box_node;

picture_node   = graphic_manipulator_$create_array (node_array, 6, code);
```

[1] Extent elements (clipping and masking) although planned, are not currently implemented in the MGS.

Notice that the user has to remember to revoke the blinking item before referencing the second box; otherwise, it too would blink. The structure created would look like Figure 2-7.

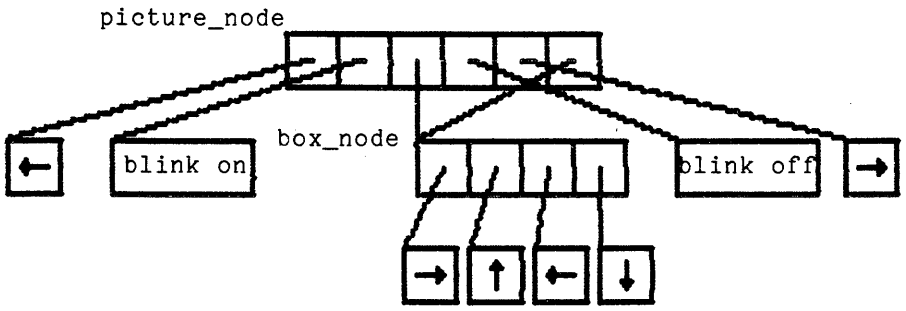


Figure 2-7. Explicit Reversion of Mode Elements

Another approach could have been taken, that of letting the local graphic environment do the reversions. The following program fragment illustrates this approach:

```
node_array (1) = graphic_manipulator_$create_mode
                   (Blink, Blinking, code);
node_array (2) = box_node;

blinking_box = graphic_manipulator_$create_array (node_array, 2, code);

node_array (1) = graphic_manipulator_$create_position
                   (Shift, -80, 0, 0, code);
node_array (2) = blinking_box;

node_array (3) = graphic_manipulator_$create_position
                   (Shift, 160, 0, 0, code);
node_array (4) = box_node;

picture_node   = graphic_manipulator_$create_array (node_array, 4, code);
```

This is slightly more difficult to understand, but it illustrates a very important property of modes. The structure just created is diagrammed in Figure 2-8.

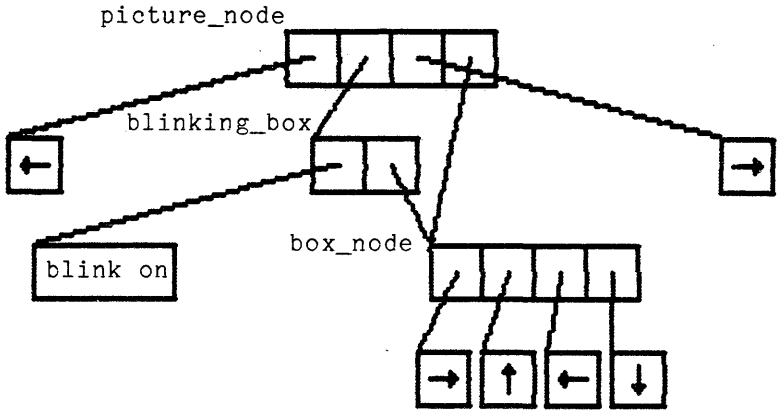


Figure 2-8. Automatic Reversion of Modes Using Structuring

Note that by making the blinking box a substructure, we contained the effect of the blink item only to that substructure. When the graphic display mechanism returns to the topmost structure, the blink item is automatically reverted. This structure results in a display that is exactly like the previous one.

USING DATABLOCKS

Datablocks are blocks of storage space associated with graphic structures. They can be used to hold nongraphic information that may be relevant to user programs which manipulate this structure.

Datablocks are useful to some graphic applications. However, their use is unnecessary to most.

For an example of an application where datablocks are useful, consider the designer of a space vehicle. As the graphic description of the creation is at least as detailed as any of the testing and simulation programs, he has decided to use it as the master data structure. Therefore, along with the description of each piece, he will probably find it necessary to store its weight, composition, and so on. He uses datablocks associated with each object to hold this information. The program which pieces the parts together into higher level objects will also combine the weights and attach this information to the new object in a datablock. The testing and simulation programs can then go as deep into the structure as they have to, testing individual pieces; or they can perform quick analyses using only the major component blocks of the vehicle. As it becomes necessary to alter the configurations of pieces, he can at the same time alter the other properties.

Datablocks may be created using the entry graphic_manipulator_$create_data. They are stored simply as bit strings. It is left to the user to design private conventions so that the kind of data contained in the datablock can be determined. either by structural position with respect to some known type of object, or by including a descriptor along with the data.

To create a datablock with the contents of a PL/I variable named "my_data", the following call could be used[1]:

```
data_node = graphic_manipulator_$create_data
            (length (unspec (my_data)), unspec (my_data), code);
```

The variable "data_node" holds the node value of the created datablock. It may be used in higher level graphic structures like any other node value. Note that "my_data" may be any PL/I data type, including scalar, array, or structure.

The following program fragment retrieves the contents of "my_data":

```
begin;

declare bit_string bit (length (unspec (my_data)));

call graphic_manipulator_$examine_data (data_node, bit_string, len, code);

unspec (my_data) = bit_string;

end;
```

The contents of the datablock are now found back in the variable "my_data". The variable "bit_string" was used as an intermediary since the entry requires a bit argument. The expression "unspec (my_data)" was not used in that position, since a PL/I call-by-value would have resulted, which is not appropriate in an output argument position.

ESTABLISHING PERMANENT LIBRARIES OF GRAPHIC OBJECTS

Graphic objects, once created, may be stored in permanent graphic segments (PGS). Objects to be stored in this manner must be defined with graphic symbols. There are two ways that an object may be stored:

- the entire contents of the WGS may be stored; or
- the structures attached to certain graphic symbols may be stored.

The entry graphic_manipulator_$save_file may be used to store the entire WGS. For example, the following call stores the contents of the WGS into a segment named "misc.pgs" in the working directory:

```
call graphic_manipulator_$save_file (get_wdir_ (), "misc", code);
```

This operation destroys any previous contents of misc.pgs. The suffix "pgs" may be supplied as part of the PGS name, or may be omitted.

The entry graphic_manipulator_$use_file loads the contents of a PGS into the WGS (replacing all the previous contents of the WGS). The following statement is used to load misc.pgs:

```
call graphic_manipulator_$use_file (get_wdir_ (), "misc", code);
```

---

[1] "length" and "unspec" are PL/I built-in functions.

To move only single graphic objects between segments, the graphic_manipulator_$put_struc and graphic_manipulator_$get_struc entries are used. The user of these entries may choose between four modes of structure replacement:

1. The structure defined by the symbol, including all subsidiary symbols and their structures, is to be moved. If any symbol name matches a symbol already defined in the target segment, abort the move and return an error code.

2. The structure defined by the symbol, including all subsidiary symbols and their structures, is to be moved. If any symbol name matches a symbol already defined in the target segment, the contents of the symbol being moved replace the previous value of the symbol. This operation loads the symbol forcibly, destroying the contents of old symbols whose presence would have caused a name conflict.

3. The structure defined by the symbol, including all subsidiary symbols and their contents, is to be moved. If any subsidiary symbol name matches a symbol already defined in the target segment, the previous contents are used. This operation allows a user to move a superstructure between segments without redefining any of the inferior symbols which may be present in the target segment.

4. The structure defined by the symbol, including all subsidiary symbols but not their contents, is to be moved. If any subsidiary symbol name matches a symbol already defined in the target segment, the previous contents are used; otherwise the subsidiary symbol is created with no contents. This operation allows a user to move a superstructure between segments, ignoring any unwanted symbol substructuring that is not essential to the application and thus was not pre-created.

To store the "box_example" symbol, use the call:

```
call graphic_manipulator_$put_struc (get_wdir_ (), "misc", "box_example",
     On_dup_error, code);
```

The usage of graphic_manipulator_$get_struc is similar.

Node values are not maintained across these operations. Programs that store information about the correspondence between certain node values and certain graphic items, and then attempt to reload a graphic structure and use this same information, will not operate properly. If node values must be used in such programs, they may be recomputed using the structure examination entries in graphic_manipulator_.

Using the Graphic Compiler

The graphic_compiler_ subroutine examines a graphic structure in the WGS, translates it into device-independent MSGC, and dispatches this code to the proper I/O switch (usually graphic_output). There are four basic entries, one of which is usually sufficient to perform the desired graphic compilation and display.

Entry graphic_compiler_$display accepts a node value as input. The following call erases the screen and displays the troop of boy scouts previously described:

```
call graphic_compiler_$display (troop_node, code);
```

Entry graphic_compiler_$display_name accepts a symbol name as input. The following call erases the screen and displays the graphic symbol "box_example":

```
call graphic_compiler_$display_name ("box_example", code);
```

The entry graphic_compiler_$display_append is similar to the entry graphic_compiler_$display. This entry appends the display to the screen without erasing the current display.

In a similar fashion, the entry graphic_compiler_$display_name_append may be used to complement the entry graphic_compiler_$display_name.

Using the Graphic Operator

Thegraphic_operator_ subroutine may be used to perform graphic actions that are not representable as elements of a graphic object, such as graphic input, animation and screen erasure. A description of the syntax of the calling sequence of each entry of the graphic_operator_ subroutine may be found in Section 5.

Graphic operations can be caused to come to a temporary halt by the use of the graphic_operator_$pause entry. Graphic operations are suspended until the user signifies, by some interaction, readiness to proceed. This may be used, for example, when displaying multiple graphs in sequence, to allow the user to fully understand the information in each graph before the screen is erased and the next graph is drawn.

The screen may be explicitly erased through the use of the graphic_operator_$erase entry.

Subroutine graphic_operator_ is also used to request input from graphic input devices. Three types of graphic input are defined in the MGS:

1. "where" input, consisting of one coordinate point.

2. "which" input, consisting of a node value and an index "pathname" that uniquely identifies an instance of a possibly shared graphic substructure or item being displayed.

3. "what" input, consisting of any graphic item or structure that is constructed at the terminal and returned to Multics.

These types of input are requested by the graphic_operator_$where, graphic_operator_$which, and graphic_operator_$what entries, respectively.

"where" input is useful when the programmer wishes the terminal user simply to pick a point. For example, a user with a requirement to label curves on a plot cannot easily write a routine to determine where each label should be placed so that it is legible (does not interfere with other curves or other labels on the graph.) However, if a "where" input is requested for each label after the graph has been drawn, the user can manually choose a satisfactory position for each label. The graphing program need only obtain the coordinate point input, and construct a graphic structure that performs a setposition to that point and displays the proper label.

"which" input is useful when the programmer wishes the terminal user to choose some graphic item being displayed. For example, a computer-aided automotive design program might display an entire automobile, and request a "which" input to allow the terminal user to choose a certain component or assembly for subsequent alteration. The terminal user could select the left front wheel with the light pen. Then via local interactions with the intelligent graphics terminal, the level of the item desired would be selected (e.g., that wheel, the entire front suspension, or only the hubcap which was actually touched). When satisfied with the choice, the user causes the terminal to return this information. The Multics-resident automobile design program might then select that substructure, display it alone, and perform further operations.

"what" input is useful when the programmer wishes to accept an arbitrary graphic object or structure from the terminal user. Usually, the type of input obtained depends heavily on the graphic input device from which the input is requested. For instance, a program may allow a user to draw an arbitrary figure using a graphic mouse, joystick, or pen and tablet. This input would be translated into an array of vectors, points, shifts, etc., inserted in the WGS, and the node value returned to the program. The program may inspect it, save it in a PGS, or incorporate it into a larger structure.

Since the operations performed by graphic_operator_ are rather sophisticated, users should refer to Section 5 for detailed information.

Examples

The following two programs serve to illustrate a typical use of the central graphics system. The first program creates a cube, assigns it the name "cube", and stores it in a permanent graphic segment named "misc.pgs". In examples that illustrate the user's interaction with the terminal, the lines typed by the user are indicated with an exclamation mark (!) to the left of the line. This is for illustrative purposes only; the user does not actually type the exclamation mark. Likewise, comments that serve an explanatory purpose may be included within the program by enclosing them within "/* ... */".

```
make_cube: proc;

declare  array (100) fixed bin (18), /* for ordering elements */
         array_index fixed bin; /* index of last-used element in array */

declare (cube_array, cube_symbol) fixed bin (18);
         /* graphic node holders */

declare  code fixed bin (35),
         com_err_ entry options (variable),
         get_wdir_ entry returns (char (168));

%include gm_entry_dcls; /* contains dcls for graphic_manipulator_ */
%include graphic_etypes;
     /* ditto for common graphic variables (Vector, etc.) */

     /* initialize the working graphic segment */
               call graphic_manipulator_$init (code);
               if code ^= 0 then goto err;

               array_index = 0;

     /* move to edge of cube */
               array (next_free ()) = graphic_manipulator_$create_position
                    (Shift, 250, 250, 250, code);
                     if code ^= 0 then goto err;
```

```
/* create faces of cube, one by one,
   out of relative vectors and shifts */

          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, -500, 0, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, 0, -500, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, 500, 0, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Shift, -500, 0, 500, code); if code ^= 0 then goto err;

          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 500, 0, 0, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, 0, -500, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, -500, 0, 0, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Shift, 0, -500, 500, code); if code ^= 0 then goto err;

          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, 500, 0, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, 0, -500, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, -500, 0, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Shift, 500, 0, 500, code); if code ^= 0 then goto err;

          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, -500, 0, 0, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 0, 0, -500, code); if code ^= 0 then goto err;
          array (next_free ()) = graphic_manipulator_$create_position
               (Vector, 500, 0, 0, code); if code ^= 0 then goto err;

/* join all the discrete elements together into a graphic array */
          cube_array = graphic_manipulator_$create_array
                       (array, array_index, code);
          if code ^= 0 then goto err;

/* name that array "cube" by making a graphic symbol containing it */
          cube_symbol = graphic_manipulator_$assign_name
                        ("cube", cube_array, code);
          if code ^= 0 then goto err;

/* save the symbol "cube" in "misc.pgs" in my working dir */
/* if the symbol already exists there, return an error and leave it. */
          call graphic_manipulator_$put_struc (get_wdir_ (),
               "misc", "cube", On_dup_error, code);

          if code ^= 0 then
err:           do; /* complain if any code nonzero */
               call com_err_ (code, "make_cube", "Can't continue.");
               return;
          end;

          return;

/* ------------------ */

next_free: procedure returns (fixed bin);

/* simply bump array index by one and return it as next free index. */
               array_index = array_index + 1;
               return (array_index);
          end next_free;
```

```
end make_cube;
```

The second program takes the cube found in misc.pgs and displays it, subject to a rotation which the user specifies.

```
show_cube: proc;

declare (cube_symbol, cube_array, junk_node, /* graphic node variables */
        rotated_cube_node, rotation_node) fixed bin (18),
        array (2) fixed bin (18);

declare  angles (3) float bin,
        (sysin, sysprint environment (interactive)) stream,
        conversion condition;

declare  com_err_ entry options (variable),
        code fixed bin (35);

declare  get_wdir_ entry returns (char (168));

%include gm_entry_dcls; /* contains dcls for graphic_manipulator_ */
%include gc_entry_dcls; /* contains dcls for graphic_compiler_ */

/* get the contents of "misc.pgs" in the WGS */
          call graphic_manipulator_$use_file (get_wdir_ (),
               "misc", code); if code ^= 0 then goto err;

/* get the node value of the graphic symbol "cube" */
          cube_symbol = graphic_manipulator_$find_structure ("cube",
               cube_array, code); if code ^= 0 then goto err;

/* construct a little structure for us to use.  The first element
   holds a rotation (later on);  the second holds the cube */
          array (1) = 0; /* null node, a graphic no-op */
          array (2) = cube_symbol;

          rotated_cube_node = graphic_manipulator_$create_array
                              (array, 2, code);
          if code ^= 0 then goto err;

/* set up the program exit condition */
          on conversion begin;
               put list ("Conversion occurred;  quitting.");
               goto quit;
          end;

/* main program loop */
          do while ("1"b);

               put list ("Enter x, y, z angles: ");
               get list (angles);

/* create a rotation element from those angles */
               rotation_node = graphic_manipulator_$create_rotation
                    (angles (1), angles (2), angles (3), code);
               if code ^= 0 then goto err;

/* replace the old rotation with the new rotation */
               junk_node = graphic_manipulator_$replace_element
                           (rotated_cube_node, 1, rotation_node, code);
               if code ^= 0 then goto err;

/* display the rotated cube */
               call graphic_compiler_$display (rotated_cube_node, code);
```

```
                    if code ^= 0 then goto err;
               end;

err:           call com_err_ (code, "show_cube", "Can't continue.");
quit:          return;

     end show_cube;
```

The following script represents a sample terminal session where these two programs were used.

First, create and save a cube.

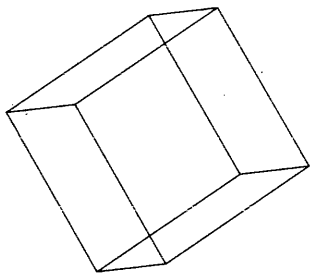
```
! make_cube
  r 1226 1.082 3.928 80
```

See what happens if we try to do it more than once.

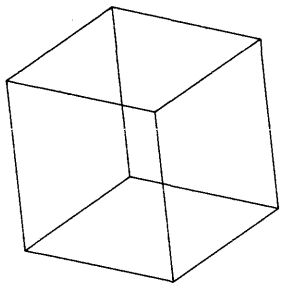
```
! make_cube
  make_cube: A name duplication has occurred in
                    moving a graphic structure.  Can't continue.
  r 1226 0.062 0.094 2

! show_cube
  Enter x, y, z angles:
! 10, 20, 30
  show_cube: No I/O switch.  Can't continue.
  r 1226 0.486 1.850 25
```

The user neglected to set up graphic I/O.

```
! setup_graphics -table tek_4014
  r 1226 0.400 2.380 20

! show_cube
  Enter x, y, z angles:
! 10, 20, 30
```



```
   Enter x, y, z angles:
!  60, 34, -10
```



Exit the program.

```
   Enter x, y, z angles:
!  xxx
   Conversion occurred;  quitting.
   r 1226 0.657 2.612 27

!  list

   Segments = 5, Lengths = 6.

   r w     1  misc.pgs
   re      1  show_cube
   r w     1  show_cube.pl1
   re      2  make_cube
   r w     1  make_cube.pl1

   r 1226 0.131 0.420 11

!  list_pgs_contents misc

                    >udd>Proj>User>g>misc.pgs

   1 symbol:

             cube

   r 1230 0.271 8.372 90
```

USING HIGHER LEVEL GRAPHIC SUBROUTINES

In addition to the central graphics system, higher level graphic subroutines are provided for special applications. As with all graphic routines, users of these subroutines must remember to set up their graphic I/O environment before attempting to perform graphic operations.

These subroutines are primarily supplied to provide:

- speed in programming small applications, and
- a simple interface to the graphics system.

In addition, each provides its own specific features, such as support for transportability, or a simple graphing application.

None of the higher level graphics subroutines contain provisions to allow the user to directly:

- perform graphic input,
- perform animation and dynamic graphics,
- control the sharing of graphic objects[1],
- edit the created graphic object,
- use graphic mappings,
- use all of the graphic modes, or
- save the created display lists[2].

Using gui_

The gui_ subroutine allows a user to create graphic objects with a minimum of effort. These objects are assembled for the user in a linear fashion as they are created. The user may display the contents of the display list at will, append more elements to the display list, and display the new additions.

Most entries in gui_ create one element or one simple figure, and append it to the display list. The other entries control miscellaneous functions such as screen erasure and display list reinitialization.

An example of a program using gui_ may be found in Section 5, "Example of gui_".

---

[1] Failure to do this can sometimes result in premature exhaustion of free space in the WGS, as well as greatly increasing the execution time of the graphic_compiler_.

[2] However, this feature can be performed by user programming through the use of graphic_manipulator_$save_file.

Users who desire the capabilities of structure editing, graphic input and controlled sharing, may easily approximate the operation of gui_ by using various entries in graphic_manipulator_ and graphic_macros_, in conjunction with the graphic_manipulator_$add_element entry, at the cost of a small to moderate amount of extra programming.

Using calcomp compatible subrs

The calcomp_compatible_subrs_ subroutine provides a compatible interface for those programs that previously ran under other operating systems and used the standard CalComp plotter calls. They may, of course, be used as a basis for new programs if the user prefers these calls to those supplied by the central graphics system or the gui_ subroutine.

Since the original plotter calls were highly sensitive to the width (in inches) of the plotter in use, programs which have been transferred from other systems must precede their use of this subroutine with a call to calcomp_compatible_subrs_$set_dimension, to inform the graphics system of the units to be used to dimension graphic items. Additionally, they must be modified if they attempt to produce multiple plots by rolling the paper forward. Instead, they should perform the proper sequence of calls to close out one plot, and then perform the call to open the next plot. A detailed description of calcomp_compatible_subrs_ implementation restrictions can be found in Section 5.

Using plot_

The plot_ subroutine simply provides a quick method of using the graphic system to produce data graphs. The user issues a call to plot_$setup to specify title, axis titles and type of graph; lets plot_ pick the graph scaling (or uses plot_$scale to force a certain scaling); and then makes one or more calls to plot_, supplying the data points to be plotted.

An example of a program using plot_ may be found in Section 5, "Example of plot_".

Using the Graphic Editor

The graphic editor provides a command interface to the central graphics system. It allows the user to construct, edit, and display graphic structures via interactive requests to the editor.

Although the user of the graphic editor need not fully understand the principles of operation of the central graphics system, users who are performing more than simple creation and editing of structures may find this understanding advantageous.

Section 5 (Commands) describes the graphics_editor and contains many simple examples of its use. Users should read and attempt to understand this material before proceeding in order to gain a feel for the syntax of requests.

The following paragraphs provide several extended examples of a terminal session using the graphic editor. Explanatory notes are interspersed with the examples.

The following example outlines the creation of a "troop of boy scouts," previously described in this section.

```
! setup_graphics -table tek_4014
  r 1715 0.673 5.254 37

! graphics_editor
  Edit.

! boy_scout = vector -10 20, vector 40, vector -10 -20, vector -20,
!           shift 10 10, /* the hat */
!           circle 0 -20, shift 0 -40, /* the head */
!           vector 0 -10; /* the neck */
```

No setposition element is included since this object is just a small component of a larger planned structure.

Display to see what it looks like so far. The origin of the object will by default be the point (0,0,0).

```
! display boy_scout;
```

Assume the hat is the wrong shape. Fix it before going further. First, get a list of the elements of boy_scout to use as an aid.

```
! show boy_scout.*;
  boy_scout.1 is vector -10. 20. 0.,
  boy_scout.2 is vector 40. 0. 0.,
  boy_scout.3 is vector -10. -20. 0.,
  boy_scout.4 is vector -20. 0. 0.,
  boy_scout.5 is shift 10. 10. 0.,
  boy_scout.6 is system macro "circle 0. -20.",
  boy_scout.7 is shift 0. -40. 0.,
  boy_scout.8 is vector 0. -10. 0.;
```

The first five elements are the hat; change them.

```
! boy_scout.1:5 = vector -6 20, vector 52, vector -6 -20, vector -40,
!                 shift 20 10;

! display boy_scout;
```

Assume this now looks satisfactory. Continue by adding elements to boy_scout.

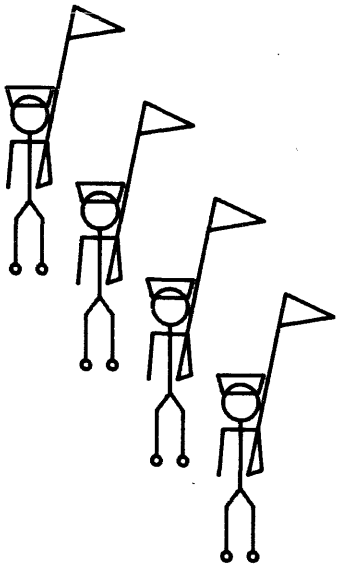
```
! boy_scout = boy_scout.*, /* all the previous elements of boy_scout, plus: */
!             vec -20 0, vec -4 -50, shift 24 50, /* right arm */
!             vec 20, vec 4 -45, vec -5 -5, /* left arm and hand */
!             vec 41 200, vec 55 -25, vec -60 -10, shift -55 -115, /* flag */
!             vec 0 -65, /* body */
!             vec 15 -20, vec 0 -50, circle 0 -5, shift -15 70, /* left leg */
!             vec -15 -20, vec 0 -50, circle 0 -5, shift 15 70; /* right leg */

! di boy_scout;
```

Assume the hand does not show enough; make it longer.

```
! show boy_scout.9:20
  boy_scout.9 is vector -20. 0. 0.,
  boy_scout.10 is vector -4. -50. 0.,
  boy_scout.11 is shift 24. 50. 0.,
  boy_scout.12 is vector 20. 0. 0.,
  boy_scout.13 is vector 4. -45. 0.,
  boy_scout.14 is vector -5. -5. 0.,
  boy_scout.15 is vector 41. 200. 0.,
  boy_scout.16 is vector 55. -25. 0.,
  boy_scout.17 is vector -60. -10. 0.,
  boy_scout.18 is shift -55. -115. 0.,
  boy_scout.19 is vector 0. -65. 0.,
  boy_scout.20 is vector 15. -20. 0.;

! boy_scout.14 = vector -15 -5;
! boy_scout.18 = shift -45 -115;
  /* to make the flag come out even at the shoulder */

! di boy_scout
```

Assume this is now satisfactory. Now make a row of boy scouts.

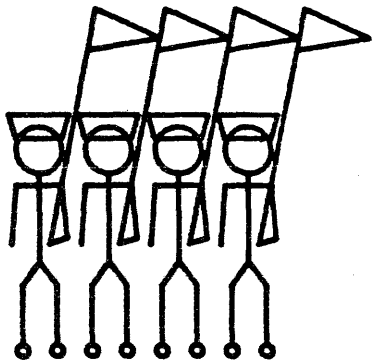
```
! row = boy_scout, shift 60, boy_scout, shift 60, boy_scout, shift 60,
!       boy_scout, shift 60;

! di row;
```



This is not right. It seems we forgot to give the boy scout a net relative displacement of zero. We must fix this.

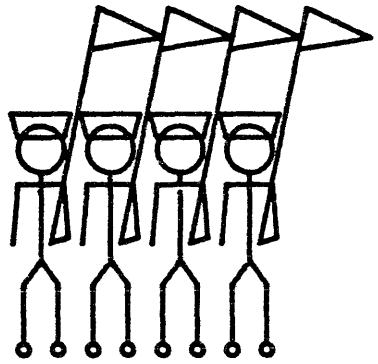
```
! show boy_scout.*;
   boy_scout.1 is vector -6. 20. 0.,
   boy_scout.2 is vector 52. 0. 0.,
   boy_scout.3 is vector -6. -20. 0.,
   boy_scout.4 is vector -40. 0. 0.,
   boy_scout.5 is shift 20. 10. 0.,
   boy_scout.6 is system macro "circle 0. -20.",
   boy_scout.7 is shift 0. -40. 0.,
   boy_scout.8 is vector 0. -10. 0.,
   boy_scout.9 is vector -20. 0. 0.,
   boy_scout.10 is vector -4. -50. 0.,
   boy_scout.11 is shift 24. 50. 0.,
   boy_scout.12 is vector 20. 0. 0.,
   boy_scout.13 is vector 4. -45. 0.,
   boy_scout.14 is vector -15. -5. 0.,
   boy_scout.15 is vector 41. 200. 0.,
   boy_scout.16 is vector 55. -25. 0.,
   boy_scout.17 is vector -60. -10. 0.,
   boy_scout.18 is shift -45. -115. 0.,
   boy_scout.19 is vector 0. -65. 0.,
   boy_scout.20 is vector 15. -20. 0.,
   boy_scout.21 is vector 0. -50. 0.,
   boy_scout.22 is system macro "circle 0. -5.",
   boy_scout.23 is shift -15. 70. 0.,
   boy_scout.24 is vector -15. -20. 0.,
   boy_scout.25 is vector 0. -50. 0.,
   boy_scout.26 is system macro "circle 0. -5.",
   boy_scout.27 is shift 15. 70. 0.;

! boy_scout.27 = shift -5 175;
```

```
! di row;
```



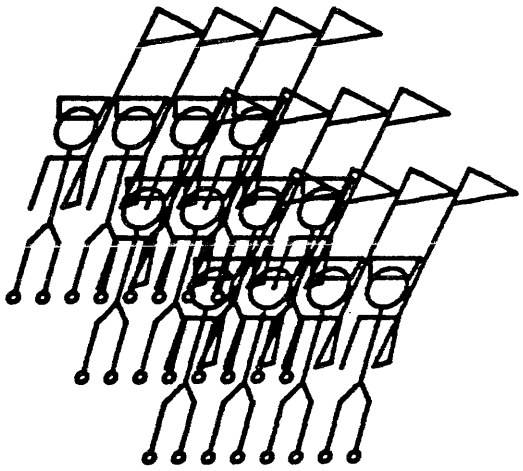
Fine! Now share the row in the same manner to make the troop in formation.

```
! troop = row, shift -240 0 -140, row, shift -240 0 -140, row;
! di troop
```



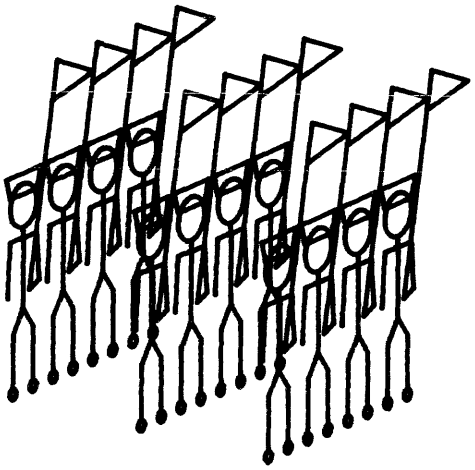
This is right, but it will have to be rotated if the z dimension is to show at all.

```
! parade = rotation 30 30, troop;  display parade;
```



This is not artistically pleasing, try a slightly different view.

```
! parade.1 = rot 30 50 25;  display parade;
```



This is satisfactory. Save the graphic structure.

```
! save illustration

! list

   4 symbols in illustration.pgs:

        boy_scout
        parade
        row
        troop

! quit
  r 1716 14.211 8.124 170
```

Using Modes or Mappings to Alter Shared Structures

This example outlines the use of a graphic superstructure to aid in the design of a three-dimensional object. A superstructure named "orth" is created, which shows a standard draftsman's three-view orthographic projection (plus a projection at a nominal angle) of the graphic structure named "item".

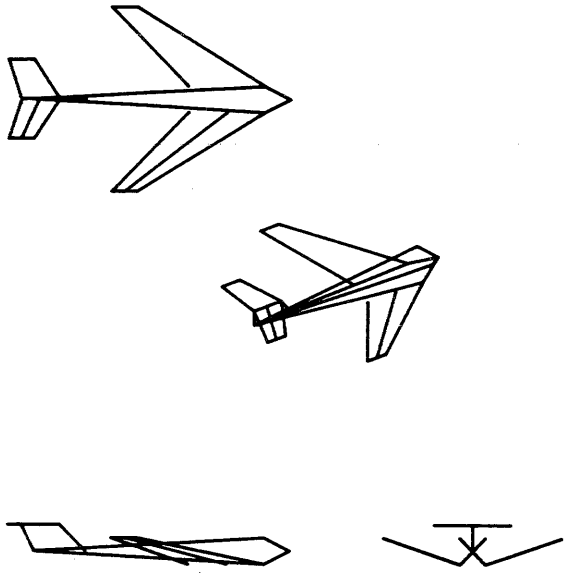
```
! graphics_editor
  Edit.

! item = null;      /* just to define it */

! orth = setposition -300 300, rot 90, item, rot 0,        /* top view */
!       setposition -300 -300, item,                       /* front view */
!       setposition 300 -300, rot 0 -90, item, rot 0,      /* side view */
!       setposition 0 0 0, rotation 60 40 20, item;        /* nominal view */

! put (aids) orth; /* put orth into aids.pgs */
! quit
```

At a later point, the user is faced with the task of assembling a three-dimensional representation of an aircraft. After creating the first attempt and naming it "aircraft", he desires to see it in an orthographic projection. He retrieves the superstructure orth, proceeding as follows:

```
! get (aids) orth
```

He assigns "aircraft" to "item", making aircraft the object to be displayed in orthographic projection.

```
! item = aircraft;

! display orth;
```



NOTE: The portion of the display commented above as "front view" actually results in a "side view" of the aircraft. This apparent discrepancy arises from the fact that the aircraft figure was originally constructed in this position, making this view the "front view" of the figure, even though it does not correspond to what is usually considered the front view of an aircraft.

Now the user may make any desired changes to the aircraft figure. After any change, he need only retype "display orth" to see the object again in all four orientations.

TERMINAL LIMITATIONS AND PECULIARITIES

Although the MGS attempts to simulate the operation of the VGT on each different type of graphic device, certain operations are not performed, either because the hardware cannot directly support the operation, or because the operation has no meaning to that particular device.

Examples of operations that are not supported because the terminal hardware cannot directly support the operation include:

1. the use of color elements on a terminal that is not equipped to display colors.
2. the use of blinking elements on a plotter.
3. dynamic animation of a graphic object on a storage-tube terminal, where pictures, once drawn, are permanent until the screen is erased.

These operations sometimes seem to be accepted, in that they may not produce error messages when used with a particular device, but nevertheless produce no effect on the display. This can happen for two reasons. First, the GDT describing the terminal may flag some operations as being futile but harmless. For example, a user sending a structure containing color elements to a monochromatic (single-color) terminal does not receive an error, even though the color element cannot be performed. In this case, such items are simply ignored. Second, the GDT can contain assumptions that the terminal class it describes includes certain features that the particular terminal in use does not. For instance, dotted and dashed lines are an extra-cost option on some terminals. If the specific terminal in use does not possess this option, linetype elements have no effect on the display. Similarly, requests for graphic input from input devices that are available and supported for a certain terminal type, but are not attached to the specific terminal in use, are accepted; and the system waits for input that cannot be sent. However, some of these operations may be mapped by software into equivalent or compromise operations. For example, the "screen erase" graphic element causes a paper advance if sent to a plotter.

Examples of operations that are not supported because the operation has no meaning to a particular graphic device include attempts to perform graphic pauses in plotter output; attempts to refer to the graphic structure in the memory of the graphic terminal when that terminal has no internal memory[1] : and attempts to perform graphic input from terminals that do not support any graphic input devices.

The user of any graphic device should clearly understand the limitations of the particular terminal being used before programming sophisticated graphic applications. In fact, to perform every operation defined by the MGS in its most general case, a user would need a very powerful system of graphics hardware, complete with sophisticated internal programming. The user should be aware that support of graphic terminals of nominal power entails compromises in terms of flexibility.

SEARCH LIST

Various entries in the graphics system use the graphics search list. For more information about search lists, see the descriptions of the search facility commands in MPM Commands (in particular, the add_search_paths command description). Type:

```
print_search_paths graphics
```

to see what the current graphics search list is. The default search list is:

```
-working_dir
>system_library_unbundled
```

[1] This ability constitutes the basis for all the graphic effectors that perform dynamic animation and real-time graphic editing, among others.

SECTION 3

STRUCTURE OF THE SYSTEM

The MGS is organized into two distinct functional parts:

- terminal-independent or central graphics system
- terminal-dependent interface

(See Figure 3-1.)

User and application programs communicate almost exclusively with the central graphics system. The central graphics system manipulates a data base containing a structured representation of a graphic picture. When a user or application program displays a portion of a graphic structure, the structure is transformed into a character-string representation known as MSGC, which is suitable for transmission through a Multics I/O switch to the terminal device. This code contains both redundant information needed by static (storage-tube) display terminals, and structure information useful to programmable or "intelligent" terminals.

The terminal-dependent portion of the system examines the MSGC, consulting a tabular description of the capabilities of the graphics terminal currently being used to decide if any operations need to be performed on the code before it is sent to the graphics terminal. Typical operations include discarding structure information for static terminals and redundant information for intelligent terminals, performing rotations and scalings for terminals lacking these features, attempting compromise operations where necessary, and translating the MSGC to the appropriate characters for controlling the particular terminal.

Graphic input from the terminal is handled in a similar fashion. The terminal interface translates the graphic input into MSGC which is interpreted by the central graphics system and returned to user or application programs as return arguments from a request for input.

The structured data base allows graphic pictures to be represented naturally (e.g., a door knob as part of a door as part of a house as part of a neighborhood), and to be edited efficiently. The terminal-independent MSGC can be stored permanently in a Multics segment, to be "played back" with low computational overhead through a terminal interface at a later time to produce a standard background scene on any terminal type. Also, in many cases, the terminal-dependent code produced by a particular terminal interface can also be stored and played back to that particular terminal type at negligible computational overhead.

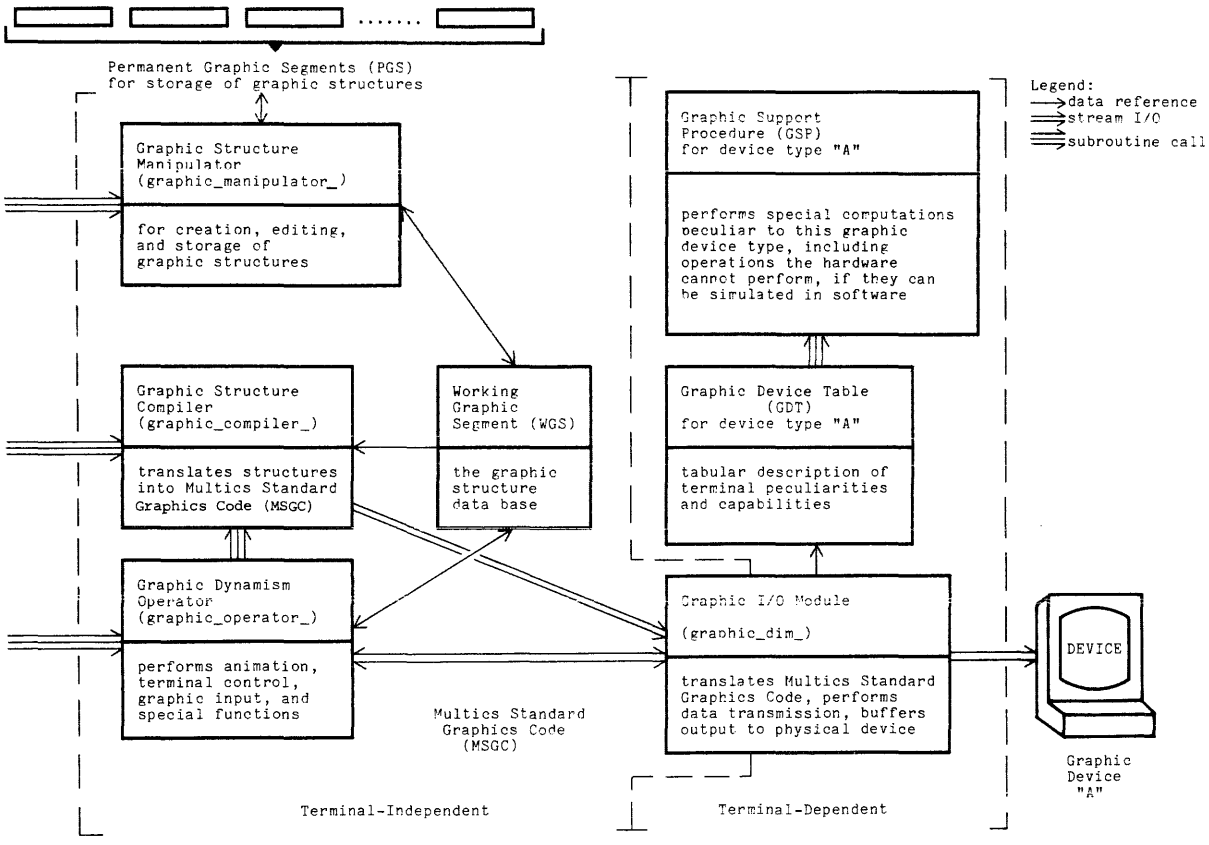


Figure 3-1. Functional Parts of the Multics Graphics System

The tabular description of a graphic terminal's capabilities and peculiarities allows new terminal types to be added to the system with minimum overhead. The ability to specify system-supplied or user-supplied utility routines to aid graphics code translation promotes terminal independence, and provides a handle for extending the basic capabilities of the MGS.

GRAPHIC STRUCTURE DEFINITION

Rather than treat graphic data as an unstructured collection of atomic graphic elements, much as a sketch could be considered an unstructured collection of points, lines, shadings, etc., the MGS deals instead with tree-structured descriptions of pictures, where atomic graphic elements form parts of higher level structures, which in turn may be parts of still higher level structures. Substructures may be shared within higher level structures. This organization has three advantages. First, it allows for fairly natural representation of graphic data. Recognizable objects (automobiles, doors, houses, etc.) can be viewed as both complex graphic entities while they are being created and edited, and as atomic graphic elements when they are being incorporated into larger scenes. Secondly, the ability to share graphic substructures eliminates a great deal of redundancy in specifying a graphic picture. (e.g., all the windows on a skyscraper can be represented by a single window referenced many times in the graphic structure.) Finally, the structured organization makes possible some relatively powerful graphic editing capabilities (such as changing the shape of all the windows below the 34th floor).

Two types of atomic elements make up a graphic structure: terminal graphic elements and nonterminal graphic elements. Terminal graphic elements represent simple graphic operations most often interpreted directly by graphic terminal hardware. These include screen positioning, line and point drawing operations, screen modes (such as blinking, intensity, linetype (e.g., solid, dashed, dotted, etc.), and sensitivity to a light pen), and coordinate rotations and scalings in three dimensions. Nonterminal graphic elements are lists that impose ordering on the elements they contain. Levels of structure are represented by including nonterminal graphic elements within other nonterminal graphic elements. Figure 3-3 depicts a portion of a graphic structure describing a simple house. The picture is structured along functional lines. For illustrative purposes, the single window design is shared in two places, one on each side of the house.

Each graphic element in a Multics segment representing a graphic structure is uniquely identified within the segment by a node value that is used to reference that element within the structure and in later operations. Nonterminal graphic elements are simply coherent lists of node values of other graphic elements (terminal or nonterminal).

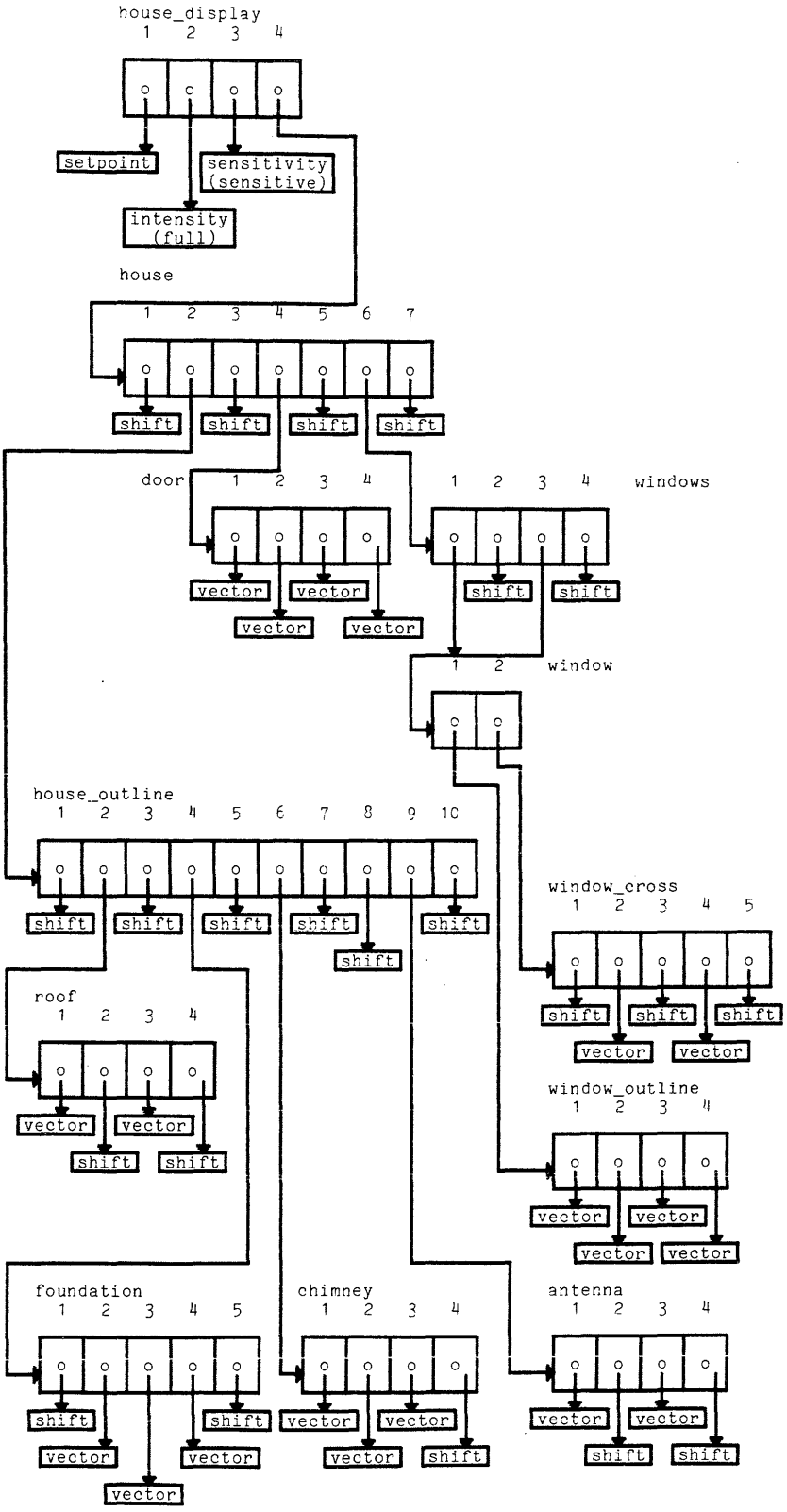


Figure 3-2. A Typical Graphic Structure Organization

In the following descriptions of the different graphic elements, the notation:

element_type (argument1, argument2, ..., argumentn)

is used to convey the essential abstract meaning of each element. It should not be interpreted as the syntax of a subroutine call or other specific implementation property. The actual semantics of subroutine calls for creating and editing graphic elements is described later in this section under "Graphic Structure Manipulation."

Nonterminal Graphic Elements

There are three types of nonterminal graphic elements used in structuring a graphic picture. They are:

- lists
- arrays
- symbols

LISTS

Lists are the most fundamental nonterminal graphic elements. A list is specified by:

list (element1, element2, ..., elementn)

where element is the node value of any graphic element. Lists serve two purposes: to order other graphic elements, and to provide structure to a picture. A list may contain any number of terminal and nonterminal graphic elements.

NOTES: Circular or recursive lists (those that contain themselves or are part of a chain of list reference that eventually leads back to themselves) have undefined meaning and are therefore invalid.

It is possible to refer to a unique element many times within one list or from many different lists. Therefore, there is no concept of a structure being "owned" by a superior structure, since every piece of structure is inherently sharable.

ARRAYS

Use of an array is structurally equivalent to use of a list, but causes all information about the structure of its elements to be lost when the structure is compiled into MSGC. The major use of arrays is to reduce the overhead associated with maintaining and forwarding unneeded structural information. This is useful for static (storage-tube) terminals that do not support dynamic graphics and thus have no use for structural information, and for those substructures that the user does not intend to alter dynamically (e.g., background scenes).

Arrays are treated specially by the graphic compiler. Since the internal structure of arrays need not be preserved past the point of compilation, the graphic compiler optimizes the contents of arrays in several ways. It combines any number of successive nondrawing position-affecting elements (e.g., shifts, invisible vectors, setpositions) into single elements. It applies mapping elements (e.g., rotations, scalings, clipping, masking) during the compilation process, so that these no longer need be computed by a GSP or by a program running in an intelligent graphic terminal. It adjusts all elements inside the array to compensate for the roundoff error that can otherwise occur when elements are translated into MSGC.

A mechanism (the "Graphic Device Table", described later in this section) exists whereby a programmer charged with the task of interfacing a particular graphic device can specify to the MGS that all MSGC sent to his device in the form of lists is to be converted to array form before transmission to the terminal. This frees the terminal programming from having to account for and correctly perform complex structuring operations. When an array is made in this fashion, it is also optimized in the manner described above except that the round-off error can no longer be compensated for, since the necessary precision has already been lost in the initial translation of the structure into MSGC. For this reason, and because most graphic structures are not used in a manner that requires the preservation of their internal structure, it is recommended that arrays be used instead of lists wherever possible.

SYMBOLS

Symbols are a special form of nonterminal graphic element used for naming graphic constructs. A symbol consists of two elements:

symbol (element, name)

where element is the node value of a terminal or nonterminal graphic element, and name is the node value of a terminal text element (see "Terminal Graphic Elements" below) containing the text of the symbol name. Symbols serve several purposes, the primary one being to uniquely identify, in a mnemonic way, graphic constructs that may be moved between several Multics segments. Symbols have their own node values, and do not share the node value of their contents. Operations on these two node values produce different results. For example, the MSGC produced by compiling a symbol node contains the symbol construct, but the code produced by compiling the node value of a symbol's contents does not include the symbol.

Terminal Graphic Elements

Terminal graphic elements are operations often understood directly by graphics terminal hardware or terminal-resident software. The order of appearance of terminal graphic elements in lists or arrays dictates the effect these elements have on other elements in the list.

There are four categories of terminal graphic elements in the MGS. They are:

- positional elements
- modal elements
- mapping elements
- miscellaneous elements

Positional elements affect the screen position (in three dimensions) of what might be thought of as a graphic cursor, (or "current graphic position"), and cause lines and points to be drawn on the screen. All positional elements use the current graphic position as their origin, cause some movement of the beam, and leave the current graphic position at their point of termination. Positions are computed within a virtual screen of 1024x1024x1024 points, with the point (0,0,0) corresponding to the center of the screen. The virtual screen is infinite in all directions but is visible on a display screen only within the limits ($-512 < x,y,z < 511$).

The coordinate system is a right-handed cartesian coordinate system, with the positive x direction toward the right, positive y upwards, and positive z coming out of the screen. Coordinates are supplied and manipulated as fractional quantities to minimize round-off errors in rotation and scaling operations.

There are two types of positional elements: absolute and relative. Absolute positional elements force the graphic cursor to a specific point on the virtual screen. Relative positional elements move the graphic cursor to a new position relative to its current position. The elements are:

absolute positioning	relative positioning
setposition	vector
setpoint	shift
	point

setposition (x, y, z)
: this element sets the current screen position to (x, y, z) without displaying any points or lines.

setpoint (x, y, z)
: this element sets the current screen position to (x, y, z), and displays a visible point.

vector (dx, dy, dz)
: this element displays a vector from the current screen position with dimensions dx, dy, and dz.

shift (dx, dy, dz)
: this element changes the current screen position by dx, dy, and dz with no visible effect.

point (dx, dy, dz)
: this element changes the current screen position by dx, dy, and dz and displays a visible point at the new position.

Relative screen positions are accumulated within a list or array from left to right. Absolute positioning elements (setposition and setpoint) are allowed only in the topmost level structures. Substructures within a list or array may change the screen position, although in general, shared substructures should have a net relative shift of (0,0,0) (i.e., the sum of the relative positioning elements in a shared list or array should normally add up to (0,0,0)).

Modal elements produce no effects on the screen by themselves, but affect the properties of successive graphic elements in defined manners. The appearance of a modal element in a list overrides a previous setting for that particular mode for the rest of that list. The defined graphic modal elements are:

- intensity (brightness)
- line type (solid, dotted, dashed, etc.)
- steady/blinking
- insensitive/sensitive (to a light pen)
- color (red, green, and blue)

intensity (value)
: this element affects the brightness of succeeding graphic elements in a list. Eight levels of intensity (0-7) are defined. Level 0 corresponds to invisible, and level 7 is the default, full intensity.

line_type (type)
: this element causes succeeding vectors to be drawn as solid, dashed, or other machine-defined types of lines. Type 0 is defined as solid (the default), type 1 as dashed, type 2 as dotted, etc.

steady/blinking (value)
: this element causes succeeding graphic elements to be displayed steadily (the default), or to blink.

insensitive/sensitive (value)
: this element causes succeeding graphic elements to be sensitive or insensitive (the default) to detection by a light pen.

color (red_intensity, green_intensity, blue_intensity)
: this element causes succeeding graphic elements to be displayed in the color specified by the intensities of the three primary colors in the additive color spectrum.

Modal elements establish a local graphics environment which governs the properties of lines and points drawn within the scope of that environment. There are several rules governing the application of modal elements depending on structure level and order in a list (or array):

1. When a modal element occurs in a list, it affects all successive elements in that list up to the next modal element of the same type.
2. A modal element overrides a previous modal element of the same type in the same list.
3. The local graphics environment (mode settings, rotations, scalings, and clippings) at the start of a substructure is defined as that environment in effect in the parent list at the point the substructure is referenced. This environment is changed by successive modal elements in the substructure. It is discarded at the end of the substructure and the modes are restored to the current values in the parent list. (In other words, modes are automatically reset to their previous values at the end of a substructure. This makes it impossible to have a substructure that changes the modes or mappings of its parent structure.)

Mapping elements cause no visible effect by themselves, but affect how succeeding graphic elements are mapped onto the screen. There are three mapping elements:

- rotation
- scaling
- clipping

rotation (/x, /y, /z)
: this element causes succeeding graphic elements to undergo a rotation about the x-, y-, and z-axes in that order. These axes are stationary relative to the screen. The units of rotation are positive degrees. Rotations are taken modulo 360 degrees.

scaling (*x, *y, *z)
: this element causes succeeding graphic elements to undergo scaling in the three separate directions defined by the stationary coordinate system. Scalings may be negative to produce mirror images.

clipping (not currently implemented)
: this element causes all succeeding normally visible graphic elements to be clipped (become invisible) if they fall outside a parallelepiped defined by the clipping parameters. If a graphic element straddles the boundary, only the part within the parallelepiped will be visible.

masking (not currently implemented)
: this element causes all succeeding normally visible graphic elements to be masked (become invisible) if they fall inside a parallelepiped defined by the masking parameters. If a graphic element straddles the boundary, only the part outside the parallelepiped will be visible.

Clipping and masking are referred to as "extent elements."

Mapping elements change the local graphics environment in somewhat the same manner as modal elements, according to three rules:

1. When a mapping element occurs in a list, it affects all subsequent elements in that list up to the next mapping element of the same type. A mapping element overrides a previous mapping element of the same type in the same list.

2. When a mapping element occurs in a list, the net mapping is the result of applying the mapping element to the mapping currently active in the parent list.

3. Mapping elements in a sublist have no effect on the mappings in a parent list.

Because mappings are noncommutative vector operations, the order of application of mapping elements to constructs in a list is important. A scene that is first scaled and then rotated will in general appear different from one that is first rotated and then scaled. Within a list, scaling is performed first, then rotation, then extent elements. This order may be overridden by using several levels of structure to achieve the desired order of application. The mappings closest to the object (on the lowest structural level) are most binding, and are applied first. The mapping elements are defined to apply to all graphic elements with the exception of text strings. For efficiency, the central graphics system assumes the use of character generating facilities in the terminal processor. Thus, the orientation and size of text strings are not altered by mapping elements. However, the positions at which text strings occur are altered.

MISCELLANEOUS GRAPHIC ELEMENTS

There are two other graphic elements that may be included in a graphic structure. They are:

- text
  for displaying textual information.

- datablock
  for storing user data within the graphic structure, or extension of the basic capabilities of the MGS.

Text

The purpose of the text element is to allow labels and other textual information to be included in a graphic structure. Its format is:

```
text (alignment, string)
```

where string is a text string of any length (although in general it will be smaller than the text line length of most graphic terminals), and alignment is a number from 1 to 9 which specifies that the text string is to be aligned in one of nine ways relative to the current screen position, as follows:

Alignment	Portion of String at Current Screen Position
1	upper left
2	upper center
3	upper right
4	middle left
5	dead center
6	middle right
7	lower left
8	lower center
9	lower right

NOTE: The string is subject to active screen modes, but not necessarily to mappings. However, the initial position of the string is subject to mappings.

Datablock

The datablock graphic element allows arbitrary program-defined bit strings representing user data to be stored as part of a graphic structure. The data is passed to the graphics terminal just as any graphic effector is, which makes it possible for a user with special applications to have his program construct datablocks that contain terminal-dependent information or commands. This also provides a straightforward and powerful facility for extending the basic capabilities of the MGS by allowing user program-to-graphic-terminal conventions.

The datablock is defined by:

datablock (user_data)

where user_data is a bit string of any length. There are no system-defined type codes for marking the user_data as representing integers, characters, etc., although the user program may define descriptors meaningful to it, and store these as part of the data.

Datablocks have no system-defined effect on other graphic elements.

GRAPHIC STRUCTURE MANIPULATION

Graphic structures are created, edited, and stored in a temporary segment in the user's process directory known as the Working Graphic Segment (WGS). User programs call entry points in the subroutine called graphic_manipulator_ (described in Section 5) to perform several categories of operations on graphic structures in the WGS:

- creation of new elements and structures
- examination of existing structures
- alteration of elements and structures
- permanent storage of named structures

Graphic elements in the WGS are referenced by node values, valid only within the current WGS. When a new graphic element is created, the node value of the created element is returned to the user program as a sort of "receipt". This node value is used in all later references to this element. Lists of graphic elements are created from PL/I-like or FORTRAN-like arrays of node values of the elements in the list. Permanent storage of all or a portion of a graphic-structure is accomplished by attaching a symbol (name) to the structure. Entry points in the Graphic Manipulator can then be used to move such named structures between the temporary WGS and one or more PGSs anywhere in the Multics storage hierarchy.

Node values are used for graphic-structure creation and editing. The central graphic system uses node values as an efficient means of locating graphic elements. Names are used for permanent storage as they are more mnemonic, and as the operation of copying a graphic structure into a PGS may perform an implicit storage compaction and garbage collection function, thereby changing the node values of most graphic elements copied.

Refer to the graphic_manipulator_ subroutine described in Section 5 of this document for the details of the various graphic structure manipulation entry points.

GRAPHIC STRUCTURE COMPILATION

When a graphic structure has been created and satisfactorily edited, a user can then produce a character-string representation of this structure for transmission through the Multics I/O system. The input to the compiler is a graphic structure resident in the WGS. The structure is designated to the graphic-structure compiler by the node value or name of its top-level list. The compiler transforms this structure into an equivalent representation in MSGC, a standard intermediate form that is terminal-independent. This code is written over the I/O switch named graphic_output. (Entries are provided that allow the user to substitute some other I/O switch for the duration of an operation.) This switch may be attached to a terminal interface, thereby directing the code to a particular graphics terminal; or it may be attached to a Multics segment, producing a permanent copy of this terminal-independent code that can be "played back" through any terminal interface at a later time.

Several different entries are provided in the graphic-structure compiler to perform some common operations on the remote terminal (such as erasing the screen, or specifying that the structure is to be loaded into an intelligent terminal's memory, but not immediately displayed). Refer to "Specification of the Virtual Graphic Terminal" in this section for additional information on this subject.

DYNAMIC GRAPHIC OPERATIONS

There are several classes of graphic operations that involve user interaction or take advantage of refreshed display screens and real-time computation in intelligent terminals:

- animation
- graphic input and user interaction
- terminal control

The basic design philosophy relating to such dynamic operations is that the graphic structures resident in the Multics system and those in the graphics terminal memory are isomorphic (structurally equivalent). In other words, there are no provisions for the user or the terminal to make changes in a terminal-resident graphic structure without mirroring them in the Multics-resident structure. All dynamic graphic operations are initiated at the request of a user program or application program in the Multics system.

There are several reasons for adopting this philosophy. First, it allows a simple and well-defined interface to a graphics terminal. Multics programs are never faced with the difficulty of passing arbitrary inputs from a terminal, but need only expect inputs in standard formats, and only in response to an operation that requests information. Second, terminal-resident programming is simplified, reducing the amount of memory required at the terminal. Finally, the problems inherent in maintaining separate copies of a data base (in this case a graphic structure) are eliminated. The nature of the dynamic graphic operators is such that both Multics-resident and terminal-resident structures are identical before and after each operation.

Dynamic graphic operations are initiated by calls to entry points in the graphic_operator_ subroutine, described in Section 5. These entry points emit characters in MSGC to cause a terminal to perform the desired operations and return to the user program any information returned by the terminal.

Animation

Animation involves moving graphic constructs on a terminal screen in a controlled manner, and dynamically changing the structure of a graphic construct being displayed.

The three dynamic operators that accomplish animation are:

- increment
- synchronize
- alter

INCREMENT

The increment operator allows a single positional or mapping element in the terminal memory to be changed some number of times with a specified real-time delay between changes.

increment (node_no no_times delay template)

where node_no uniquely identifies the element to be changed, no_times is the number of times the incrementation is to be performed, delay is the real-time the terminal is to wait between successive increments, and template is a complete graphic element of the same type as the element specified by node_no, whose arguments are the increments to each of the parameters in the element being incremented.

The increment operator is defined to enable asynchronous operation with all other activities at the graphics terminal, including other increments. This allows several graphic constructs to move independently of each other. Note that this incrementation allows only straight-line trajectories to be specified in each occurrence of an increment operator. Curves may be realized by using several separate increment operators.

SYNCHRONIZE

Because several constructs may be moving simultaneously, movements must be coordinated to allow events to be properly sequenced (e.g., balls bouncing off each other). The synchronize primitive simply commands the graphics terminal to complete all operations received prior to the synchronize operator before beginning any subsequently received operators.

synchronize (no arguments)

ALTER

The alter operator effects changes in the structure of graphic constructs already in terminal memory by allowing list elements to be replaced.

```
alter (list_id index new_element)
```

where list_id is the node value of a list already resident in terminal memory, index is the index of the element of the list to be replaced, and new_element is the node value of the new element, which must also be resident in terminal memory. (The indicated list is updated both in the WGS in Multics, and in the terminal-resident structure.)

Graphic Input and User Interaction

There are three operators for graphic interaction with users:

- query
- control
- pause

QUERY

It is often desirable to obtain input from a user that is more easily expressible with a graphic input device (such as a light pen) than by keyboard characters. There are three general classes of graphic input built into the MGS:

1. where (coordinate position) - the user indicates one position in the stationary x,y,z coordinate system.
2. which (structure specification) - the user indicates a particular subtree of a displayed graphic structure.
3. what (new structure) - the user creates a new graphic structure at the terminal and returns it to Multics.

```
query (input_type device_type)
```

where input_type is a code indicating which of the three inputs are desired (1 is "where", 2 is "which", and 3 is "what"), and device_type indicates the graphic input device from which the input is desired. (It may also indicate that the user is to be given a choice of input devices.)

CONTROL

There is also a fairly stylized form of graphic input that allows the user to experiment with the current displayed structure to see what it looks like before reflecting a change to the Multics system. This kind of operation is implemented by use of the "control" dynamic operator.

```
control (device_type node_no)
```

where device_type indicates the graphic input device from which the input is desired (it may also indicate that the user is to be given a choice of input devices), and node_no is the unique ID of a positional modal or mapping element in the terminal memory whose value is to be placed under control of the user via some input device.

A typical use of this facility is to place the endpoint of a line or the starting position of a construct under control of a light pen, to allow the user to move it around, or to place the orientation (rotation) of a scene under control of a trackball. Upon completion of a control interaction, the structure resident in the system is updated to mirror the changes made.

PAUSE

Occasionally it is desirable to allow a step by step progression through a sequence of displays at a user's own speed. If there is no new computation required of the Multics system between steps, there is no reason for an interaction with it between steps. The pause operation causes the terminal to delay processing of subsequently received graphic data until it receives indication that the user is ready to proceed. In this way, all graphic operations for such a session can be preloaded into the terminal and operated with a minimum of Multics interaction.

pause (no arguments)

Terminal Control

There are three housekeeping functions that need to be performed when dealing with graphics terminals:

- screen control
- terminal memory management
- communications control and error handling (described later in this section)

SCREEN CONTROL

All graphics currently displayed on the screen can be erased, and graphic structures resident in the terminal's memory selectively displayed.

Erase

To erase graphics displays from the screen, use the erase operator:

erase (no arguments)

Display

To display graphic structures selected from terminal memory use the display operator:

```
display (node_no)
```

where node_no is the unique ID of the top level of a graphic structure to be displayed. The structure must already be in the terminal memory.

MEMORY MANAGEMENT

New graphic structures can be loaded into terminal memory and structures that are no longer needed can be deleted. Loading is accomplished implicitly by simply using the graphic_compiler_ subroutine to send a new graphic structure to the terminal.

Delete

The delete operator allows individual structures to be deleted, presumably freeing space in terminal memory:

```
delete (node_no)
```

where node_no is the unique ID of the top-level list of a graphic structure to be deleted. If it is zero, all graphic structures in terminal memory are deleted.

Reference

The reference operator is a special operator generated only by the graphic I/O module when communicating with dynamic graphic devices. It occurs within graphic structures, and signifies a reoccurrence, in the structure, of a piece of graphic structure that has already been sent to the terminal and is therefore already in the terminal memory. The reference operator occurs in place of the duplicate structure, which is not sent to the terminal. This not only optimizes transmission time, but allows the programming of the graphic terminal to make a direct shared reference to the previously-loaded piece of graphic structure instead of having to interpret the same structure twice and replacing the old structure with the "new" identical structure.

```
reference (node_no)
```

where node_no is the unique ID of the piece of the graphic structure that occurs again at the given point.

Multics Standard Graphics Code

MSGC allows graphic structures and graphic operators to be represented as character strings suitable for transmission over a Multics I/O switch. It allows the representation of structural information useful to intelligent terminals and redundant information necessary to display shared substructures on nonintelligent terminals.

MSGC is terminal-independent in two senses: it contains no specification of any particular terminal type, and it contains all information necessary to produce graphics on all supported terminals.

The MSGC for a graphic structure is produced by a leftmost tree walk of the structure in the current WGS. Terminal graphic elements are represented simply as a single-ASCII-character element code followed by argument values coded into ASCII characters in standard formats:

element_code arg1 arg2 ... argn

Levels of list structure are represented by nestings of paired parentheses, and include a list/array indicator and a node value followed by the list elements, in order. The node value is retained to aid intelligent terminals in constructing their internal representations of graphic structures and to allow identification of shared substructures. Terminal elements, as well as non-terminal elements, are considered levels of list structure, and are therefore bounded by parentheses and contain a node value. Terminal elements that are not arrays contain an indicator that identifies them as lists. The specification of a (non-array) terminal element as transmitted is:

(list_indicator node_no element_code arg1 ... argn)

The specification of an array terminal element as transmitted is:

(array_indicator node_no element_codea arg1a ... argna element_codeb
arg1b ... argnb ...)

The specification of a list non-terminal element as transmitted is:

(list_indicator node_no subelem1 subelem2 ... subelemn)

where subelem represents subsidiary elements, each surrounded by its own parentheses and containing its own node values.

Other graphic operations (animation, input, etc.) are also represented by a single-ASCII-character operator code followed by arguments:

operator_code arg1 arg2 ... argn

MSGC is designed around the printing ASCII characters (from 40 to 177 octal) to prevent confusion with the ASCII control characters (0 to 37 octal). Element and operator codes occupy the ASCII characters from 40 to 77 octal. Argument values are encoded in the ASCII characters from 100 to 177 octal, with the six low-order bits in each character representing data values.

Table 3-1. ASCII Character Set on Multics Graphic System

	0	1	2	3	4	5	6	7
unused	000 (NUL)							BEL
	010 BS	HT	NL	VT	NP	CR	RRS	RRS
	020	DC1*						
	030							
	040 space	!	"	#	pause $	reference %	increment &	alter '
	050 node-begin (	node-end)	control *	display +	query ,	erase -	synchronize .	delete /
	060 setposition 0	setpoint 1	vector 2	shift 3	point 4	scaling 5	rotation 6	extent 7
	070 intensity 8	line-type 9	blinking :	sensitivity ;	color <	symbol =	text >	datablock ?
permissible range for operator arguments	100 @	A	B	C	D	E	F	G
	110 H	I	J	K	L	M	N	O
	120 P	Q	R	S	T	U	V	W
	130 X	Y	Z	[	\	]	^	_
	140 `	a	b	c	d	e	f	g
	150 h	i	j	k	l	m	n	o
	160 p	q	r	s	t	u	v	w
	170 x	y	z	{	|	}	~	PAD

*in conjunction with other characters, is used to signal beginning and end of graphic transmissions to intelligent terminals.

The following four figures depict the formats used for transmission of argument values in MSGC.

The single-precision integer (SPI) format is used for transmission of small nonnegative values from 0 to 63.

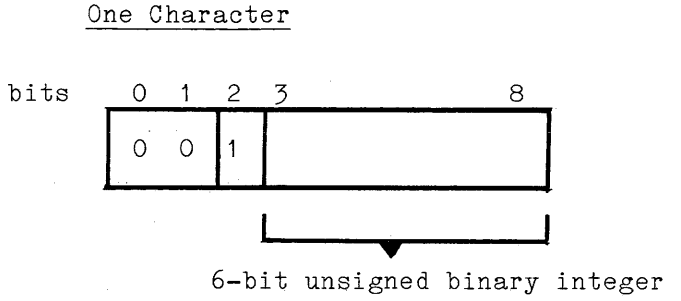


Figure 3-3. Single-Precision Integer Format

The double-precision integer (DPI) format is used for signed integers ranging from -2048 to 2047.

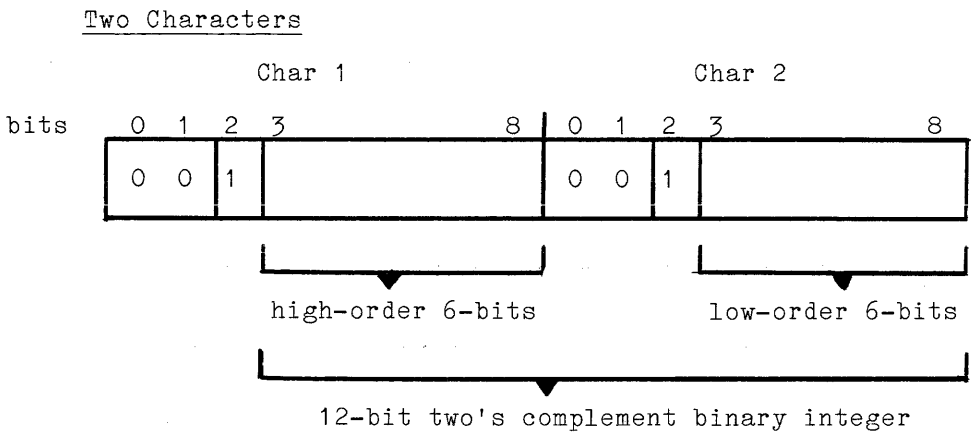


Figure 3-4. Double-Precision Integer Format

The scaled fixed-point (SCL) format is used for numbers with fractional parts. It has the same range as the DPI format, but is accurate to fractional parts of 1/64.

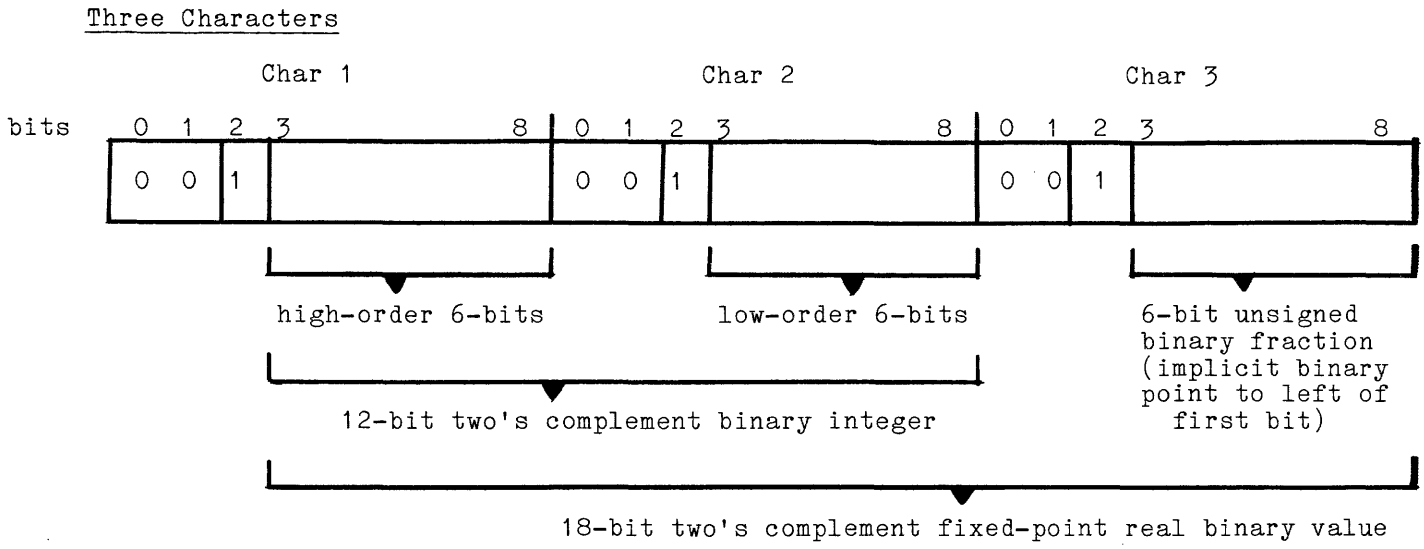


Figure 3-5. Scaled Fixed-Point Format

The unique identifier (UID) format is used to transmit 18-bit node numbers.

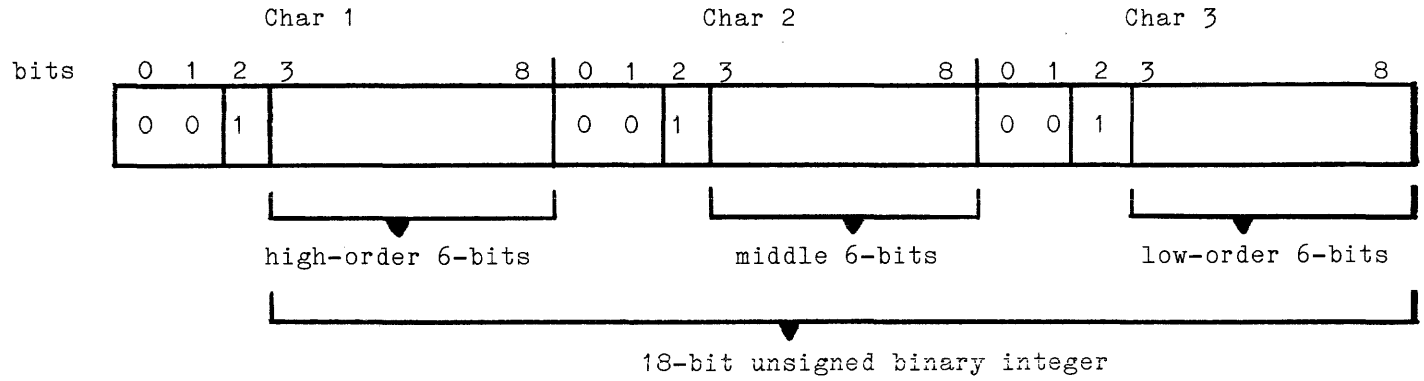


Figure 3-6. Unique Identifier Format

Following are the character codes and argument list formats for the operators in MSGC. (Refer to "Terminal Graphic Elements," "Dynamic Graphic Operations," and Table 3-1 detailed earlier in this section for descriptions of the following operators.)

POSITIONAL OPERATORS

```
setposition ("0")
setpoint    ("1")
vector      ("2")   xpos  ypos  zpos
shift       ("3")   (SCL) (SCL) (SCL)
point       ("4")
```

where xpos, ypos, and zpos are the coordinates of the desired positioning operation.

MODAL OPERATORS

```
intensity ("8") value
                (SPI)
```

where value is a number from 0 (invisible) to 7 (fully visible).

```
line_type ("9") value
                (SPI)
```

where value is one of the following:

0	solid line
1	dashed line
2	dotted line
3	dashed-dotted line
4	long-dashed line
5-63	reserved for expansion

```
blink/steady (":") value
                   (SPI)
```

where value is either 0 (steady), or 1 (blinking).

```
sensitivity (";") value
                  (SPI)
```

where value is either 0 (insensitive), or 1 (sensitive).

```
color ("<") red_intensity green_intensity blue_intensity
               (SPI)         (SPI)          (SPI)
```

where the arguments are the intensities of the three primary additive colors, with 0 representing no intensity and 63 representing full intensity.

MAPPING OPERATORS

scale ("5") xscale yscale zscale
(SCL) (SCL) (SCL)

where xscale, yscale, and zscale are the scale factors along the three stationary coordinate axes.

rotate ("6") xangle yangle zangle
(DPI) (DPI) (DPI)

where xangle, yangle, and zangle are the numbers of degrees of rotation around each of the three stationary axes.

MISCELLANEOUS OPERATORS

text (">") alignment length string
(SPI) (DPI) (ASCII)

where:

1. alignment
is a number from 1 to 9 that specifies the text string is to be aligned in one of nine ways relative to the current screen position, as follows:

Alignment	Portion of String at Current Screen Position
1	upper left
2	upper center
3	upper right
4	middle left
5	dead center
6	middle right
7	lower left
8	lower center
9	lower right

2. length
is the number of characters in the text string.

3. string
is a character string.

datablock ("?") length string
(DPI) (ASCII)

where:

1. length
is the number of data bits to follow.

2. string
is a character string with data bits packed six to a character in the low-order bits.

STRUCTURAL OPERATORS

node_begin ("(") struc_type node_no
(SPI) (UID)

where:

1. struc_type
   is either 0 (list), or 1 (array).

2. node_no
   is the unique ID associated with the list or array.

node_end (")") (no arguments)

symbol ("=") length name
(DPI) (ASCII)

where length and name are the number of characters and the text of the symbol name associated with the immediately following graphic structure.

reference ("%") node_no
(UID)

where node_no is the unique ID of a node already resident in terminal memory that is used in successive references to shared substructures. Users wishing to construct and output their own graphic code should refrain from using this operator, as it will limit their graphic code to intelligent terminals. This operator is normally inserted into the graphic switch at run time by the graphic I/O module.

ANIMATION OPERATORS

increment ("&") node_no times delay template_node
(UID) (DPI) (SCL)

where:

1. node_no
   is the unique ID of a node already resident in the terminal memory that is to be incremented.

2. times
   is the number of times the increment is to be performed.

3. delay
   is the amount of time, in seconds, that the terminal is to delay before each increment.

4. template_node
   is a complete graphic element containing the relative increment to be performed, and including the element code in its own format.

synchronize (".") (no arguments)

alter ("'") node_no index new_node
(UID) (DPI) (UID)

where:

1. node_no
   is the unique ID of the list or array node being altered.

2. index
   is the index in the list of the element to be replaced.

3. new_node
   is the unique ID of the new node to be inserted in the list.

INPUT AND USER INTERACTION OPERATORS

query (",") input_type input_device
(SPI) (SPI)

where:

1. input_type
   is the type of graphic input desired:

   1 where
   2 which
   3 what

2. input_device
   is the graphic input device to be used to generate the indicated input:

0	terminal processor or program
1	keyboard
2	mouse
3	joystick
4	tablet and pen
5	light pen
6	trackball
7-62	reserved for expansion
63	any device

control ("*") node_no
(UID)

where node_no is the unique ID of a node to be controlled by the terminal or user.

pause ("$") (no arguments)

erase ("-") (no arguments)

display ("+") node_no
(UID)

where node_no is the unique ID of the top-level list node to be displayed.

delete ("/") node_no
(UID)

where node_no is the unique ID of a node which is resident in the terminal memory and is to be deleted. If node_no is zero, all nodes are deleted.

TERMINAL-INTERFACING CONSIDERATIONS

One of the features of the MGS is its ability to accept new types of graphics terminals with a minimum of coding. In most cases, the user need only specify the special characteristics of the terminal in a GDT and code a graphic-support procedure (GSP) to perform any code conversion necessary. Then any existing program or graphic file (within the capabilities of the graphic device) can be used and comparable results obtained on the user's own device. Neither the GDT nor the GSP need be installed as part of the graphics system in order to be used in conjunction with it. This section describes considerations pertaining to the specification and creation of GDTs and GSPs for new terminal types.

Specification of the Virtual Graphic Terminal

The terminal-independent portion of the MGS is designed to interface with an ideal device known as the VGT. This terminal possesses all the properties and functionality necessary to perform any valid operation that is defined in the graphics system. Although it corresponds to no one real-world device, it serves to define a standard interface that all graphics terminals on Multics must simulate to the greatest possible degree. Actually, two slightly different configurations of the VGT are defined. The first defines the VGT that may be used for static (e.g., storage-tube) graphics. The second is a superset of the first with abilities to perform dynamic or "intelligent" graphics (animation, control of elements by the terminal, etc.).

The VGT is the "virtual hardware embodiment" of a processor that can operate directly on graphic structures as defined by the MGS. It is responsible for implementing every graphic element exactly as defined in the graphic structure definition, including their various defaults. For example, the VGT normally maintains the state of modes and mappings in their default state and, in the course of normal operation, always returns these modes and mappings to their default states after receiving and processing each complete graphic structure.

The VGT uses a system of right-handed cartesian coordinates as its screen definition. The origin point (0,0,0) lies at the center of the screen. The screen area is regarded as three-dimensional, extending from -512 to 511 on each axis. The entire area of the screen is defined as (1024x1024x1024) points. The relation between points and any measurements of real distance (such as millimeters) is not defined. (In general, this relationship is different for different devices seeking to simulate the VGT.) As the user is facing the screen, the positive x-axis extends to the right, the positive y-axis extends upward, and the positive z-axis "comes out of the screen."

The VGT possesses a hardware character generator that is capable of generating all printable ASCII characters. The dimensions of the character produced by the character generator are undefined. The terminal is able to align the string in relation to the current graphic position by any of nine points on the string. (Refer to "Miscellaneous Graphic Elements" previously described in this section.)

The VGT has a mechanism by which modes and mappings may be "stacked" as the display mechanism searches deeper into a graphic substructure. (The concept of stacked modes and mappings is explained more fully in the descriptions of modal and mapping elements earlier in this section.) It can apply mode and mapping transformations to positional effectors, at display time, to produce new values for these effectors. (The dynamic VGT must not perform these transformations at loading time; ideally, they should be performed by the display hardware.)

The VGT accepts and processes MSGC directly. Because of the format of positional effectors in MSGC, the terminal can handle references to positions outside of the (1024x1024x1024) screen area, to the range of (4096x4096x4096). However, while a picture is being displayed, only the portion within the screen boundaries is visible. The result of any attempt to use an area larger than the maximum (4096x4096x4096) area is undefined.

The VGT for static graphics refuses certain graphic effectors. The "refused" effectors are:

- reference
- increment
- alter
- control

If these effectors occur in the MSGC being sent to a static VGT, they have no effect other than that of an error being returned to the user.

The VGT for static graphics also ignores certain other graphic effectors. The "ignored" effectors are:

- synchronize
- symbol
- data
- delete
- display

NOTE: The transmission of a graphic structure to a dynamic VGT simply "loads" the structure until the "display" effector is received for that structure, whereas the act of transmitting a graphic structure to a static VGT is interpreted as an implicit command to display it.

The dynamic, or intelligent VGT has sufficient processor power to manage its own memory, decode MSGC, and generate meaningful status messages, including error messages, if necessary. Its display mechanism is able to perform calls to display subroutines, (corresponding to possibly shared graphic substructures) to any depth. In addition to the required stack for modes and mappings, it maintains a stack of return addresses to be used in conjunction with these display subroutine calls. The terminal possesses a means of obtaining the contents of this stack on demand, and of analyzing the sequence of subroutine calls which has brought it to any point in the graphic structure, if necessary. It maintains in its own memory an isomorphic (identically-structured) representation of all structures sent to it from the WGS. Nonterminal elements are represented as a list of display subroutine calls, and terminal elements are represented by whatever display commands cause that type of element to be displayed. The terminal creates and maintains a list that correlates addresses of display subroutines in its own memory with the node values of the substructures they represent. Operations on the contents of each piece of substructure are performed in a manner to ensure that the value and structural location of any effector is the same in both terminal memory and the WGS at any point in time. Equal value ensures, for example, that the dynamic incrementation of a vector that is under the influence of some rotation produces an incrementation which itself is rotated. If the rotation were done at load time, and the run-time rotation were discarded, the incrementation would be applied to this different-valued vector. The result would be that a retransmission of the contents of the WGS at this time would actually result in a different picture (the correct picture) being displayed. Similar structural location of effectors ensures that operations such as the alter operation and the "which" graphic input operation (previously described in this section) work correctly.

Graphic Device Table (GDT)

The GDT for a graphic terminal is basically a description of the capabilities of that particular device in terms of the capabilities of the defined VGT. Each effector (graphic element or action) that is defined for the VGT is listed in the GDT, and a description is given specifying how this particular effector is to be handled by this device. Several options are available:

1. The MSGC representation of a particular effector is comprehensible to the terminal, and should be transmitted directly to the terminal with no change. This option is useful in the case of an intelligent graphics device, which may be programmed to understand and decode MSGC directly.

2. The action or element described by this effector is totally unimplementable (or is presently unimplemented) on the terminal, and an error message should inform the user of that fact. This option covers, for example, attempts to perform dynamic movement on storage-tube screens, or requests for graphic input on devices for which no input handler has been written in the GSP.

3. The action or element described by this effector is unimplementable or unimplemented by this device, but may be ignored, since its presence is not entirely necessary to produce an understandable and useful image on the screen. For example, this course may be chosen to discard color effectors from the MSGC sent to a monochromatic terminal, or blinking effectors from the MSGC sent to a plotter. (The availability of this option to users of intelligent devices is somewhat restricted.) Because of the requirements for structural isomorphism, effectors cannot be deleted from the stream sent to an intelligent terminal. Rather, they should be forwarded, and the terminal-resident software should recognize and replace them with no-ops, or some other sort of placeholder in the display list.

4. The effector must be translated from MSGC to some terminal-dependent representation before transmittal. This option allows the creator of a GDT to specify that a GSP entry point be called during the code-transmission phase of processing, at every occurrence of the effector specified. This entry point can do code conversion and any other housekeeping functions necessary to drive the terminal. This is the most exercised option in GDTs which deal with static devices.

5. The effector (for this case, mode and mapping effectors exclusively) cannot be handled by the graphics hardware (or the terminal-resident software) in the environment of "stacked modes and mappings," but must be performed in software. This option causes the entire substructure inferior to the list containing the current effector to be dynamically transformed from a "stacked" list to a graphic array. This operation resolves a graphic list structure into a one-level equivalent structure, with inferior symbol effectors removed, and with modes and mappings explicitly applied and reverted at points where the "hardware" of the VGT would have specified such application and implicit reversion. This frees the GSP from having to maintain a stacked environment for application and reversion of modes and mappings which have to be applied to positional effectors.

The contents of a GDT may also include the name of the GSP (if any is required), the size of the hardware characters generated by that terminal, a default action to be performed for effectors for which no action is explicitly specified, and other information.

FORMAT OF A GDT

A GDT consists of many pairs, each containing one keyword and one or more values to be associated with the keyword. The "major" keywords specify things about the device proper and about the entire GDT. The "minor" keywords specify values for graphic elements recognized by the MGS. These values represent actions to be taken by the graphic I/O module when it encounters the elements described by the keyword. Several actions are possible and provided.

Major Keywords

Character_size
: specifies the character parameters of the terminal as a service to users who may wish to write character-size dependent code (such as a flow-charting program). It must be followed by three values, each representing a dimension of the character. The values, in order, represent the height, width, and spacing of the characters in points. These numbers are interpreted as floating values. If this keyword is omitted, the values all are -1.

Default
: sets the global default action for all minor keywords not specifically mentioned in the body of the GDT. If this keyword is omitted, the global default action is "pass". With the exception of "call", any value described below (following the keywords descriptions) is acceptable.

end
: must appear at the end of the text of a GDT.

Message_size
: specifies the number of characters that may not be exceeded in one transmission to the terminal. This is useful for avoiding input buffer overruns when using intelligent terminals. On intelligent terminals, each message is followed by the request for status character (ASCII 035), and transmission does not continue until the status requested has been received. If this keyword is omitted, it is assumed that the terminal can handle entire graphic messages.

Name
: specifies the name of the graphic device for which the GDT is applicable. It may be followed by any string of up to 32 characters in length. This keyword must be supplied.

Points_per_inch
: specifies the defined number of VGT points per inch on this display. This value is provided for the use of device-dependent software, and reliance on the significance of this value by users will adversely affect the terminal-independence of their applications. This keyword must be followed by a value representing a floating number. If this keyword is omitted, the value is -1.

Procedure
: specifies the segment name containing the GSP. This is the procedure whose entries are specified in uses of the "call" value. If this keyword is omitted, the default segment name is constructed by adding the suffix "_util_" to the value of the "Name" keyword.

Type
: specifies which generic type of graphics terminal is being handled. The only permissible values for this keyword are "dynamic" and "static", signifying a refreshed, intelligent terminal, and a storage tube or refreshed unintelligent terminal, respectively. This keyword must be supplied.

Minor Keywords

Each minor keyword represents an allowable graphic effector within the MGS. The keywords used to describe the effectors are:

alter	increment	scaling
blinking	intensity	sensitivity
clipping	line_type	setpoint
color	node_begin	setposition
control	node_end	shift
data	pause	symbol
delete	point	synchronize
display	query	text
erase	rotation	vector

In addition to the above keywords, there are several that control special actions of the graphic I/O module. These are:

close
specifies the action to be taken by the graphic I/O module when the graphic switch using the specified GDT is closed. Only the call option is valid for this keyword. The entry point is only called once per target-switch, regardless of how many switches using the GDT are closed. (The target-switch is the switch "closer" to the physical device, i.e., "tty_i/o" under normal usage.)

graphic_mode
specifies the action to be taken by the graphic I/O module before it switches the terminal into graphic mode (whenever it encounters graphic data after having processed nongraphic data). The only two values valid for this keyword are "call" and "ignore."

input
specifies the action to be taken by the graphic I/O module when it encounters a request for graphic input. If the call option is specified for this keyword, the actual reading of the data and its translation into MSGC is assumed to be done in the entry called. (Note the definition of the second argument of the entry, "input_string", under "Graphic-Support Procedures" below, and its application to the input keyword.)

modes
specifies the action to be taken by the graphic I/O module when it is requested to change the I/O modes on a graphic I/O switch. This feature enables GSPs to define their own set of I/O modes that they can use to identify and properly compensate for subtle differences between mostly identical devices of the same type. (See "Modes in Graphic-Support Procedures," below.) The only two values valid for this keyword are "call" and "ignore."

open
specifies the action to be taken by the graphic I/O module when it is notified of the GDT to be used. Only the call option is valid for this keyword. The entry point is only called once per target-switch, regardless of the number of switches subsequently using the GDT.

reference
specifies the action to be taken by the graphic I/O module when it inserts a reference effector in the output buffer. Only the call option is valid for this keyword. The entry point called must not return any characters.

text_mode
specifies the action to be taken by the graphic I/O module before it switches the terminal into nongraphic mode (whenever it encounters nongraphic data after having processed graphic data). The only two values valid for this keyword are "call" and "ignore."

Values

call <entryname>
specifies that upon encountering the effector, the graphic I/O module is to call the specified entry in the GSP so that it can perform translation or screen position bookkeeping. The syntax of the actual call is described later in this section under "Graphic-Support Procedures."

error
specifies that the I/O module is to return the error code "graphic_error_table_$unimplemented_effector" to the user. This is useful for flagging attempts to perform dynamic operations on a static terminal.

expand
specifies that the effector cannot be handled by the terminal in a stacked-list-structure manner, and that the list containing the effector must be forcibly expanded into an array. This operation may cause the graphic I/O module to backtrack to the beginning of the enclosing list. If the portion of code in which the effector is found is already in array form (or has been previously expanded by the I/O module into an array) this value has no effect.

flush
specifies that the DIM is to dispatch all its buffered output to the terminal before considering this effector. This is useful for performing the query operation in a correct manner, for example, when one must ensure that the graphic structure that is to be used in the query is actually already loaded and displayed, and not waiting in the transmission buffer.

ignore
specifies that the effector should be discarded and not inserted into the final output switch (e.g., to ignore dotted-line-type effectors at terminals without dotted-line capability).

pass
specifies that nothing is to be done with the effector other than passing it directly to the terminal. It is effectively a no-op, and may be mixed with other values for clarity purposes.

Knowledge of the order in which these actions are checked for and performed by the graphic I/O module may be helpful to create desirable results. This order is as follows:

1. expand
2. flush
3. ignore
4. call
5. error
6. pass

Graphic-Support Procedures

Graphic-Support Procedures (GSPs) support graphic devices whose limited (or complete lack of) local intelligence renders them unable to completely simulate the VGT and interpret MSGC directly. In this case, a Multics-resident GSP compensates for the missing local intelligence. The GSP may perform the entire task of simulating the VGT in the case of nonintelligent terminals, or it may simply supply occasional assistance to an intelligent device that cannot itself perform a certain few defined functions. In either case, the combination of the set of functions performed by the GSP and those performed by the terminal must comprise a complete and correct simulation of the VGT, including initial conditions and defaults. For example, the GSP for a device with no local intelligence must initialize the default values for all modes and mappings prior to every top-level graphic structure received (see "Sample Table for a Static Terminal" below for one means of doing this).

GSPs contain a set of entry points that are called at the times specified in the GDT. Each entry point can perform the housekeeping or code conversion that is necessary to implement the operation or element that causes it to be invoked. The invocation of entry points is performed by the graphic_dim_ subroutine described in Section 5 under the direction of the GDT. The exact conventions of the call, the number of arguments, and their values are explained in the compile_gdt command described in Section 4.

GSPs rarely perform actual I/O. Characters that are produced by code conversion, etc., are returned to the graphic_dim_ as a returned value in one of the arguments to the call. At times, some I/O operation (e.g., attachment of a tape for an offline plotter) may be in order, but this is the exception rather than the rule. At times when this type of operation is likely to occur, the GSP is supplied with an I/O switch name that may be used in the operation.

GSPs may keep any information necessary to describe the state of the terminal. Notification will always be given (if specified in the GDT) when the terminal changes state due to completion of graphics, unexpected terminal I/O, use of the "quit" mechanism, and so on.

The graphic I/O module calls the GSP for a particular effector when the "call <entryname>" value is specified in the GDT. The segment name of the GSP is taken from the name given with the "Procedure" keyword of the GDT, and the entryname is that specified by the "call" value. The entry is called with six arguments, in the format shown below:

```
declare <segname>$<entryname> entry (fixed bin, char(*), char(*),
     fixed bin(21), pointer, fixed bin(35));

call <segname>$<entryname> (operator_value, input_string, output_string,
     chars_out, state_ptr, code);
```

where:

1. operator_value (Input)
   is the decimal value of the internal representation of the character in MSGC which represents the operator.

   A list of these values, including values for the special minor keywords which are not graphic operators follows:

36	pause	47	delete	58	blinking
37	reference	48	setposition	59	sensitivity
38	increment	49	setpoint	60	color
39	alter	50	vector	61	symbol
40	node_begin	51	shift	62	text
41	node_end	52	point	63	data
42	control	53	scaling	64	input
43	display	54	rotation	65	graphic_mode
* 44	query	55	clipping	66	text_mode
45	erase	56	intensity	68	open
46	synchronize	57	line_type	69	close
				70	modes

2. input_string (Input)
   has various meanings depending on operator_value:

   For operator_value <= 63, input_string is the operator and all its parameters (in MSGC) being acted upon.

For operator_value = 70 (modes), it is a string representing a set of I/O modes.

For all other values of operator_value, it is the name of the I/O switch on which the input or output is to be performed. This is not the switch leading into the graphic I/O module (such as graphic_output under normal use), but the switch closer to the physical device (such as tty_i/o under normal use).

3. output_string (Output)
is the entire unused portion of the graphic I/O module's output buffer provided for the characters output by the GSP. Since this usually represents the better portion of a segment, it is important that support procedures reference this string with the PL/I "substr" pseudo-variable.

4. chars_out (Output)
is the number of significant characters which the GSP is returning to the graphic I/O module.

5. state_ptr (Input/Output)
is a pointer that is saved by the graphic I/O module and presented to the GSP on each call. The GSP can use this pointer to identify and differentiate between various allocations of state variables that may be active in the same process (e.g., if the process is running two graphic devices of the same type). Typically, the GSP allocates and initializes a generation of state variables when called at its open entry, and sets state_ptr so that they may be located on subsequent calls.

6. code (Output)
is a standard error code.

MODES IN GRAPHIC-SUPPORT PROCEDURES

The creator of a graphic-support procedure may define a set of I/O modes that are to be managed by the GSP. This feature enables a GSP to identify and properly compensate for subtle differences between similar devices of the same type (e.g., special hardware options that implement additional modes or different paper sizes for a plotter).

When the graphic I/O module is requested to change the I/O modes of a graphic I/O switch, it first attempts to pass the entire string of modes to its target switch. If the target switch accepts them, it returns. If the target switch refuses them, the graphic I/O module checks to see if the GDT has specified an entrypoint in the GSP that accepts I/O modes. If one is specified, the graphic I/O module breaks up the mode string into separate single mode specifications. Then, after having saved the current I/O modes of both the target switch and the GSP, it feeds each of the new modes (in turn) first to the target switch, and then (if refused) to the GSP. If both refuse the mode, the previous modes of the target switch and the GSP are restored. Otherwise, the operation is successful.

The GSP entrypoint that implements the modes operation must be able to accept an arbitrary string of modes of the form:

mode1,mode2,mode3...modeN

which may also be the null string. If the GSP does not recognize a particular I/O mode, or if some other error condition is encountered, an appropriate error code must be returned by the GSP. In any event, the GSP must return (in the output_string argument) a similar string representing the state of the modes after completion of the operation, and must set the chars_out argument to the appropriate value.

Since each I/O mode is presented to the target switch before it is presented to the GSP, care should be taken by the creator of a GSP to avoid defining mode specifiers that conflict with I/O modes used by system I/O modules.

EXAMPLES OF GDTS

Following are three examples of proper GDTs, for devices of differing capabilities.

The first is an example of a GDT for a typical static graphics terminal. The terminal has no intelligence and no remote memory; therefore the structuring information in MSGC is not useful, and the GDT creator elects to force all graphic data to be arrays rather than lists by expanding on all node_begin effectors. (Because of this, his GSP need not handle application of mappings, handling of zero-intensity, accounting for structural levels, and implicit reversions of other modes due to structuring of graphic objects.) Code conversion is necessary to convert positional effectors in MSGC into the format understood by the terminal. The device supports some graphic input, and possesses certain optional capabilities which the GDT creator decides to describe in terms of modes.

Sample Table for a Static Terminal

```
Name:                Sample_Static;

Type:                static;
Procedure:           static_device_n;

Character_size:      16, 13, 3;
Points_per_inch:     100.0;

/* Effector          Action */

setposition:         call position;       /* do code conversions */
setpoint:            call position;
vector:              call position;
shift:               call position;
point:               call position;

scaling:             error;               /* should never appear */
rotation:            error;               /* in arrays */
clipping:            error;

intensity:           ignore;              /* only one intensity on this device */
line_type:           call line_type;      /* set proper line type */
blinking:            ignore;              /* can't blink */
sensitivity:         ignore;              /* no light pen */
color:               ignore;              /* no color on this device */

symbol:              ignore;              /* unimportant to static device */
text:                call text;           /* put out hardware character string */
data:                ignore;

pause:               flush, call pause    /* put out buffer and wait for user */
reference:           error;               /* should never occur */
increment:           error;
alter:               error;
node_begin:          expand, call init_state;      /* make arrays out of
                                                      everything */
node_end:            ignore;
control:             error;
display:             ignore;
query:               call query;          /* prepare for graphic input */
erase:               call erase;
synchronize:         flush;
delete:              ignore;

input:               call input;          /* process graphic input */

text_mode:           call mode_switch;    /* prep terminal for */
graphic_mode:        call mode_switch;    /* proper mode */

open:                call open;           /* create state variables */
close:               call close;
modes:               call changemode;

end;
```

The second is an example of a GDT for a dynamic graphics terminal with limited intelligence. The terminal has remote memory and some capabilities for display subroutines; however, it has no mapping hardware and insufficient memory to properly simulate such hardware. Additionally, its display subroutine capabilities do not include storing and reloading the state of any modes except sensitivity, when entering and exiting display subroutines. For this reason, the GDT creator specifies expansion of all lists that contain any mappings or any other modes. Most other effectors are passed directly to the terminal, even including those that represent unimplemented modes or structural items (e.g., color). This is done to preserve the ordinality of elements within lists, so that operations such as "alter" and "which input" may be performed properly. (The terminal programming is assumed to use some sort of placeholding no-op in terminal memory to represent the unimplemented effectors.)

Sample Table for a Semi-intelligent Terminal

```
Name:               Semi_Intelligent;

Type:               dynamic;
Procedure:          dynamic_device_n;
Message_size:       300;
Default:            pass;               /* terminal understands MSGC */

Points_per_inch:    76.35;

/* Effector         Action */

setposition:        pass;
setpoint:           pass;
vector:             pass;
shift:              pass;
point:              pass;

scaling:            expand, error;      /* once expanded, */
rotation:           expand, error;      /* should never appear */
clipping:           expand, error;      /* in arrays */

intensity:          expand, pass;
line_type:          expand, pass;
blinking:           expand, pass;
sensitivity:        pass;
color:              pass;               /* no color on this device */

symbol:             pass;
text:               pass;
data:               pass;

pause:              pass;
reference:          pass;
increment:          pass;
alter:              pass;
node_begin:         pass;
node_end:           pass;
control:            flush, pass;
display:            pass;
query:              flush, pass;
erase:              pass;
synchronize:        pass;
delete:             pass;
```

```
input:              pass;

text_mode:          call mode_switch;   /* prep terminal for */
graphic_mode:       call mode_switch;   /* proper mode */

open:               call open;          /* create state variables */
close:              call close;
modes:              ignore;

end;
```

The third is an example of a GDT for a device with the capability of completely simulating the Virtual Graphics Terminal. It possesses either the hardware to perform VGT operations directly and in a structured manner, or has been programmed to simulate the proper operation in software.

Sample Table for a VGT Simulator

```
Name:               Intelligent_device;

Type:               dynamic;

Character_size:     20, 12, 2.4;
Points_per_inch:    121.45;

Default:            pass;

end;
```

Terminal-Resident Programming

In order to use an intelligent graphics device with the MGS, some programming must be performed in the device itself. The apparent level of intelligence of a device depends on how closely the combination of terminal hardware and terminal-resident software approximates the operation of the VGT. The MGS does not differentiate between the abilities of the hardware and the resident software of an intelligent terminal, as long as the inputs to the "black-box" are well-defined in the GDT, and the output on the screen is a reasonable representation of the graphic structure that was sent.

A terminal possessing a refreshable screen and a programmable display processor does not necessarily always qualify for representation as an intelligent terminal. The most important requirement of a dynamic terminal is that it have (or be able to simulate having) the ability to perform display subroutine calls. Once the ability to represent references to graphic substructures as display subroutines (the basis of structural isomorphism) can be provided, most dynamic operations are possible.

Although the VGT is rigidly defined, there are a number of decisions available to the programmer of an intelligent terminal that must be made. The few examples which follow are not exhaustive:

1. The VGT ignores the datablock effector. For special functions of an intelligent device for which the graphic system has no other provisions for exploitation, the terminal programmer may choose to use some special coded command contained in a datablock.

2. The definition of "what" input is purposely broad. Most programmers of graphics devices choose to implement some subset of allowable "what" inputs on their terminals, such as input from the keyboard, a "where" type input, etc. A programmer may also choose to define the "terminal processor" input device for the terminal as performing some special function (such as allowing the terminal user, through some protocol, to construct entire graphic substructures to be returned to the Multics program).

3. The terminal programmer must decide whether to return an error status or to use some equivalent existing device, should input be requested from an unimplemented graphic input device.

4. The terminal programmer must decide whether to implement asynchronous incrementation.

Formats for Input Information

Graphic input information and terminal status codes have defined formats (shown below), of which the programmer of intelligent terminals must be aware. These formats are described in the include-file "graphic_input_formats.incl.pl1" (see Section 8).

Table 3-2. "where" Input Format

Character Position	Format or Literal	Represents
1	"("	a node-begin character
2	SPI	an array indicator ("A")
3-5	UID	the zero node ID ("@@@")
6	"0"	a "setposition" indicator
7-9	SCL	the returned x-coordinate
10-12	SCL	the returned y-coordinate
13-15	SCL	the returned z-coordinate
16	")"	a node-end character
17	<NL>	a newline character (ASCII 012)

Table 3-3. "which" Input Format

Character Position	Format or Literal	Represents
1	"("	a node-begin character
2-4	UID	node ID of the top-level node in the structure which was selected
5	SPI	integer representing the depth, in levels, of the component selected
6-7	DPI	index in top-level list of inferior substructure selected
8-9	DPI	index in next-level list (described by last index) of inferior substructure selected
etc. to correct number of indexes (determined by depth indicator)		
(N-1)	")"	a node-end character
(N)	<NL>	a newline character (ASCII 012)

Table 3-4. "what" Input Format

Character Position	Format or Literal	Represents
1	"("	a node-begin character
2	SPI	integer representing the input device actually used
3 to (N-2)	(see Note below)	graphic code portion of the input
(N-1)	")"	a node-end character
(N)	<NL>	a newline character (ASCII 012)

NOTE: The "graphic code portion" of the returned string must be a valid string of MSGC. All graphic structures must begin with a "node begin" character and end with a "node end" character. It should not contain any effectors that may not be found inside a graphic structure (e.g., delete and erase). The node IDs returned need not have any relationship with node IDs currently in the WGS -- they may be generated sequentially, for instance, if desired. However, the node IDs returned are checked for uniqueness for the duration of one input message. If one block of "what" input contains more than one structure with the same node ID, it is assumed that the reference signifies a shared relationship between the two occurrences of the substructure, and the contents of the new substructure replaces the old. Multiple (shared) references to the same substructure may be done either in this manner or by the use of the reference effector. The node IDs created by the decompilation process and insertion into the WGS have no relation to the node IDs supplied in the input.

CONTROL INPUT FORMAT

No special format exists for control input. The string returned must be a valid string of MSGC. It should consist of exactly one effector, the contents of which include the effector code and the new values that result from the control operation, in exactly the same format in which it could have been output to the terminal. The returned string must be followed by a newline character (ASCII 012).

Communications Control and Error Handling

There are several problems that fall under the heading of communications control. It is necessary to distinguish character strings representing graphic structures and operations from normal text. Since most intelligent terminals are minicomputers with limited memory and terminal communication buffers, there will often be limits on the speed with which the terminal can process incoming graphics. And because fairly complex structures are being transmitted, some high-level protocol for discovering and reporting errors to the Multics system is necessary.

For intelligent terminals, two ASCII control sequences are defined to have the following meanings:

(let "#" represent the character "DC1", octal 021)

```
#A   enter graphic mode
#B   enter text mode
```

where:

1. "#A"
   indicates that all subsequent characters should be interpreted as representing graphic structures and operators.

2. "#B"
   indicates that succeeding characters are normal text.

The problems of finite terminal input buffers and error reporting are solved by a Multics output buffering and status reporting protocol. The GDT describing a terminal indicates the size of the terminal's input buffer. The strategy is to dispatch no more than this number of characters to the terminal, followed by a request for status character (ASCII 035). The terminal then responds with a status message in a standard format preceded by a left parenthesis ("(") and followed by a right parenthesis and a newline character (")<NL>").

Table 3-5. Status Message Format

Character Position	Format or Literal	Represents
1	SPI	error code for discovered error
(If the error code is zero, meaning no errors detected, the following characters need not be sent.)		
2	ASCII	character code of graphic element in which error occurred
3-5	UID	unique ID of top-level node in graphic structure in which error was detected
6	DPI	depth of error in list structure
7	DPI	list index of top level list element
8 on	DPI	list index of each succeeding element until done

If the error code returned is 0, then the next buffer of characters is output to the terminal. Otherwise, the error is reflected back to the user program and the as-yet-unsent characters are discarded.

A list of the short numeric error codes defined for use by an intelligent graphics terminal and their corresponding definitions in graphic_error_table_ may be found in the include-file "graphic_terminal_errors.incl.pl1" (see Section 8).

Because of the way status handling is performed, an intelligent graphic terminal must return its status string before any other characters, such as responses to queries and controls. However, the terminal should not return the status string before all queries and controls are fully completed, so that the possibility of errors occurring from these operations may be handled. If the implementor of a graphics protocol on an intelligent terminal feels that buffering a variable number of these input strings for return after the status message is a problem, constructs in the terminal's GDT (e.g., "flush") may be used to ensure that multiple inputs or queries do not occur within one graphic message.

Many graphic elements must be sent immediately to the terminal because they require terminal response before more graphic data is generated. However, it is desirable to keep the frequency of status request interactions to a minimum because half-duplex communications protocols insert rather substantial delays. Control over when the Multics output buffer is sent is exercised in two ways. First, in the GDT describing a terminal, it can be specified for each graphic operator in MSGC whether the buffer must be sent when this operator occurs. Normally, the buffer must be sent only on query and control operators, where input from the terminal is necessary. Secondly, an entry point in the graphic_operator_ subroutine (described in Section 5) sets an internal mode known as the "immediacy" mode. When immediacy is turned on, all graphic operators are sent immediately as they are generated, each followed by a request-for-status message. When immediacy is off, graphic output is buffered until the buffer is full or until a graphic operator is encountered that must be sent immediately, in which case the entire buffer is sent.

GRAPHIC CHARACTER TABLE (GCT)

A Graphic Character Table (GCT) is a description of a character set that provides enough information for the MGS to draw (stroke) the characters. Unlike the text element, which relies on the hadrware character printing capability of the graphic terminal, characters produced from GCTs may be scaled, rotated, and otherwise manipulated like any other piece of graphic structure.

Each entry in a GCT describes the representation of one character of the set. The character is described as a collection of shifts and vectors. Each character carries with it its own spacing (margin) both to the left and to the right. In general, the initial graphic position, before any character is drawn, is considered to be at the upper-left-hand corner of an imaginary box surrounding the character and its desired margins. The shifts and vectors prescribed for each character must move the graphic position from that initial position, draw the character, and set the final graphic position to the upper-right-hand corner of the box.

The imaginary box represents a character position. The width of the box may vary for different characters. The height of the box is constant for all the characters in a single character set. However, for any one character set, the height of the box in points is arbitrary and may be chosen for the convenience of the creator; thus, it may vary from set to set. However, no dimension of the box may exceed 511, nor may any dimension of a shift or vector comprising any character be outside the range $-512 \leq n \leq 511$.

For each character set, the height of the box is defined by a character chosen as the "metric" for that character set. The metric character is one that is defined to extend exactly to the top and bottom of the desired box. The character "0" is usually used as the metric. It is allowable for characters to extend above or below the bounds of the defined box. For example, many lowercase letters such as "p," "g," or "j" will generally extend below the bottom of the box; special characters such as parentheses may often extend above.

Certain characters in all character sets are considered "special format characters" and are not defined in GCTs. These characters are the carriage return, linefeed, space, tab, backspace, and underscore. For various reasons, the handling of these characters (e.g., width and appearance) must be computed at run time and cannot adequately be described in a table.

Format of a GCT Source Segment

The source for a GCT must reside in a segment with the suffix ".gct". The compile_gct command compiles a GCT source segment into a graphic character table. See the description of the compile_gct command in Section 4 for more information on compiling GCT source segments.

A GCT source segment consists of an optional metric statement followed by a number of character descriptions. Comments must be enclosed by "/* ... */" and may appear anywhere.

The syntax of the metric statement is:

```
metric char
```

where char is the name of a character that is to serve as the metric for this character set.

A character description is composed of the name of a character followed by a colon, a list of vectors and shifts describing the character, and an end statement. The character names used in a GCT must be selected from a list of character names known to the compile_gct command. These may be found in the PL/I include-file "gct_char_names.incl.pl1". Most character names are straightforward: "A" for the character "A," "g" for the character "g." Any character can be defined in a GCT except the special format characters listed above. Creators of GCTs should refer to the include file to obtain the names of nonalphabetic characters.

The syntax of vectors and shifts is:

```
vector x_len y_len
shift x_len y_len
```

No punctuation is allowed other than whitespace.

The syntax of the end statement is:

```
end
```

It is required at the end of each character description. No special end statement is needed to signify the end of the character table.

Character tables specified in this manner are usually used with the graphic_chars_ subroutine by specifying their names in a call to graphic_chars_$set_table. Occasionally, however, a user may wish to write a program that interprets the contents of a GCT directly. The internal format of a GCT is described here. All of the structures described below may be found in the PL/I include-file "graphic_char_dcl.incl.pl1".

Every GCT contains two segdefs, named "character_sizes" and "char_ptr". These segdefs (which are similar to entry points) may be located by the use of the hcs_$make_ptr subroutine.

The segdef character_sizes contains storage that is described by:

```
dcl 1 character_sizes based,
      2 height fixed bin(35),
      2 width fixed bin(35),
      2 margin_adj fixed bin(35);
```

where:

1. height
   is the height of the metric character in points.

2. width
   is the width of the metric character in points, including margins. Note that the width of the metric character in no way constrains the width of any other character in the set.

3. margin_adj
   is a negative number, representing the negative of the sum of the margins of the metric character, in points.

The segdef char_ptr contains storage that is described by:

```
dcl char_ptr (0 : 127) pointer based;
```

where:

1. char_ptr (i)
   is a pointer to the character description of the (i+1)'th character in the ASCII collating sequence.

Each character description pointed to by an element of char_ptr has the following description:

```
dcl 1 graphic_char_structure aligned based,
    2 header_word aligned,
      3 (n_elements,
         width,
         left_margin,
         right_margin) fixed bin(8) unaligned,
    2 word_align aligned,
      3 move_type (0 refer (graphic_char_structure.n_elements))
                   bit(1) unaligned,
    2 coords (0 refer (graphic_char_structure.n_elements)) unaligned,
      3 (x_length,
         y_length) fixed bin(8) unaligned;
```

where:

1. n_elements
   is the total number of vectors and shifts in the character description. If n_elements is -1, this character is one of the special format characters listed above, and special computation must be performed to implement it properly.

2. width
   is the width of this character (including margins) in points.

3. left_margin
   is the left margin of this character in points.

4. right_margin
   is the right margin of this character in points.

5. move_type (i)
   = "1"b if the i'th element of the character description is a vector;
   = "0"b if the i'th element of the character description is a shift.

6. x_length (i)
   is the x dimension of the i'th vector or shift.

7. y_length (i)
   is the y dimension of the i'th vector or shift.

SECTION 4

COMMANDS

COMMAND DESCRIPTIONS

This section contains descriptions of the Multics graphics commands, presented in alphabetical order. Each description contains the name of the command (including the abbreviated form, if any), discusses the purpose of the command, and shows the correct usage. Notes and examples are included when deemed necessary for clarity.

The commands are:

1. compile_gct
   compiles a segment containing the source of a Graphic Character Table.

2. compile_gdt
   compiles binary versions of Graphic Device Tables.

3. graphics_editor, ge
   an interactive tool which creates and edits graphic structures.

4. remove_graphics, rg
   terminates a working session.

5. setup_graphics, sg
   initializes the process environment for a particular graphics terminal.

Name: compile_gct

The compile_gct command compiles a segment containing the source of a GCT.

Usage

compile_gct segname {-control_args}

where:

1. segname
   is the name of a segment containing the source of a GCT. The segment name must contain the suffix ".gct". If this suffix is not an explicit part of segname, it is assumed.

2. control_args
   may be either of the following control arguments:

   -check, -ck
   specifies that the ALM assembler is not to be invoked, and that the intermediate assembler source file is to be retained.

   -list, -ls
   specifies that the ALM assembler is to produce a listing of the GCT it creates.

Notes

The compile_gct command compiles a GCT source segment into an assembly language source segment. The Multics ALM assembler is then invoked internally to assemble this intermediate segment into a GCT. The final segment produced has the name of the source segment without the suffix ".gct".

See Section 3 for a description of Graphic Character Tables.

Name: compile_gdt

The compile_gdt command causes a segment containing the source of a GDT to be compiled.

Usage

compile_gdt segname {-control_args}

where:

1. segname
   is the name of a segment containing the source of a GDT. The segment name must contain the suffix ".gdt". If this suffix is not an explicit part of segname, it is assumed.

2. control_args
   may be either of the following control arguments:

   -check, -ck
   specifies that the ALM assembler is not to be invoked, and that the intermediate assembler source file is to be retained.

   -list, -ls
   specifies that the ALM assembler is to produce a listing of the GDT it creates.

Notes

A GDT source segment is conventionally named with the name of the graphic terminal it describes, with the suffix ".gdt" (e.g., "tek_4014.gdt" or "ards.gdt"). The compile_gdt command compiles a GDT source segment into an assembly language source segment. The Multics ALM assembler is then invoked internally to assemble this intermediate segment into a GDT. The final segment produced has the name of the source segment without the suffix ".gdt".

See Section 3 for a description of Graphic Device Tables.

Name: graphics_editor, ge

The graphics_editor is an interactive tool that may be used to create and edit graphic structures. It is capable of storing these structures into, and retrieving them from, permanent graphic segments.

Usage

graphics_editor {seg1} {seg2} ... {segn}

where segi is a pathname specifying a segment to be read into the graphic editor. This segment must contain the suffix ".ge". If the suffix is not explicitly stated in the command line, it is assumed. This segment may contain a list of editor commands or assignments, in the same format as they might have been typed into the editor interactively. The segments are interpreted by the editor in the order specified.

Notes

If errors occur while reading segment specified on the command line, processing of that file ceases.

When graphics_editor is ready to receive input from the user's terminal, it replies with "Edit.". The user may then begin to issue requests.

The Command Language

Requests fall into two categories: commands and assignments. In general, commands may be terminated with either a semicolon (;) or a newline. Assignments (due to their ability to be quite lengthy) may be terminated only with a semicolon. However, sometimes one or more of the arguments of a command may be an assignment.

The formal rules for statement terminator parsing are:

1. A semicolon is always required to end a statement if it contains any assignments.
2. A newline does not automatically terminate a statement as long as:
   a. an assignment is pending, or
   b. there exist some unclosed sets of parentheses, or
   c. the last token on that line is a comma.
3. Newlines regain their ability to become statement terminators when:
   a. all parentheses have been closed and
   b. the last token on the last (current) line is not a comma.

4. Newlines never regain their ability to become statement terminators once an assignment is performed within a statement. (This is simply a restatement of rule 1.)

As a general rule: When in doubt, type semicolon.

Comments are enclosed by "/* ... */" and may be interspersed with any input lines.

Symbols

Symbols in the graphics_editor are alphanumeric representations of node values. A node value is a "receipt" returned by the graphics system whenever it is asked to create some graphic element. Symbols have a value that consists of exactly one such node value. (For a more complete description of node values, refer to Section 3.)

Symbols may be divided into four classes:

- system symbol -- predefined and represents a primitive operation or element
- user symbol -- is defined by the user at some time with an assignment
- macro -- is defined by the user, but takes "arguments," and has no permanent value of its own
- system macro -- is a macro defined by the system.

SYSTEM SYMBOLS

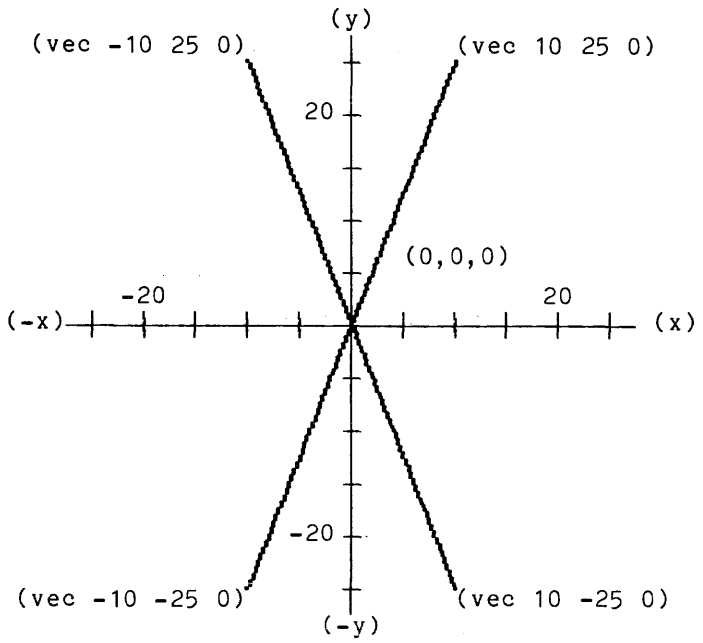
System symbols have no permanent value. They take one or more arguments, either implied or explicit. The use of a system symbol represents a request that a new element be created. The node value returned from that creation is then used in any subsequent operation of that particular expression.

Examples of system symbol expressions are:

1. vector 10 25 — a vector of length (10, 25, 0) (see "Illustration A" below).
2. "Axolotl" uc — a text string containing the string "Axolotl", aligned by the upper center edge.
3. array (a,b,c) — an array containing the nodes represented by user symbols a, b, and c. (See "Tuples," below.)
4. lin dotted — a mode element for dotted lines.

graphics_editor

Illustration A



A list of system symbols and descriptions of their use may be found at the end of this command description.

USER SYMBOLS

User symbols may be up to 32 characters in length, and may consist of any combination of uppercase and lowercase alphabetics, numerals, and the underscore ("_"), provided that the first character is nonnumeric. System symbols, system macros, and commands are considered "reserved words," and may not be used as user symbols. Attempts to define commands as symbols result in ill-formed execution of those commands.

Examples of user symbols are:

1. foo
2. Front_porch
3. bolt_23w9

User symbols are stored in the graphic symbol table of the WGS. They are transferred to and from PGSs whenever the save, use, put, and get system commands are used. (For a complete explanation of graphic symbols, see Section 3.)

MACROS

Macros are user symbols which take arguments like system symbols. Whenever a macro expression is evaluated, the arguments supplied are substituted for the dummy arguments with which the macro was defined. Macros must be defined by macro assignments. For example:

```
macro diamond x y = vec x y, vec -x y, vec -x -y, vec x -y;
```

defines a macro named "diamond" with dummy arguments x and y. The reference:

```
diamond 10 30
```

represents a diamond 10 units in x and 30 units in y, and is exactly equivalent to the expression:

```
vec 10 30, vec -10 30, vec -10 -30, vec 10 -30
```

Macro definitions must be on a single line. Macro names are stored in the graphic symbol table of the WGS, and may be transferred to and from PGSs with the save, use, put, and get commands.

SYSTEM MACROS

System macros are provided so that the user may construct commonly used graphic objects (such as circles and boxes) that are not primitive graphic objects.

System macros possess some of the properties of both macros and system symbols. They take arguments, either actual or implied. Like system symbols, they always represent single graphic elements (one-tuples). However, the element is not a primitive nonterminal graphic element, but is an array of many other graphic elements, as may be done with a macro. Unlike a macro, though, once it is used and "expanded" it does not disappear (e.g., when a structure created with a macro is replayed); rather, the system keeps track of the macro and replays it in the manner in which it was typed in.

Tuples

A tuple is simply a group of one or more values. Every complete symbol (i.e., a user symbol, or a macro, or system symbol with its arguments) is a tuple in itself (a one-tuple). A tuple of more than one element may be expressed as its elements separated by commas such as:

```
a, b, b, vec 10 4 3, intensity 1, xxx
```

This is a tuple of 6 elements.

A tuple which has more than one element represents more than one graphic entity. Therefore, it cannot have one node value. To convert a tuple to a single graphic entity, two system symbols are available: array and list. These two "functions" gather the elements of the tuple into a graphic array, or a graphic list, respectively. (For a more complete explanation of graphic arrays and lists, see Section 3.) The creation of this array or list produces a node value that may be assigned to a user symbol, or may be used without assignment in some larger expression. For example:

```
one_array = array (a, b, c, d, b);
```

is an assignment that creates a graphic array with the elements (a, b, c, d, and b), and assigns to "one_array" the value of this list.

Assignments

An assignment is an operation that extracts the value of one tuple and assigns it to another tuple. The assignment operator is the infix "=" sign.

The simple assignment:

```
foo = bar;
```

specifies that the value of foo is to become the symbol bar. An important point to keep in mind is that this does not mean that foo and bar both refer to the identical piece of graphic structure. Rather, foo contains bar, and (of course) indirectly also contains the entire structure contained by bar. If foo is undefined at the time of assignment, it is created. If it had a previous value, that value is replaced. Any other graphic structures that referenced foo still refer to it, but now contain (indirectly) its new value. (It is possible to assign the value of a symbol to another symbol, rather than assigning one symbol to another; this operation is discussed below under "Qualified Expressions.")

In general, only tuples of like dimensionality (i.e., having the same number of elements) may be assigned to each other. For example:

```
a, b, c = d, e, f;
x = array (p, q, r);
```

are both valid assignments. However,

```
one, two = three, four, five;
```

is not a valid assignment.

Two exceptions exist for this rule: first, if the object to the right of the assignment operator is a one-tuple, it may always be "promoted" into the dimensionality of the object to the left of the assignment operator. For example:

```
a, b, c = d;
```

is equivalent to:

```
a = d; b = d; c = d;
```

The second exception is that if the object to the left of the assignment operator is a one-tuple, and the object to the right of the assignment operator is not a one-tuple, then the "array" operator is assumed. For instance, the assignments:

```
a = b, c, d;
a = array (b, c, d);
```

are equivalent. Note that the promotion facility and the implicit-array operator can never be used simultaneously. This feature disallows statements such as:

```
one, two = three, four, five;
```

which more probably represents a user error than a useful statement.

Assignments also have values. The value of an assignment is the value of the tuple into which the assignment is done. For example, the value of:

```
foo = bar;
```

is the item foo. This feature allows nested assignments, as in the following example:

```
pic = some_setpos, (line = vector 100);
```

which is equivalent to:

```
line = vector 100;
pic = some_setpos, line;
```

Note the use of the parentheses for precedence definition. The parentheses in the expression are necessary since tuple formation has precedence over assignment. If the expression had been written as:

```
pic = some_setpos, line = vector 100;
```

it would have been performed as the operations:

```
some_setpos, line = vector 100;          /* a promotion */
pic = some_setpos, line;              /* an implicit array */
```

Distinction Between Tuples and Arrays

It is worth stressing that tuples of elements do not represent arrays of elements. The user is cautioned against letting the convenience of the assumed array operator within assignments blur this distinction. For example, a common error is to assume implicit array operations when using macros. Macro creations are not true assignments, but definitions; therefore, they do not automatically assume the array operator. Given the macro "diamond" described above, the assignment:

```
big_diamond = diamond 300 300;
```

is equivalent to:

```
big_diamond = vec 300 300, vec -300 300, vec -300 -300, vec 300 -300;
```

which becomes (via the assumed array operation of the assignment):

```
big_diamond = array (vec 300 300, vec -300 300, vec -300 -300, vec 300 -300);
```

However, the use of a macro is a non-assignment context such as:

```
display diamond 300 300;
```

simply evaluates to:

```
display vec 300 300, vec -300 300, vec -300 -300, vec 300 -300;
```

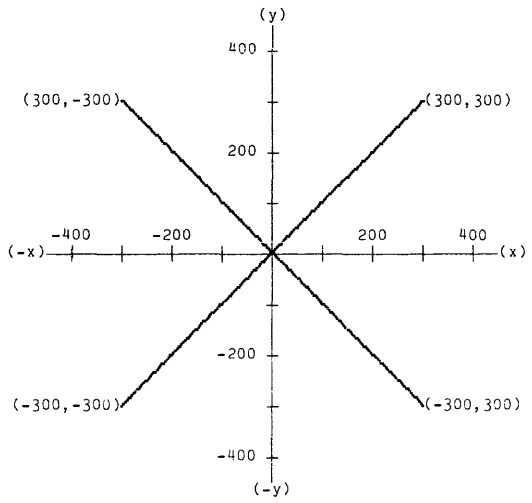
which is a request to display four separate objects, each originating (via the convention of the graphic system) at location (0,0,0). This expression will actually result in a cross being displayed in the center of the screen (see "Illustration B" below). The desired effect could have been obtained by the request:

```
display array (diamond 300 300);
```

or by using the value of a temporary assignment such as:

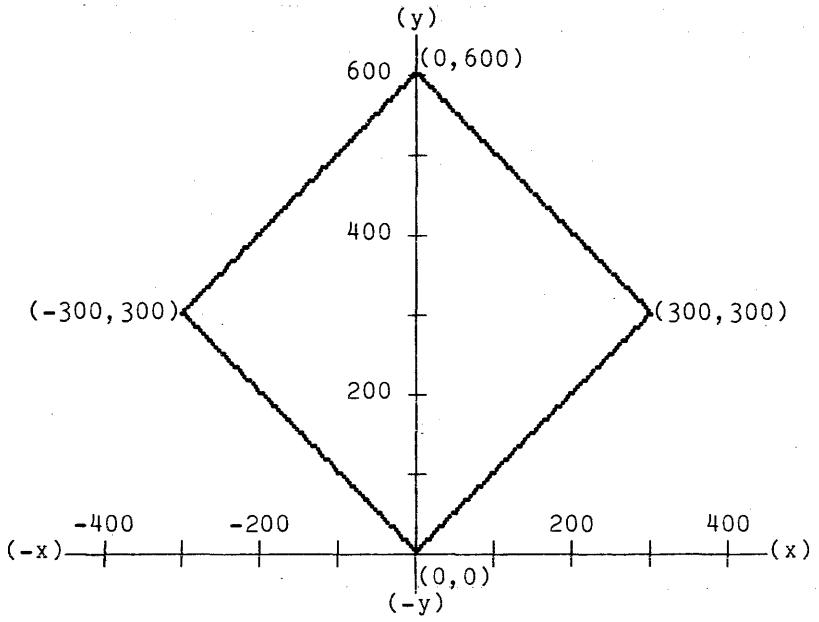
```
display big_diamond = diamond 300 300;  (see "Illustration C" below)
```

Illustration B



graphics_editor

Illustration C



Qualified Expressions

It is possible to refer to any element (or tuple of elements) of a symbol that represents an array or list by the use of a qualified expression. The simplest qualified expression consists of a symbol, followed by a period. This represents "the value of." In the first example:

```
foo = bar;
```

bar is assigned as the value of foo. The relationship of foo to bar is a superior/inferior, or father/son relationship. If instead the user types:

```
foo = bar.;
```

then the value of bar is assigned to foo. This makes both foo and bar refer to the identical piece of graphic structure. The symbols now have a "brother" relationship.

Successive trailing periods denote further levels of evaluation. Assume the following assignments:

```
box = vec 10, vec 0 10, vec -10, vec 0 -10;
a = b = c = d = box;
```

The following relations hold on these symbols: (Read "=" as "is identical to")

```
a. = b
a.. = b. = c
a... = b.. = c. = d
a.... = b... = c.. = d. = box
```

The assignment:

```
a... = null;
```

actually assigns null to d. (This example is for illustrative purposes only, and is not a typical use of qualified expressions.)

Additional types of qualified expressions make it possible to refer to elements of lists. The element desired is denoted by an integer following the appropriate levels of qualification. For example:

```
box.2
```

is the second element of box (vector 0 10). Tuples of contiguous elements may be specified by using a range expression, which consists of two integers (representing the first and last element desired), separated by a colon (":"). For example,

```
bottomless_box = array (box.2:4);
```

creates a symbol that contains an array made up of all elements of box except the first.

The star ("*") has a special meaning in a qualified expression. If used by itself, e.g., "box.*", it refers to a tuple made up of all the elements of "box". It may also be used as the last part of a range expression, e.g., "box.2:*", which refers to a tuple made up of all the elements of "box" from the second to the last. The assignment:

```
bottomless_box = array (box.2:*);
```

is equivalent to the previous example. Note that if a star occurs in a qualified expression, it must be the last character. It may neither be followed by the second component of a range expression (e.g., "box.*:3") nor by further levels of qualification (e.g., "box.*.1").

Because a user may not always know exactly how many levels of symbol indirection exist between the symbol name he is working with and the arrays or lists with which he desires to work, any reference to an element (or range of elements) of a list found in a qualified expression will cause the evaluator to skip any number of levels of symbol indirection. Using one of the previous examples, this means that:

```
a.1 = a.....1 = box.1
```

This frees the user from typing long, and possibly inaccurate, strings of periods, but it does allow the user who wants to maintain fine control of his indirect symbol structuring to do precisely that.

Certain qualified expressions may have different meanings on the left side of an assignment than they do on the right side. This is particularly important to note when using nested assignments. In particular, qualified expressions that evaluate to an element of an array or list, or to a tuple of such elements, have different meanings in these two contexts. If such an expression occurs on the right side of an assignment, its value consists of references to the values of the elements that make up the list. A previous example ("bottomless_box") showed how this usage is interpreted. On the left side of the assignment, however, the expression denotes element replacement. For instance, assume the following assignments:

```
box = vec 10, vec 0 10, vec -10, vec 0 -10;
elem = box.3;
box.3 = shift -10;
```

The first assignment defines "box". The second assignment causes "elem" to refer to the same piece of graphic structure which is the third element of box. The third assignment changes the "top of the box" from a visible vector to an invisible shift by redefining the third element of "box" to be a shift of equal magnitude. This does not change the value of "elem". It simply breaks the association between the list "box" and the construct which was its third element. If the actual changing of that construct were desired, the third assignment of the above example could be replaced with:

```
box.3. = shift -10;
```

This assignment would in fact change the value of "elem". A side-effect of this property is that the expressions "symbol.n" and "symbol.n." are equivalent on the right side of an assignment, but are not equivalent on the left side.

Node Constants

It is possible for node values to exist in the WGS without being assigned to any symbol. For instance, a user program could be called from inside the editor to construct a particularly intricate "canned" graphic structure which may be inefficient or difficult to construct by hand. The program could print the number of the top-level node in the structure, so that the user could reference it by assigning a name to it. The number of this node may be typed in, preceded by the character "#".

For example: if the node constant "#12345" appears as such an output, and a user wishes to assign to this node the name "orphan", the assignment:

```
orphan = #12345;
```

may be used.

Octal node values may be expressed directly as node constants without user conversion by immediately following the "#" with the lowercase letter "o", e.g., "#o144" is equivalent to "#100".

Although node constants and qualified expressions based on node constants are allowed on the left-hand side of assignment statements, their use is strongly discouraged.

Commands

Following is a list of editor commands shown in alphabetical order. Arguments enclosed in braces {} denote optional arguments. Each command whose argument is signified by <exprn> accepts single elements, tuples, assignments, or any combination of these as its argument. For example:

display, di <exprn>
: erases the screen and displays the specified graphic structure. If the argument is a tuple, no erase is performed between each element of the tuple. The tuple is not collected into an implicit array, but is displayed one element at a time. This means that the current graphic position reverts to (0,0,0) before displaying each new element.

display pic = array (house, street, parked_cars);
serves the dual purpose of defining "pic" and displaying it.

execute, exec command_line
passes the <command_line> to the command processor.

get {mode} {(pathname)} sym1 {sym2} ... {symn}
gets the structures (or macros) named "sym1 ... {symn}" from the PGS specified by (pathname). (This notation means that "pathname", if it is given, must be within parentheses.) The mode argument determines what action is taken on attempts to redefine an existing name:

-force
redefines the symbol and all subsidiary symbols.

-replace_all, -rpa
redefines the symbol. If subsidiary symbols are duplicated in the WGS, uses the copies in the WGS. For any subsidiary symbols that do not exist in the WGS, uses the ones in the PGS.

-replace_only, -rpo
redefines the symbol. If subsidiary symbols are duplicated in the WGS, uses the copies in the WGS. For any subsidiary symbols not so duplicated, creates null (empty) symbols.

-safe
leaves the old symbol as is and prints an error message.

If mode is not specified, "-safe" is assumed. The mode and (pathname) arguments, if present, may occur in either order, but must precede any symbol names. If pathname is not given, the last pathname specified for a "put" or "get" operation is used.

help, ?
directs the user to relevant documentation.

input (device_name) sym1 ... symn
requests that one or more "what" inputs be requested from device (device_name). The inputs are collected, interpreted, made into graphic structures, and assigned to symbols "sym1 ... symn". The (device_name) may be any input device name found in the include file "graphic_enames.incl.pl1" (see Section 3). The device names currently recognized are:

- any
- joystick
- keyboard
- lightpen
- mouse
- tablet

- terminal_program
- trackball

list, ls {starname1 ... starnameN} {-control_args}
lists selected symbol tables. Any number of control arguments may be specified. The following control arguments are allowed:

-all, -a
lists all of the following.

-commands, -com
lists the editor commands and their abbreviations.

-macros, -mc
lists the defined macros.

-symbols, -sym
lists the user symbols.

-system, -sys
lists the available system symbols, system macros, and their abbreviations.

If no control arguments are given, -symbols and -macros are assumed, with the listing of macros suppressed if there are none.

Each starnamei may be a simple item name, or may be a properly-formed name according to the rules of the Multics star convention facility (see description of "Star Names" in the MPM Reference Guide). If any starnamei arguments are given, only items which match one or more of the given starnames are listed. If no starnamei arguments are given, "**" is assumed.

macro name {arg1} ... {argn} = <exprn>
macro show name1 ... {namen}
macro replay name1 ... {namen}
The first form defines a macro with name "name", and arguments {arg1} ... {argn}. The other two forms expand the macro command (see "replay" and "show" commands later in this description).

put {mode} {(pathname)} sym1 {sym2} ... {symn}
stores the structures (or macros) named sym1 {sym2} ... {symn} into the PGS specified by (pathname). The mode argument determines what action is taken on attempts to redefines an existing name:

-force
redefines the symbol and all subsidiary symbols.

-replace_all, -rpa
redefines the symbol. If subsidiary symbols are duplicated in the PGS, uses the copies in the PGS. For any subsidiary symbols that do not exist in the PGS, uses the ones in the WGS.

-replace_only, -rpo
redefines the symbol. If subsidiary symbols are duplicated in the PGS, uses the copies in the PGS. For any subsidiary symbols not so duplicated, creates null (empty) symbols.

-safe
leaves the old symbol as is and prints an error message.

If mode is not specified, -safe is assumed. The mode and pathname arguments, if present, may occur in either order, but must precede any symbol names.

quit, q
exits from the editor.

read pathname
interprets the file specified by pathname as a set of editor commands. If the suffix ".ge" is not explicitly provided in pathname, it is assumed. Any read command encountered in a file switches the input source to the specified file. When the commands in the specified file have been exhausted, control returns to the user's terminal, or to the original file issuing the "read." Errors encountered while reading from a segment cause control to be immediately returned to the user's terminal.

remove sym1 {sym2} ... {symn}
removes those elements named from the table of known user symbols. The symbol in the WGS is also deleted, and all references to it are transformed into direct references to whatever contents it possessed.

replay <exprn>
replay -all, -a
prints an abbreviated description of the tuple <exprn> on the user's terminal. If the value represents a terminal graphic element, its contents are printed, or if it represents a nonterminal element, it is described and the number of its elements given, except that the entire graphic subtree inferior to the chosen node is described in assignment notation along with nested assignments where appropriate. This command replays a graphic structure in a form acceptable as input to the graphics_editor. If the control argument "-all" is given, all user symbols are replayed. If this control argument is present, it must be the only argument.

restart
reinitializes the editor, the WGS, and all associated symbol tables. Any remaining command line, as well as any file "reads" pending, are flushed without execution. The state of the editor after a "restart" is the same as the state of the editor when it is first invoked.

save {pathname}
saves the contents of the WGS in a PGS specified by pathname. If pathname is not supplied, graphics_editor uses the pathname that was last supplied to a use or save command. If no such pathname exists, an error occurs. If an error occurs during the execution of a save command, the "last pathname" is deliberately forgotten.

show <exprn>
prints an abbreviated description of the tuple <exprn> on the user's terminal. If the value represents a terminal graphic element, its contents are printed. If it represents a nonterminal element, it is described and the number of its elements given.

use {pathname}
loads the PGS specified by pathname into the WGS. This allows the editor to use a previously-constructed set of graphic structures. If pathname is not supplied, graphics_editor uses the pathname that was last supplied to a use or save command. If no such pathname exists, an error occurs. If an error occurs during the execution of a use command, the "last pathname" is deliberately forgotten.

vtype {pathname}
makes the graphic character table specified by pathname the default character table used by subsequent varying_text elements.

. (period)
identifies the graphics_editor by name and version.

Defined System Symbols

POSITIONAL ELEMENTS

All positional elements take arguments of the form "x y z". If any of these arguments are not supplied, they are assumed to be zero. It is possible to supply no arguments, only "x", only "x y", or all of "x y z". No other combinations (e.g., "x z") are parsable.

point, pnt
setpoint, spt
setposition, sps
shift, sft
vector, vec

MODAL ELEMENTS

In general, nonnumeric values for mode settings may be any string that is used in the include file "graphic_enames.incl.pl1" to describe that particular mode. Values shown here represent the strings currently recognized.

blink, blk

Argument: Integer, 0 or 1, where:

0 steady
1 blinking

color

Argument: (up to three) must be of the form:

<color_spec> {<intensity>}

The <color_spec> may be "red", "blue", or "green". Intensities may be any integer from 0 to 63. If a <color_spec> is not followed by an intensity, 63 is assumed.

intensity, int

Argument: Integer, 0 through 7, where:

0 off
7 on
7 full

linetype, lin

Argument: Integer, 0 through 4, where:

0 solid
1 dashed
2 dotted
3 dashed-dotted
4 long-dashed

sensitivity, sns

Argument: Integer, 0 or 1, where:

0 insensitive
1 sensitive

MAPPING ELEMENTS

rotation, rot

Argument: "x_rotation y_rotation z_rotation" in floating or integer degrees. If any of these arguments are not supplied, they are assumed to be zero.

scaling, scl

Argument: "x_scale y_scale z_scale" in integer or floating notation. If any of these arguments are not supplied, they are assumed to be one.

MISCELLANEOUS ELEMENTS

null

No arguments. This element represents the "zero node." It is a placeholder, or a graphic no-op.

text "string" {position}, "string" {position}

The abbreviated form of the text string is implicitly understood. If the long it contains no spaces or other characters used as item separators. The string may not exceed 200 characters. The optional position argument specifies the string alignment. (For a more complete explanation of string alignments, refer to Section 3.) Any character may appear within the string. If it is desired for a quote to appear as part of the string, it may be doubled, as in PL/I. The argument may be either a string or an integer, from the following correspondence list:

integer	string
1	upper_left, ul
2	upper_center, uc
3	upper_right, ur
4	left, l
5	center, c
6	right, r
7	lower_left, ll
8	lower_center, lc
9	lower_right, lr

* datablock, data <element>

creates a datablock containing the element <element>. This element may be of a form acceptable as a symbol name, numeric, or a string enclosed in quotes. It may not be, or contain, a break character (";", ",", etc.) unless enclosed in quotes. The string may not exceed 200 characters. Datablocks may be used to hold information relevant to the structure, within the structure itself. The format of the data within a datablock created by the graphic editor is arranged strictly for the convenience of the graphic editor. User programs should not be dependent on this format. (For a more complete explanation of datablocks, refer to Section 3.)

Defined System Macros

arc x_dist y_dist fraction

creates an arc starting from the current graphic position, with center at (x_dist, y_dist) from the current position. The fraction specifies what fraction of a circle (1 = entire circle) this arc is to represent.

box x_len y_len

creates a box whose first vector is (x_len, 0), whose second vector is (0, y_len), etc. from the current position.

circle x_dist {y_dist}

creates a circle starting from and ending at the current graphic position, with center at (x_dist, y_dist) from the current position.

ellipse x_dist y_dist eccentricity axis_angle {fraction}

creates an ellipse with epicenter (geographical center) at (x_dist, y_dist) from the current position, with the given eccentricity (long_axis/short_axis), and with the long axis at axis_angle from the x axis. If fraction is specified, that fraction of an ellipse is drawn.

polygon x_dist y_dist n_sides

works like circle. n_sides specifies how many sides the polygon is to have.

varying_text "string" {position {width {height}} {character_set}}
vtext "string" {position {width {height}} {character_set}}

creates a representation of the specified character string from vectors and shifts. (Refer to Section 5 and the graphic_chars_ subroutine.) The position argument is as described above for text. The string may not exceed 200 characters. The arguments width and height specify the average width and height of each character of the resultant string. The character_set argument specifies the name or pathname of a graphic character table (GCT). (Refer to Section 3 and the documentation on graphic character tables.)

NOTE: No dynamic operations are presently defined for the graphics_editor.

Name: remove_graphics, rg

The remove_graphics command terminates a session of working with the graphics system. It detaches graphic switches that were set up by setup_graphics.

Usage

```
remove_graphics {switch_1} ... {switch_N}
remove_graphics -all, -a
remove_graphics
```

If given, the optional arguments "switch_1" to "switch_N" specify the switches to be detached. The second and third forms detach all graphic switches known to setup_graphics, including online switches.

Name: setup_graphics, sg

The setup_graphics command attaches and manipulates graphics I/O switches. Its simplest use is to inform the graphics system as to which type of graphic device is being used.

Usage

setup_graphics {-control_args}

where -control_args consists of some combination of the following control arguments:

-from, -fm switchname {mode_spec}
specifies the graphic switches to be attached. More than one set of -from control arguments may be given to the command. The optional mode_spec argument specifies one of the opening modes defined by the iox_ subroutine. If mode_spec arguments are not supplied, stream_input_output is assumed. If no -from options appear in an invocation of the command, the assumed default is: "-from graphic_output stream_output -from graphic_input stream_input".

-modes mode_string
specifies GDT or device modes to be applied (via iox_$modes) to the switches named in the -from control arg(s).

-offline
is equivalent to "-to offline_graphics_ stream_output".

-online
specifies that the user_i/o switch, as well as all the switches mentioned in -from options, is to be channeled through the graphic I/O module graphic_dim_ to the user's terminal. This is the normal mode of operation for online (i.e., terminal) devices.

-output_file path, -of path
specifies that graphic I/O is to be routed to a file instead of a graphic device. If the -table control argument is not specified, the MSGC produced by graphic operations is routed to the specified file without further translation. In this case, the suffix ".graphics" is added to path, if it is not already present. If the -table control argument is specified, the MSGC is translated into device-dependent form before being routed to the specified file. In this case, the name of the specified GDT is added as a suffix to path, if it is not already present (see "Notes" below).

-table, -tb gdt_name
causes the graphic device table named gdt_name to be associated with the graphic switches set up on this invocation of the command. This option must appear in the command line if the -output_file control argument is not specified.

-to switch_name
specifies the target switch name to which all the switches mentioned in -from options are to be attached, through graphic_dim_. If this option is not given in the command line, the assumed default is: "-to tty_i/o -online" (see "Notes" below).

setup_graphics

Notes

If the first argument to setup_graphics is not a control argument, it is assumed to be the name of a GDTS, as in the -table option.

A description of modes accepted by each GDT may be found in the descriptions of the GDTs in Section 6.

Use of the -output_file and -table control arguments to route device-dependent character codes into a file may not always produce usable results. Certain graphic devices possess dependencies which must be compensated for at runtime by the graphic I/O module. Examples of these are baud rate dependencies with respect to delays, tape record blocking for offline devices, and graphic pauses. No guarantee is made as to the repeatability or quality of the results achieved through the use of device-dependent files. Conversely, files containing device-independent MSGC produce correct results, not only on various devices of the same type, but on supported devices of other types as well.

SECTION 5

SUBROUTINES

This section contains descriptions of the Multics graphics subroutines, presented in alphabetic order. Each description contains the name of the subroutine, discusses the purpose of the subroutine, lists the entry points, and describes the correct usage for each entry point. Notes and examples are included when deemed necessary for clarity. (Refer to Appendix A for a list of subroutine and entry point abbreviations that are supported, but not documented in the body of the text.)

The following subroutines are described:

calcomp_compatible_subrs_, ccs_
: incorporates a set of graphic subroutines identical to those supplied by California Computer Products (CalComp) Inc.

gf_int_
: interprets and prints MSGC on a nongraphic terminal.

gr_print_
: interprets an ASCII character string and prints a text description of the graphic action represented.

graphic_chars_
: accepts a character string and creates a list of vectors to represent those characters.

graphic_code_util_
: contains a set of entries that encode into and decode from MSGC formats.

graphic_compiler_, gc_
: compiles a graphic structure into MSGC.

graphic_decompiler_
: constructs a graphic structure in the WGS from a string of MSGC.

graphic_dim_
: communicates with all graphic devices.

graphic_element_length_
: determines length of graphic effector in MSGC.

graphic_error_table_
: contains messages and error codes for the MGS.

graphic_macros_, gmc_
: provides the ability to create common graphic objects not directly representable as primitive graphic elements.

graphic_manipulator_, gm_
: provides facility for the creation, editing and permanent storage of graphic items.

*

graphic_operator_, go_
: contains entry points for animation, graphic input and user interaction, and graphics terminal control.

graphic_terminal_status_
: interprets error messages sent from a remote programmable graphic terminal.

gui_
: module to provide the casual user a means of performing simple graphics.

*

plot_
: user interface to create a two dimensional graph from input data.

Name: calcomp_compatible_subrs_, ccs_

This subroutine incorporates a set of graphic subroutines that have names, calling sequences, and argument conventions that are identical to those supplied on other computers by CalComp Incorporated. These routines use entries in the MGS to perform graphics operations, and are therefore compatible with any device supported by the MGS.

Since the names of these subroutines do not follow the system standard convention of having trailing underscores, the entry names are not added to calcomp_compatible_subrs_. The entries may be called in one of two ways:

1. using calcomp_compatible_subrs_ as an explicit segment name, e.g., "call calcomp_compatible_subrs_$factor ..."; (the preferred method for native Multics programs), or

2. linking to calcomp_compatible_subrs_ in the system library and adding the entry names as alternate names to the link (allows programs transferred from other systems to continue to work without editing).

The CalComp plotter software and the MGS differ on basic conventions such as screen (page) size, location of origin, etc. These differences and their resolutions are described under "Programming Considerations" at the end of this subroutine description.

Entry: calcomp_compatible_subrs_$axis

This entry causes one axis to be created, labeled, and delineated with tick marks and coordinate values. When operating in native Multics mode, (i.e., in points rather than inches), one tick mark is provided every 100 points. When simulating the page size of another system through the use of the set_dimension entry point, one tick_mark is provided every "inch."

Usage

```
declare calcomp_compatible_subrs_$axis entry (float bin, float bin,
     char(*), fixed bin, float bin, float bin, float bin, float bin);

call calcomp_compatible_subrs_$axis (x_position, y_position, title,
     control, axis_len, angle, first_value, delta_value);
```

where:

1. x_position (Input)
   is the distance in the x direction between the current origin and the desired end point of the axis.

2. y_position (Input)
   is the distance in the y direction between the current origin and the desired end point of the axis.

3. title (Input)
is the title to be placed along the axis.

4. control
is a general control argument that specifies:

- by its magnitude, the number of significant characters in title;
- by its sign, the side of the axis on which the title and coordinate values are to be placed. (A positive value places these items on the "clockwise" side of the axis; a negative value, on the "counterclockwise" side.)

5. axis_len (Input)
is the length of the axis desired.

6. angle (Input)
is the angle at which the string (or symbol) is to be plotted. An angle of zero plots the string in line with the x-axis, while an angle of 90 plots the string along the y-axis, with the tops of the characters in the -x direction. The title and coordinate values, as well as the axis line itself, are influenced by this angle.

7. first_value (Input)
is the value to be placed at the first tick mark on the axis.

8. delta_value (Input)
is the difference between successive tick marks.

Notes

The variables first_value and delta_value may be computed using the scale entry, or computed by the user.

Entry: calcomp_compatible_subrs_$dfact

This entry acts much like the factor entry point, except that it allows independent x and y scaling factors.

Usage

```
declare calcomp_compatible_subrs_$dfact entry (float bin, float bin);

call calcomp_compatible_subrs_$dfact (x_scaling, y_scaling);
```

where:

1. x_scaling (Input)
is a scale factor to be applied to the x component of all further picture elements.

2. y_scaling (Input)
is a scale factor to be applied to the y component of all further picture elements.

Notes

The scaling factors produced by the factor and the dfact entry points are not independent. A call to either entry destroys any scaling factors set up by any previous call to either entry. A byproduct of this fact is that the statement:

```
call calcomp_compatible_subrs_$factor (any_scale);
```

is exactly equivalent to the statement:

```
call calcomp_compatible_subrs_$dfact (any_scale, any_scale);
```

Entry: calcomp_compatible_subrs_$dwhr

This entry acts much like the where entry point, except that it includes extra arguments to return the separate scaling factors that may have been set by a call to the dfact entry point.

Usage

```
declare calcomp_compatible_subrs_$dwhr entry (float bin, float bin,
     float bin, float bin);

call calcomp_compatible_subrs_$dwhr (x_position, y_position, x_scaling,
     y_scaling);
```

where:

1. x_position (Output)
is the distance in the x direction between the pen and the user-defined origin.

2. y_position (Output)
is the distance in the y direction between the pen and the user-defined origin.

3. x_scaling (Output)
is the presently active scaling factor in the x direction, as set by a call to the factor or the dfact entry points.

4. y_scaling (Output)
is the presently active scaling in the y direction, as set by a call to the factor or the dfact entry points.

Notes

The scaling returned by this entry point and the dwhr entry point does not reflect the effects of the scale factor (if any) set by calls to the set_dimension entry point.

If this entry point is called while two independent x and y scaling fact entry pointers are active (as set by the calcomp_compatible_subrs_$dfact entry point), scaling is returned as the greater of the two factors, and an error message is printed.

Entry: calcomp_compatible_subrs_$factor

This entry sets a scaling factor that applies to all graphic work following the call. This factor prevails until the entry point is called again with a different factor. A call to this entry point does not alter the values of graphic elements created with a previous scaling factor.

Usage

```
declare calcomp_compatible_subrs_$factor entry (float bin);

call calcomp_compatible_subrs_$factor (scaling);
```

where scaling (Input) is a scaling factor to be applied to all subsequent graphic elements.

Notes

The scaling factors produced by the factor and the dfact entry points are not independent. A call to either entry destroys any scaling factors set up by any previous call to either entry. A byproduct of this fact is that the statement:

```
call calcomp_compatible_subrs_$factor (any_scale);
```

is exactly equivalent to the statement:

```
call calcomp_compatible_subrs_$dfact (any_scale, any_scale);
```

Entry: calcomp_compatible_subrs_$line

This entry produces a line plot given two arrays of data points. A symbol may be plotted at each point.

Usage

```
declare calcomp_compatible_subrs_$line entry (float bin dimension(*),
     float bin dimension(*), fixed bin, fixed bin, fixed bin, fixed bin);

call calcomp_compatible_subrs_$line (x_array, y_array, n_points,
     step_size, line_type, symbol_no);
```

where:

1. x_array (Input)
   is an array of independent variables to be plotted along the x-axis. It must contain the values "first_value" and "delta_value" as the two trailing elements of the array, as produced by the entry point scale, below.

2. y_array (Input)
   is an array of dependent variables to be plotted along the y-axis. It must also contain the values "first_value" and "delta_value".

3. n_points (Input)
   is the number of significant data points in each array. It does not include "first_value" and "delta_value".

4. step_size (Input)
   indicates which elements of the array are to be scanned, according to the following rules (where "n" is any positive quantity):

   +n every nth element is to be scanned, starting with the first. The first_value is to be a minimum, and delta_value is to be positive.

   -n every nth element is to be scanned, starting with the first. The first_value is to be returned as a maximum, and delta_value is to be negative.

5. line_type (Input)
   specifies the type of plot to be produced:

   - if zero, only lines connect the points.
   - if positive, symbols are plotted every (line_type) points, connected by lines.
   - if negative, only symbols are plotted every (-line_type) points.

6. symbol_no (Input)
   is the number of a special symbol to be plotted.

Notes

Use of a symbol_no that is not defined results in an error message being printed and the character "*" being used instead.

Refer to Table 5-1 located at the end of this subroutine description for a list of Multics CalComp symbols.

Entry: calcomp_compatible_subrs_$newpen

This entry implements color changes.

Usage

```
declare calcomp_compatible_subrs_$newpen entry (fixed bin);

call calcomp_compatible_subrs_$newpen (color);
```

where color (Input) is defined as:

```
1   blue
2   green
3   red
```

Notes

In plotting applications, it is assumed that pen #1 is a blue pen, pen #2 is a green pen, and pen #3 is a red pen.

Entry: calcomp_compatible_subrs_$number

This entry plots a floating number according to a user-supplied format indicator.

Usage

```
declare calcomp_compatible_subrs_$number entry (float bin, float bin,
     float bin, float bin);

call calcomp_compatible_subrs_$number (x_position, y_position, height,
     float_num, angle, precision);
```

where:

1. x_position (Input)
   is the desired distance in the x direction between the current origin and the lower left-hand corner of the character or string (see "Notes" below).

2. y_position (Input)
   is the desired distance in the y direction between the current origin and the lower left-hand corner of the character or string (see "Notes" below).

3. height (Input)
   is the desired height of the character. A character (i.e., one character including the space between it and the next) is square (see "Notes" below).

4. float_num (Input)
   is the number to be drawn.

5. angle (Input)
   is the angle at which the string (or symbol) is to be plotted. An angle of zero plots the string in line with the x-axis, while an angle of 90 plots the string along the y-axis, with the tops of the characters in the -x direction.

6. precision (Input)
   controls the precision of the drawn number according to the following rules:

   - if precision > 0, then (precision) digits are displayed to the right of the decimal point.
   - if precision = 0, only the integer portion and the decimal point are displayed.
   - if precision = -1, only the integer portion is displayed, with no decimal point.
   - if precision < -1, then <abs (precision)-1> digits are removed from the rightmost portion of the integer part before displaying.

Notes

The magnitude (absolute value) of precision should not exceed 9.

All values are rounded to the precision given, not truncated. For example, the number "8765.432" in conjunction with precision "-2" produces 877.

Entry: calcomp_compatible_subrs_$offset

This entry specifies the parameters of an alternate two-dimensional coordinate system, into which selected vectors are translated before plotting. (See the description of the "indicator" argument to the entry point plot, below.)

Usage

```
declare calcomp_compatible_subrs_$offset entry (float bin, float bin,
     float bin, float bin);

call calcomp_compatible_subrs_$offset (x_zero, x_factor, y_zero,
     y_factor);
```

where:

1. x_zero (Input)
   is the x value of the virtual origin of the alternate coordinate system.

2. x_factor (Input)
   is the scale factor by which to divide the x component of vectors in the alternate coordinate system.

3. y_zero (Input)
   is the y value of the virtual origin of the alternate coordinate system.

4. y_factor (Input)
   is the scale factor by which to divide the y component of vectors in the alternate coordinate system.

Notes

The relationship of the translated x and y components to the x and y components given as arguments to the plot entry point are:

```
translated_x = (given_x - x_zero) / x_factor * x_scaling;
translated_y = (given_y - y_zero) / y_factor * y_scaling;
```

where x_scaling and y_scaling are the (possibly identical) scaling factors that may have been set by calls to either the fact or the dfact entry points. (See the description of these entry points for an explanation of these arguments.)

Note that x_factor and y_factor perform scaling by a division operation rather than by multiplication, as opposed to the multiplicative scaling factor set by the factor and the dfact entry points.

If the user changes the origin of the primary coordinate system through a call to the plot entry point with a negative indicator argument, the origin of the alternate coordinate system changes by the same amount.

Entry: calcomp_compatible_subrs_$plot

This entry draws vectors and shifts. It may redefine the origin (0, 0) of the picture.

Usage

```
declare calcomp_compatible_subrs_$plot entry (float bin, float bin,
     fixed bin);

call calcomp_compatible_subrs_$plot (x_position, y_position, indicator);
```

where:

1. x_position (Input)
   is the desired absolute positioning of the pen from the origin in the x direction.

2. y_position (Input)
   is the desired absolute positioning of the pen from the origin in the y direction.

3. indicator (Input)
   is a generalized switch controlling all other parameters of the line:

   If indicator is negative, the action is performed as described below as if indicator were positive. After the action is performed, the origin is redefined to be the current (final) position.

   If indicator is positive, the action is performed as follows with no redefinition of the origin:

2	a visible vector is drawn
3	an invisible shift is drawn
12	a visible vector is drawn subject to the offset and scaling given in the last call to the offset entry point
13	an invisible shift is drawn, subject to the offset and scaling given in the last call to the offset entry point
22	same as 2
23	same as 3
> 30	same as -3. In addition, all undisplayed graphic work to this point is displayed. Further graphic work may not be added to this picture.

Notes

Unlike the CalComp packages, indicator > 30 does not "close" the graphic device. However, it "closes" the picture, in that it is displayed to the device, and then destroyed. Further graphic work results in a new picture.

Calls to this entry point do not advance "search records," as this construct is without counterpart in the MGS.

Entry: calcomp_compatible_subrs_$plots

This entry initializes calcomp_compatible_subrs_. All static information (e.g., current scale, current pen-position) with the exception of the virtual screen size set by the entry point set_dimension (see below) is destroyed and reinitialized to default values. If this entry is called more than once in a process, it destroys all undisplayed graphic work slated for the current output device since the previous invocation of plots. (See the description of the plot entry point above for an explanation of displayed versus undisplayed graphics.)

Usage

```
declare calcomp_compatible_subrs_$plots entry;

call calcomp_compatible_subrs_$plots;
```

No arguments are necessary. Any arguments supplied are ignored.

Entry: calcomp_compatible_subrs_$scale

This entry enables the user to place on the plotting package the burden of scaling data arrays to be plotted. A call to this entry point causes the calcomp_compatible_subrs_ to compute an initial value, "first_value", and an increment, "delta_value" (which is in units of given_data_units/(100 points) in native Multics mode, or in given_data_units/inches when simulating the page size of another system through the use of the set_dimension entry point.) These values are stored at the end of the array of data values supplied. They are more fully described in the description of the axis entry, above.

Usage

```
declare calcomp_compatible_subrs_$scale (float bin dimension(*), float bin,
     fixed bin, fixed bin);

call calcomp_compatible_subrs_$scale (data_array, axis_len, n_points,
     step_size);
```

where:

1. data_array (Input/Output)
   is the array of data points to be examined.

2. axis_len (Input)
   is the length of the axis along which this array is to be plotted.

3. n_points (Input)
   is the number of useful input values in data_array.

4. step_size (Input)
indicates which elements of the array are scanned, according to the following rules (where "n" is any positive quantity):

+n every nth element is scanned, starting with the first. The first_value is to be a minimum, and delta_value is to be positive.

-n every nth element is scanned, starting with the first. The first_value is to be returned as a maximum, and delta_value is to be negative.

Notes

The subscripted locations of the returned values, first_value and delta_value are:

```
first_value = data_array (n_points * step_size) + 1;
delta_value = data_array ((n_points + 1) * step_size) + 1;
```

The user must ensure that enough unused elements are left at the end of data_array in which to store the returned information.

Entry: calcomp_compatible_subrs_$set_dimension

This entry enables the user who has transferred working programs from other systems to define the screen (page) dimensions of the graphic device so as to appear to have the same dimensions as the device previously used.

Usage

```
declare calcomp_compatible_subrs_$set_dimension entry (float bin);

call calcomp_compatible_subrs_$set_dimension (size);
```

where size (Input) is the greater of the two dimensions of the previously used graphic device.

Notes

This entry should only be called immediately before a call to the plots entry point. A call to this entry at any other time (i.e., during picture construction) produces undefined results.

Since every graphic device attached to the Multics system is defined to have a square (or cubical) working area, the user's plot requests are scaled so that the longest dimension of the previous device corresponds to 1024 Multics points.

The size, set by this entry point, remains in effect for the duration of the process. It is not destroyed or reset by calls to any other entry point, including plots. This datum may be reset by another call to the set_dimension entry point. It may be reset to the default value by specifying size = 1024.

Values for scaling that are returned by entries such as the where, dwhr, wofst, etc. entry points are unaffected by any setting of apparent screen size performed by this entry.

Entry: calcomp_compatible_subrs_$symbol

This entry displays strings of alphanumerics and specially-defined symbols.

Usage

```
declare calcomp_compatible_subrs_$symbol entry options (variable);

call calcomp_compatible_subrs_$symbol (x_position, y_position,
     height, string, angle, string_len);
```

or

```
call calcomp_compatible_subrs_$symbol (x_position, y_position, height,
     symbol_no, angle, symbol_ctl);
```

where:

1. x_position (Input)
   is the desired distance in the x direction between the current origin and the lower left-hand corner of the character or string (see "Notes" below).

2. y_position (Input)
   is the desired distance in the y direction between the current origin and the lower left-hand corner of the character or string (see "Notes" below).

3. height (Input)
   is the desired height of the character. A character (i.e., one character including the space between it and the next) is square (see "Notes" below).

4. string (Input)
   is the character string to be drawn.

5. symbol_no (Input)
   is the number of a special symbol to be plotted.

6. angle (Input)
is the angle at which the string (or symbol) is plotted. An angle of zero plots the string in line with the x-axis, while an angle of 90 plots the string along the y-axis, with the tops of the characters in the -x direction.

7. string_len (Input)
is the length of the string (in characters) supplied in "string". This number must be positive, to signify that a string, and not a symbol number, has been supplied.

8. symbol_ctl (Input)
is zero or negative, to signify that a symbol number, and not a character string, has been supplied.

0 shift position to coordinates given and plot the symbol
-1 same as 0
-2 draw visible line to coordinates given and plot the symbol

Notes

If either x_position or y_position (or both) = 999, it is left unchanged from the pen's current position, i.e., where the pen was last left.

If height = 999, the last height given is used. Note that the height saved from calls to either symbol or number is used.

The current pen position after drawing a symbol, as determined by a call to the where entry point, is defined to be the point at which the symbol was placed (i.e., (x_position, y_position)). Separate calls to concatenate symbols using the "999" feature return the pen to the same position after completing the new symbol. This is true no matter how many symbols are concatenated using this feature.

Use of a symbol_no that is not defined results in an error message being printed and the character "*" being used instead.

Refer to Table 5-1 located at the end of this subroutine description for a list of Multics CalComp symbols.

Entry: calcomp_compatible_subrs_$where

This entry may be used to determine the present position (relative to the user-defined origin) of the pen, and to determine the present applicable scaling factor.

Usage

```
declare calcomp_compatible_subrs_$where entry (float bin, float bin,
     float bin);

call calcomp_compatible_subrs_$where (x_position, y_position, scaling);
```

where:

1. x_position (Output)
   is the distance in the x direction between the pen and the user-defined origin.

2. y_position (Output)
   is the distance in the y direction between the pen and the user-defined origin.

3. scaling (Output)
   is the present applicable scaling factor, as set by a call to the factor entry point.

Notes

The scaling returned by this entry and the dwhr entry point do not reflect the effects of the scale factor (if any) set by calls to the set_dimension entry point.

If this entry point is called while two independent x and y scaling factors are active (as set by the dfact entry point), scaling is returned as the greater of the two factors, and an error message is printed.

Entry: calcomp_compatible_subrs_$wofst

This entry retrieves the arguments given in the last call to the offset entry point.

Usage

```
declare calcomp_compatible_subrs_$wofst entry (float bin, float bin,
     float bin, float bin);

call calcomp_compatible_subrs_$wofst (x_zero, x_factor, y_zero, y_factor);
```

where:

1. x_zero (Input)
   is the x value of the virtual origin of the alternate coordinate system.

2. x_factor (Input)
is the scale factor by which to divide the x component of vectors in the alternate coordinate system.

3. y_zero (Input)
is the y value of the virtual origin of the alternate coordinate system.

4. y_factor (Input)
is the scale factor by which to divide the y component f vectors in the alternate coordinate system.

Programming Considerations

CalComp routines on other systems express coordinates and lengths in inches. The MGS defines all graphic devices to have screens (or page sizes) of 1024x1024 (x1024) points. The relationship between inches and points is different for different devices. To enable the user of calcomp_compatible_subrs_ to use all the space available, the subroutine initially accepts coordinates in points. Put another way, the graphic device seems to have an area of 1024x1024 inches. (These "virtual inches" almost invariably are much smaller than the real thing.) The user who has programs transferred from another system may use one call to the set_dimension entry to adjust the size of the plot.

The coordinate origin (0,0) is initially defined as the point (-512,-512,0) of the Multics virtual graphics terminal screen. The user may change the location of this origin with the use of any of several entries.

The plots entry point performs no attachments to graphic devices. The attachment of a graphic device must be done beforehand with the setup_graphics command. All picture requests are processed and stored internally by the MGS. When a call to the plot entry point is encountered with indicator > 30, the picture is considered completed. It is then transmitted to the graphic device. No further picture requests are honored until the plots entry point is again called. A call to plots at this time causes an "erase" (page advance, on a plotter) to be generated, and processing continues.

Special symbols are located in a permanent graphic segment named "calcomp_special_symbols_.pgs", which is located by the use of the search rules. Symbols are named "calcomp_symbol_n", where n is the integer corresponding to the symbol in Table 5-1 below.

The display list created by ccs_ has the name "ccs_display_list_".

Since no provisions for returning error codes are possible, calcomp_compatible_subrs_ calls com_err_ with a suitable explanation if at any time an error condition is detected.

calcomp_compatible_subrs_

Table 5-1. Multics CalComp Symbols

0	□	20	⌟	40	Ш	60	:	80	E	100	Y	120	s		
1	⏲	21	‡	41	⋋	61	,	81	F	101	Z	121	t		
2	⏢	22	‡	42	∝	62	$	82	G	102	a	122	u		
3	+	23	Δ	43	δ	63	?	83	H	103	b	123	v		
4	×	24	⎓	44	∈	64	&	84	I	104	c	124	w		
5	◇	25	∧	45	η	65	(	85	J	105	d	125	x		
6	⏚	26	≠	46	▫	66	)	86	K	106	e	126	y		
7	⨯	27	±	47	\	67	+	87	L	107	f	127	z		
8	⨯	28	Σ	48	÷	68	-	88	M	108	g	128	0		
9	Y	29	V	49	⊛	69	/	89	N	109	h	129	1		
10	⌘	30	~	50	¢	70	*	90	O	110	i	130	2		
11	✳	31	≈	51	¬	71	=	91	P	111	j	131	3		
12	⧖	32	}	52	𝄞	72	.	92	Q	112	k	132	4		
13	I	33	{	53	←	73	%	93	R	113	l	133	5		
14	✡	34	Ш	54	↑	74	_	94	S	114	m	134	6		
15	☰	35	Π	55	↓	75	!	95	T	115	n	135	7		
16	|	36	ϕ	56	ϒ	76	A	96	U	116	o	136	8		
17	→	37	⊖	57	⋄⋄	77	B	97	V	117	p	137	9		
18	_	38	Y	58	∞	78	C	98	W	118	q	*138			
19	≤	39	X	59	;	79	D	99	X	119	r				

*138 is the symbol for "blank".

Name: gf_int_

This I/O module interprets MSGC and prints it in a form suitable for inspection at a terminal that lacks graphic capabilities. It uses the subroutine gr_print_ to interpret and print out the graphic elements by name and value.

Usage

```
io attach graphic_output gf_int_ user_output

io open graphic_output stream_output
```

The opening mode must be "stream_output", as in the example. The target switch must be "user_output", as gr_print_ simply calls ioa_ to print its data.

Name: gr_print_

This subroutine interprets an ASCII character string assumed to contain Multics graphics characters. For each sequence of graphics characters in the string, an "English" text description of the graphics action represented by that sequence is printed out.

Usage

```
declare gr_print_ entry (char(*));

call gr_print_ (string)
```

where string (Input) is the ASCII character string to be interpreted as MSGC.

Name: graphic_chars_

This subroutine accepts a character string and creates a list of vectors that represent those characters. Unlike the characters that compose a graphic character string, these vector-composed characters may be scaled and rotated.

Entries exist to allow graphic_chars_ to make use of any number of special fonts and styles contained in graphic character tables (GCTs). Section 7 contains a list of supported GCTs. Section 3 contains information about creating new GCTs.

Declarations for all the user-callable entries in graphic_chars_ are contained in the PL/I include file "gch_entry_dcls.incl.pl1" (see Section 8). Users may include this file (using the PL/I "%include" facility) in their source programs to save typing and syntax errors.

FORTRAN programmers should check "Programming Considerations" in Section 2 for special instructions about entries that return fixed bin quantities.

Usage

```
declare graphic_chars_ entry (char(*), fixed bin, float bin, float bin,
     fixed bin(35)) returns (fixed bin(18));

node = graphic_chars_ (string, alignment, x_size, y_size, code);
```

where:

1. node (Output)
   is a node value representing a list of vectors that represent the character string.

2. string (Input)
   is a character string that is to be simulated by a list of vectors.

3. alignment (Input)
   indicates which portion of the character string is to be displayed at the current screen position. The following values are used:

   1 top left
   2 top center
   3 top right
   4 middle left
   5 dead center
   6 middle right
   7 bottom left
   8 bottom center
   9 bottom right

4. x_size (Input)
   is the desired dimension, in points, of the width of a character (including the space between it and the next character).

5. y_size (Input)
is the desired dimension, in points, of the height of a character.

6. code (Output)
is a standard status code.

Entry: graphic_chars_$init

This entry initializes graphic_chars_. Programs using graphic_chars_ must call this entry whenever they perform an operation that destroys the current contents of the working graphic segment, such as calling graphic_manipulator_$init or graphic_manipulator_$use_file (see "General Notes" at the end of this subroutine description.) This entry does not affect the setting of the current graphic character table as specified in previous calls to graphic_chars_$set_table.

Usage

```
declare graphic_chars_$init entry;

call graphic_chars_$init;
```

there are no arguments.

Entry: graphic_chars_$set_table

This entry selects a graphic character table other than the default character table.

Usage

```
declare graphic_chars_$set_table entry (char(*), char(*), fixed bin(35));

call graphic_chars_$set_table (dirname, ename, code);
```

where:

1. dirname (Input)
is the directory portion of the pathname of the desired graphic character table. If dirname is the null string, the graphic character table specified by ename is located via the graphics search list (see "Notes" below).

2. ename (Input)
is the entryname portion of the pathname of the desired graphic character table.

3. code (Output)
is a standard status code.

Notes

The graphics search list is described fully in Section 2.

Entry: graphic_chars_$get_table

This entry returns the name of the graphic character table currently in use by graphic_chars_.

Usage

```
declare graphic_chars_$get_table entry (char(*), char(*));

call graphic_chars_$get_table (dirname, ename);
```

where:

1. dirname (Output)
is the directory portion of the pathname of the graphic character table currently in use.

2. ename (Output)
is the entryname portion of the pathname of the graphic character table currently in use.

Entry: graphic_chars_$long

This entry functions as the main entry, described at the beginning of this subroutine. In addition, it returns values describing the coordinate differences between the start of the character string and the end of the character string.

Usage

```
declare graphic_chars_$long entry (char(*), fixed bin, float bin, float
     bin, float bin, float bin, fixed bin(35)) returns (fixed bin(18));

node = graphic_chars_$long (string, alignment, x_size, y_size, x_spread
     y_spread, code);
```

where:

1. node (Output)
   is a node value representing a list of vectors that represent the character string.

2. string (Input)
   is a character string that is simulated by a list of vectors.

3. alignment (Input)
   indicates which portion of the character string be displayed at the current screen position. The following values are used:

   1 top left
   2 top center
   3 top right
   4 middle left
   5 dead center
   6 middle right
   7 bottom left
   8 bottom center
   9 bottom right

4. x_size (Input)
   is the desired dimension, in points, of the width of a character (including the space between it and the next character).

5. y_size (Input)
   is the desired dimension, in points, of the height of a character.

6. x_spread (Output)
   is the x distance, in points, between the location of the start of the character string and the end of the character string.

7. y_spread (Output)
   is the y distance, in points, between the location of the start of the character string and the end of the character string.

8. code (Output)
   is a standard status code.

Notes

The array of vectors and shifts that is produced by graphic_chars_ is guaranteed to begin and end at the same point, thus ensuring a net relative shift of zero for any pseudo character-string produced by calling this subroutine. This entry is provided mainly for the use of programmers of applications packages who may find it necessary, for example, to append one string to the end of another. It is of little use to the average user.

Entry: graphic_chars_$long_tb

This entry is used exactly as graphic_chars_$long. Unlike the other entries, which strip trailing blanks from a character string before converting it, this entry transforms trailing blanks encountered into shifts.

Usage

```
declare graphic_chars_$long_tb entry (char(*), fixed bin, float bin,
     float bin, float bin, float bin, fixed bin(35)) returns
     (fixed bin(18));

node = graphic_chars_$long_tb (string, alignment, x_size, y_size, x_spread,
     y_spread, code);
```

All arguments are as described for graphic_chars_$long above.

General Notes

Use of any entry described here must be preceded at some point by a call to graphic_manipulator_$init, which creates a WGS. These entries assume such a segment exists, and attempt to create graphic structures in it.

For efficiency, once any character has been converted into vectors, graphic_chars_ remembers its node value. A subsequent encounter of that same character causes graphic_chars_ to use the same list, rather than reassemble the character from vectors. This "memory" must be cleared by a call to graphic_chars_$init whenever any operation destroys the contents of the WGS.

All entries to graphic_chars_, with the exception of long_tb, strip trailing blanks from strings before aligning and converting them.

Name: graphic_code_util_

This subroutine contains a set of entries that encode into and decode from parameter formats used in the MGS. For a description of the different formats in the MSGC, refer to Section 3.

Entry: graphic_code_util_$decode_dpi

This entry decodes DPI-format characters into an array of numbers.

Usage

```
declare graphic_code_util_$decode_dpi entry (pointer, fixed bin, (*)
     fixed bin);

call graphic_code_util_$decode_dpi (string_ptr, count, array);
```

where:

1. string_ptr (Input)
   points to the string from which the encoded values of the numbers are taken. The string need not be aligned.

2. count (Output)
   is the number of elements in the array, starting with the first to be converted.

3. array (Output)
   is an array of numbers, each of which has been converted from characters.

Entry: graphic_code_util_$decode_scl

This entry decodes SCL-format into an array of numbers.

Usage

```
declare graphic_code_util_$decode_scl entry (pointer, fixed bin, (*)
     float bin);

call graphic_code_util_$decode_scl (string_ptr, count, float_array);
```

where:

1. string_ptr (Input)
   points to the string from which the encoded values of the numbers are taken. The string need not be aligned.

2. count (Output)
   is the number of elements in the array, starting with the first to be converted.

3. float_array (Output)
   is an array of numbers, each of which has been converted from characters.

Entry: graphic_code_util_$decode_scl_nonzero

This entry decodes SCL-format into an array of numbers. This entry should be used when decoding scaling factors, since to preserve the integrity of rotation/scaling matrices, it always returns the value 1e-6 in place of a zero.

Usage

```
declare graphic_code_util_$decode_scl_nozero entry (pointer, fixed bin,
     (*) float bin);

call graphic_code_util_$decode_scl_nozero (string_ptr, count, float_array);
```

where:

1. string_ptr (Input)
   points to the string from which the encoded values of the numbers are taken. The string need not be aligned.

2. count (Output)
   is the number of elements in the array, starting with the first to be converted.

3. float_array (Output)
   is an array of numbers, each of which has been converted from characters.

Entry: graphic_code_util_$decode_spi

This entry decodes SPI-format characters into an array of numbers.

Usage

```
declare graphic_code_util_$decode_spi entry (pointer, fixed bin, (*)
     fixed bin);

call graphic_code_util_$decode_spi (string_ptr, count, array);
```

where:

1. string_ptr (Input)
   points to the string from which the encoded values of the numbers are taken. The string need not be aligned.

2. count (Output)
   is the number of elements in the array, starting with the first to be converted.

3. array (Output)
   is an array of numbers, each of which has been converted from characters.

Entry: graphic_code_util_$decode_uid

This entry decodes UID-format characters into an array of numbers.

Usage

```
declare graphic_code_util_$decode_uid entry (pointer, fixed bin, (*)
     fixed bin);

call graphic_code_util_$decode_uid (string_ptr, count, array);
```

where:

1. string_ptr (Input)
   points to the string from which the encoded values of the numbers are taken. The string need not be aligned.

2. count (Output)
   is the number of elements in the array, starting with the first to be converted.

3. array (Output)
   is an array of numbers, each of which has been converted from characters.

Entry: graphic_code_util_$encode_dpi

This entry encodes an array of numbers into DPI-format characters.

Usage

```
declare graphic_code_util_$encode_dpi entry ((*) fixed bin, fixed bin,
     pointer);

call graphic_code_util_$encode_dpi (array, count, string_ptr);
```

where:

1. array (Input)
   is an array of numbers, each of which is to be converted to DPI-format.

2. count (Input)
   is the number of elements in the array, starting with the first to be converted.

3. string_ptr (Input)
   points to the string in which the encoded values of the numbers are to be returned. The string need not be aligned.

Entry: graphic_code_util_$encode_scl

This entry encodes an array of numbers into SCL-format characters.

Usage

```
declare graphic_code_util_$encode_scl entry ((*) float bin, fixed bin,
     pointer);

call graphic_code_util_$encode_scl (float_array, count, string_ptr);
```

where:

1. float_array (Input)
   is an array of numbers, each of which is to be converted to SCL-format.

2. count (Input)
   is the number of elements in the array, starting with the first to be converted.

3. string_ptr (Input)
   points to the string in which the encoded values of the numbers are to be returned. The string need not be aligned.

Entry: graphic_code_util_$encode_spi

This entry encodes an array of numbers into SPI-format characters.

Usage

```
declare graphic_code_util_$encode_spi entry ((*) fixed bin, fixed bin,
     pointer);

call graphic_code_util_$encode_spi (array, count, string_ptr);
```

where:

1. array (Input)
   is an array of numbers, each of which is to be converted to SPI-format.

2. count (Input)
   is the number of elements in the array, starting with the first, to be converted.

3. string_ptr (Input)
   points to the string in which the encoded values of the numbers are to be returned. The string need not be aligned.

Entry: graphic_code_util_$encode_uid

This entry encodes an array of numbers into UID-format characters.

Usage

```
declare graphic_code_util_$encode_uid entry ((*) fixed bin, fixed bin,
     pointer);

call graphic_code_util_$encode_uid (array, count, string_ptr);
```

where:

1. array (Input)
   is an array of numbers, each of which is to be converted to UID-format.

2. count (Input)
   is the number of elements in the array, starting with the first to be converted.

3. string_ptr (Input)
   points to the string in which the encoded values of the numbers are to be returned. The string need not be aligned.

Name: graphic_compiler_, gc_

This subroutine contains entry points that compile a graphic structure resident in the WGS into MSGC. (See Section 3 for detailed descriptions of both graphic structures and MSGC.)

Graphic structure compilation consists of a left-most tree walk of the structure starting at the topmost list node. MSGC is generated for each node encountered, and the entire character string representation of the structure is output over the I/O switch named graphic_output. (Entries exist that allow the user to perform graphic operations on I/O switches other than graphic_output.) There are several compilation entry points to allow a graphic structure to be indicated by node value or symbol name, to cause the terminal screen to be erased before displaying the structure, and to allow the structure to be loaded into the terminal memory, but not be immediately displayed.

Because of the multitude of entry points in graphic_compiler_, declarations for all the user-callable entry points are contained in the PL/I include file gc_entry_dcls.incl.pl1 (see Section 3). Users may include this file (using the PL/I "%include" facility) in their source programs to save typing and syntax errors. In addition, because user programs normally call many entry points repeatedly, many entry points are given two names: a descriptive long name, and an abbreviation, both of which may be referenced externally.

GENERIC ARGUMENTS

The entry points of this subroutine (described below) do not include individual argument descriptions with each "entry" description. Reference is implied back to this paragraph.

1. node (fixed bin(18)) (Input)
   is the node value in the WGS of the topmost node of the graphic structure to be compiled.

2. name (char(*)) (Input)
   is a name in the WGS of the topmost node of the graphic structure to be compiled.

3. code (fixed bin(35)) (Output)
   is a standard status code.

GENERIC ENTRIES

Several of the entries in graphic_compiler_ have counterparts that perform the same operations as other entry points, except that they perform output over a user-specified I/O switch. Each of these entries has the same calling sequence as its counterpart, but takes one additional argument in the last argument position:

switch_ptr (Input)
is a pointer to the I/O switch on which the output is desired. If this is null, switch "graphic_output" is assumed.

The "variable switch" entry points are named similarly to their counterparts, with the suffix "_switch". They are listed below.

ENTRY	VARIABLE SWITCH COUNTERPART
display	display_switch
display_append	display_append_switch
display_name	display_name_switch
display_name_append	display_name_append_switch
load	load_switch
load_name	load_name_switch

Entry: graphic_compiler_$display

This entry erases the terminal screen, compiles the indicated graphic structure, loads it into terminal memory, and displays it.

Usage

```
declare graphic_compiler_$display entry (fixed bin(18), fixed bin(35));

call graphic_compiler_$display (node, code);
```

Entry: graphic_compiler_$display_append

This entry operates exactly as display, except that the terminal screen is not erased. This allows several independent structures to be superimposed.

Usage

```
declare graphic_compiler_$display_append entry (fixed bin(18),
     fixed bin(35));

call graphic_compiler_$display_append (node, code);
```

Entry: graphic_compiler_$display_name

This entry operates exactly as display, except that the topmost node is indicated by symbol name rather than by node value.

Usage

```
declare graphic_compiler_$display_name entry (char(*), fixed bin(35));

call graphic_compiler_$display_name (name, code);
```

Entry: graphic_compiler_$display_name_append

This entry operates exactly as display_append, except that the topmost node is indicated by symbol name rather than by node value.

Usage

```
declare graphic_compiler_$display_name_append entry (char(*),
     fixed bin(35));

call graphic_compiler_$display_name_append (name, code);
```

Entry: graphic_compiler_$load

This entry compiles a graphic structure and loads it into terminal memory. It does not erase the screen or display the structure.

Usage

```
declare graphic_compiler_$load entry (fixed bin(18), fixed bin(35));

call graphic_compiler_$load (node, code);
```

Notes

The loaded structure may be displayed at a later time by a call to graphic_operator_$display.

The concept of loading but not displaying a graphic structure in a terminal without its own memory makes no sense. Use of this entry point on such a terminal is equivalent to the use of the display_append entry point.

Entry: graphic_compiler_$load_name

This entry operates exactly like load, except that the topmost node is indicated by symbol name rather than by node value.

Usage

```
declare graphic_compiler_$load_name entry (char(*), fixed bin(35));

call graphic_compiler_$load_name (name, code);
```

Entry: graphic_compiler_$expand_string

This entry accepts a string of MSGC representing a graphic structure, transforms the structure into a graphic array, and returns the resultant MSGC.

Usage

```
declare graphic_compiler_$expand_string entry (char(*), fixed bin(21),
     pointer, fixed bin(21), fixed bin(35));

call graphic_compiler_$expand_string (input_string, chars_used, output_ptr,
     chars_output, code);
```

where:

1. input_string (Input)
   is a valid string of MSGC to be transformed into a graphic array.

2. chars_used (Output)
   is the number of characters scanned from input_string until the end of the graphic structure was detected.

3. output_ptr (Input)
   points to the string in which the resultant MSGC array is to be returned. It is the user's responsibility to ensure that the storage provided is adequate to hold the entire output string.

4. chars_output (Output)
   is the number of characters returned in the output string.

5. code (Output)
   is a standard status code.

Diagnostic Information

Various errors may occur during graphic structure compilation. For this reason, the location in the graphic structure of the current node being compiled is maintained in static storage during compilation. When an error occurs, users may obtain this information to locate and fix the inconsistencies.

Entry: graphic_compiler_$error_path

Usage

```
declare graphic_compiler_$error_path entry (fixed bin(18), fixed bin,
     fixed bin dimension(*), fixed bin(35));

call graphic_compiler_$error_path (top_node, depth, path_array, code);
```

where:

1. top_node (Output)
   is the node value of the top-level node of the graphic structure being compiled at the time of the error.

2. depth (Output)
   is the number of structure levels in the path to the substructure or element in which the error was discovered.

3. path_array (Output)
   is an array of list indexes comprising a unique path through the structure to the substructure or element in which the error was discovered.

4. code (Output)
   is a standard status code.

Notes

If path_array is too small to hold the entire path, the error code error_table_$smallarg is returned. In this case, depth contains the size of the array needed to hold the entire tree path.

Upon the occurrence of an error, graphic structure compilation is aborted, and no characters are output.

Name: graphic_decompiler_

This subroutine accepts, as input, a string of MSGC and constructs an isomorphic structure in the WGS from it. Node values encountered in the input string are mapped one-to-one into different, but equivalent, node values in the WGS.

FORTRAN programmers should check "Programming Considerations" in Section 2 for special instructions about entries that return fixed bin quantities.

Usage

```
declare graphic_compiler_entry (char(*), fixed bin(35)) returns
     (fixed bin(18));

node_no = graphic_decompiler_ (string, code);
```

where:

1. node_no (Output)
   is the node value in the WGS of the structure represented by the input string. If code is nonzero, this argument is -1.

2. string (Input)
   is a string of MSGC.

3. code (Output)
   is a standard status code.

Notes

This procedure does not initialize the WGS. Therefore, any call to it must have been preceded, at some time in the process, by a call to graphic_manipulator_$init.

Name: graphic_dim_

This I/O module is used to communicate with all graphic devices. Its main purpose is to translate the device-independent MSGC into device-dependent code with equivalent meaning, and dispatch this code to a graphic device. It also can intercept all the output of a process in order to ensure that the terminal is in the correct mode (graphic-accepting mode or text-accepting mode) to display the output. It has the responsibility of polling intelligent terminals to determine their status.

This module is not directly called by the user, but is referenced by the Multics I/O system whenever appropriate I/O system calls are made by the user or by other MGS subroutines (Further information on the I/O system may be found in the MPM Reference Guide.) The following writeup describes how the graphic_dim_ module responds to certain I/O system calls.

Notes

This I/O module interfaces to all graphics terminals supported by the MGS in a graphics capability. It is able to make many different device-dependent translations by virtue of relying on different GDTs and GSPs, each written for a different type of terminal. The syntax and description of both GDTs and GSPs may be found in the compile_gdt command described in Section 4.

The term "dynamic terminal" as used here refers to a programmable intelligent graphics terminal that is capable of performing some or all of the dynamic effectors provided for by the MSGC. Although it is possible to have a dynamically refreshed unintelligent terminal, this type of terminal is referred to as "static", and is grouped with storage-tube terminals.

Any attempt to send an incomplete graphic structure over a graphic output switch is in error. Any call to transmit a graphic structure must supply the I/O module with a complete graphic structure.

Permitted I/O System Calls

The following I/O system calls are implemented by this module:

attach
close
control
detach
get_chars
get_line
modes
open
put_chars

Opening Modes

The following attachment mode may be specified in calls to attach:

graphic
: indicates that all data coming from this switch is to be treated as MSGC. If this attribute is not on, the I/O module treats the data as normal text.

Control Requests

The following control requests are implemented:

set_table
: causes a GDT to be associated with a switch. The switch must be graphic. The data pointer in the order call points to a string declared as "char(32)" which is the name of the GDT to be used. It is located by the use of the search rules.

get_sdb
: sets the pointer argument of the order call to point to the switch datablock for this switch.

device_info
: causes information about the graphic device to be returned. The data pointer should be supplied pointing to the following structure which is filled in by the call:

```
dcl 1 device_info aligned,
      2 gdt_name char(32) aligned,
      2 gdt_ptr pointer,
/* device_data is taken from graphic_device_table.incl.pl1 */
      2 device_data aligned,
        3 version_number fixed bin,
        3 terminal_name char(32) aligned,
        3 terminal_type char(4) aligned,
        3 charsizes(3) float bin,
        3 message_size fixed bin(35),
        3 points_per_inch float bin(63),
        3 pad(10) fixed bin;
```

where:

1. gdt_name
   is the name of the GDT as supplied in the "set_table" control request.

2. gdt_ptr
   is the pointer to the segdef named "table_start" within the GDT.

3. version_number
   is the version number of the GDT.

4. terminal_name
   is the name of the terminal as given in the "Name:" statement of the GDT.

5. terminal_type
   is "stat" for a static terminal and "dyna" for a dynamic terminal.

6. charsizes
   are (in order) the height, width, and spacing of the character set provided by the machine. (These are measured in points.)

7. message_size
   is the size of the maximum message which may be sent to a terminal before requesting status. If the terminal can accept entire messages, this number will be the maximum number of characters in a segment.

8. points_per_inch
   is the number of virtual points per physical inch of device screen.

9. pad
   is unused space.

debug
: prevents the graphic I/O module from going into raw output and input mode when attempting to perform graphic I/O.

nodebug
: resets the effect of "debug", above.

Any control order not recognized by the graphic_dim_ is passed downstream.

Control Operations From Command Level

All control orders can be performed using the io_call command. The general format is:

```
io_call control switch_name order {optional_args}
```

where:

1. order
   is any of the control orders supported by graphic_dim_. If the control order is not recognized by graphic_dim_, it is passed to the next I/O module in the line of attachment.

2. optional_args
   are as required for various orders as indicated in the descriptions of the orders.

Status Codes

The following status codes may be returned by graphic_dim_:

error_table_$badopt
error_table_$invalid_mode
error_table_$long_record
error_table_$negative_nelem *
error_table_$noarg
error_table_$not_attached
error_table_$not_detached
error_table_$undefined_order_request
error_table_$unimplemented_version
graphic_error_table_$gdt_missing
graphic_error_table_$impossible_effector_length
graphic_error_table_$incomplete_structure
graphic_error_table_$invalid_node_no
graphic_error_table_$node_list_overflow
graphic_error_table_$node_not_active
graphic_error_table_$nongraphic_switch
graphic_error_table_$not_a_gdt
graphic_error_table_$term_bad_err_msg *
graphic_error_table_$too_many_node_ends
graphic_error_table_$unimplemented_effector
graphic_error_table_$unrecognized_effector

Any graphic error message initiated by an intelligent graphics terminal is reflected through this I/O module. In addition, the I/O module reflects any error status from other I/O modules that appear later.

graphic_element_length_

Name: graphic_element_length_

This subroutine determines the length, in characters, of any given graphic effector in MSGC format.

FORTRAN programmers should check "Programming Considerations" in Section 2 for special instructions about entries that return fixed bin quantities.

Usage

```
declare graphic_element_length_ entry (char(*), fixed bin(24)) returns (fixed
     bin);

len = graphic_element_length_ (string, index);
```

where:

1. len (Output)
   is the length, in characters, of the given effector.

2. string (Input)
   is a string of MSGC containing the effector to be inspected.

3. index (Input)
   is the index, within the string, of the beginning of the desired effector.

Notes

In the case of graphic effectors that "include" as arguments other graphic effectors (e.g., symbol effectors or increment effectors), only the length of the effector itself and any arguments up to the included subeffector (which is always the last argument, if it occurs) are returned.

Name: graphic_error_table_

This is an error table used in conjunction with the MGS. It is used in the same way as error_table_ (see MPM Reference Guide, Section 7) and contains those messages and error codes applicable to the graphics system.

Messages and Error Codes

The following pages contain a list of the long messages in alphabetical order. Each long message is followed by the name of the definition corresponding to it, (e.g., definition "struc_duplication", the short form of the message ("strucdup") shown at the extreme right margin on the same line with the definition), and an explanation of the error, which may include a description of possible corrective action.

The table contains two types of error messages. One type is returned by the part of the graphics system that is resident in Multics. The other is returned by intelligent terminals wishing to report an error condition. These latter errors may be distinguished by the fact that their definition names all begin with the characters "term_", and that their messages specifically mention that the terminal is returning this error (short form messages begin with the character "T").

A name duplication has occurred in moving a graphic structure.
(struc_duplication) strucdup

Meaning: While attempting to transfer a graphic structure between WGS and a PGS, the user tried to redefine a name which already existed. The user did not specify that this redefinition should be "forced," so the structure move was aborted.

A negative delay between increments has been specified.
(neg_delay) negdelay

Meaning: The user attempted to specify a negative delay time in an increment command.

An absolute effector appears in an array within the scope of an extent element.
(abs_pos_in_clipping) abs_clip

Meaning: The graphic compiler cannot process an array because it contains an absolute graphic effector (i.e., setposition or setpoint) that occurs within the influence of an extent element (i.e., clipping or masking) whose absolute extents are not known (i.e., no absolute graphic effectors occurred in the array prior to the occurrence of the active extent element). Such an array cannot be processed. This problem may be solved by either removing the absolute element and replacing with an equivalent relative element, or by inserting an absolute element at the beginning of the array.

Data is not a graphic structure.
(not_a_structure) notstruc

Meaning: The user attempted to write on an I/O switch which was attached as a graphic I/O switch, and the data written was not in MSGC. If this

message is returned from the graphic compiler or the graphic operator, it represents a graphics system malfunction.

Effector not implemented by this graphic device.
(unimplemented_effector) unimpeff

Meaning: The graphic I/O module detected the occurrence of some graphic command that the GDT lists as meaningless to the device being used. (Example: a query command to a plotter, or an increment command to a storage-tube device.) If using other than a system-supplied GDT, the GDT may be in error. If not, there is a high probability that the call causing the error condition is to graphic_operator_ . The user should examine the program to find occurrences of calls that may not be applicable to the particular device specified by the GDT.

Encountered effector has an impossible length.
(impossible_effector_length) badeffln

Meaning: The graphics system encountered inconsistent MSGC. If this message is reflected by the graphic compiler or graphic operator, then this occurrence should be reported as a bug.

Graphic clipping is not yet implemented.
(clipping_unimplemented) cantclip

Meaning: This error code is returned by an obsolete and no-longer-documented entry in graphic_manipulator_.

Graphic device table was not specified or is internally inconsistent.
(gdt_missing) no_gdt

Meaning: The graphic I/O module has not been supplied with the name of a GDT to associate with a particular graphic I/O switch. This error occurs with any attempt to write upon such an I/O switch. Or: A GDT has internal inconsistencies. The latter problem may be solved by a new process. If not, and a user-supplied GDT is being used, the GDT may have been damaged and may need to be recompiled.

Graphic input device number is not defined.
(bad_device_type) badevice

Meaning: The number supplied by the user in a request for graphic input to specify the device type from which the input was desired has not been assigned to any device by the MGS.

Internal graphic compiler error.
(compiler_error) comp_err

Meaning: The graphic compiler has experienced a software logic failure. This should be reported as a bug.

*

No working graphic segment exists.
(no_wgs_yet) no_wgs

Meaning: The user has attempted to use the graphics system without initializing the WGS. The procedure for doing so is explained in the description of graphic_manipulator_ provided later in this section.

Node is not a defined graphic datum.
(bad_node) bad_node

Meaning: A node value specified in a call to the graphics system refers to a node that has never been created (has invalid contents).

Not enough node ends encountered.
(incomplete_structure) <nodends

Meaning: This represents an inconsistency in produced MSGC. If it is reflected from a call to the graphic compiler or the graphic operator, it should be reported as a bug.

Segment is not a graphic device table.
(not_a_gdt) notagdt

Meaning: A segment that has been specified as a GDT for use with the graphic I/O module is not a GDT. Since search rules are used to find GDTs, it is possible that a segment of the same name precedes the real GDT in the search path. It is also possible for this message to be returned if a user-supplied GDT is damaged.

Supplied list index is outside the bounds of the list or array.
(list_oob) list_oob

Meaning: The user has requested an operation to be performed on the nth element of a list or array, where n is negative or the list or array does not contain n elements.

Symbol not found in symbol table.
(lsm_sym_search) nosymbol

Meaning: The symbol requested was not found. This may occur on calls to graphic_manipulator_ entries (find_struc, put_struc, or get_struc).

Terminal cannot increment requested node.
(term_bad_increment_node) Tnoincnd

Meaning: The terminal has refused a request to increment a node because the terminal program does not assign any meaning to the incrementing of the graphic data type represented by the node.

Terminal cannot locate requested node.
(term_node_not_found) Tno_node

Meaning: The graphics terminal cannot locate a node that was necessary to this operation. The node may have been previously deleted by the user.

Terminal does not implement requested input device.
(term_bad_input_device) Tbad_dev

Meaning: The user has requested a specific input device to be used in "what" input. The terminal either is not programmed to utilize this device or does not have such a device attached to it.

Terminal encountered an unimplemented graphic effector.
(term_bad_effector) Tbad_eff

Meaning: Similar to "unimplemented_effector" above. Warning: this error message may signify a discrepancy between the terminal's capabilities and the description in the GDT. It also may signify that software in the graphics terminal is in error.

Terminal encountered too many node ends.
(term_too_many_ends) T>nodend

Meaning: Incorrect MSGC was sent to the terminal. This message may occur if software in the graphics terminal is in error.

Terminal graphic buffer full.
(term_no_room) Tno_room

Meaning: There is no more room in the terminal for graphics structure. This message may occur if graphic structure is not deleted when it is no longer necessary. Certain entries in the graphic compiler automatically issue delete requests as noted in the graphic_compiler_ subroutine previously described in this section. (To explicitly delete structures in the graphics terminal memory, see the description of graphic_operator_ described later in this section.)

Terminal reported error in graphic message contents.
(term_bad_message) Tgarbage

Meaning: The graphics terminal encountered an error that did not express easily as a defined error condition. The problem may be a transmission problem. If this message persists, it should be reported as a bug.

Terminal reported parity error in graphic message.
(term_bad_parity) Txparity

Meaning: The terminal reported a bad parity occurrence. The error is almost certainly a hardware/transmission problem.

Terminal reported replacement node was too large.
(term_node_too_large) Tnodsize

Meaning: The terminal was requested to dynamically replace the contents of a node. The replacement node was too large to fit into the space originally allocated for the node. The structure should be resent with the new contents in place of the old.

Terminal reported stack depth overflow.
(term_too_many_levels) T>levels

Meaning: Structure depth has exceeded the number of levels supported by the graphics terminal. Note that this may be much smaller than the number of levels allowed by the Multics-resident portion of the system.

Terminal reported that no structure was active.
(term_no_active_structure) Tnotactv

Meaning: Some graphic operation was performed that incorrectly assumed a structure was being displayed by the terminal. (Example: a "which" query performed while the screen is blank of graphics.)

Terminal reported unimplemented effector in increment command.
(term_bad_increment_eff) Tnoincef

Meaning: The terminal does not implement the effector which the user is attempting to increment. If this error message occurs, it should be reported as a bug.

Terminal returned a garbled error message.
(term_bad_err_message) Tbaderr

Meaning: The terminal returned data which was not in status message format when the Multics-resident portion of the graphics system expected either an acknowledgement or an error message.

Terminal returned an invalid error code.
(term_bad_err_no) Tbaderr#

Meaning: The terminal returned an error code that was not assigned a meaning. This is most likely a transmission problem or a bug in the terminal's programming.

The alignment provided for a text node is undefined.
(bad_align) badalign

Meaning: An attempt was made to create a text effector with an undefined alignment code.

The directed operation is invalid.
(lsm_invalid_op) lsm_Xop

Meaning: lsm_ was requested to perform an unrecognized operation. This code should never be returned by the MGS.

The graphic effector type specified is invalid for this operation.
(inv_node_type) inv_type

Meaning: A graphics system entry was called that requires as an argument a node value representing some graphic type, but the node given was not of this type. Example: calling an entry to examine the contents of a list, but supplying the node value of a terminal element instead of a list.

The graphic input received was malformed.
(malformed_input) malformd

Meaning: A graphic device returned graphic input in an incorrect format. This may be a transmission problem. It also may signify that a GSP is receiving input from a device that it is not equipped to handle.

The graphic segment is full.
(lsm_full) lsm_full

Meaning: No more room exists in the graphic segment in question. Usually this means that the WGS is full, but it can also occur when putting structures into PGSs.

The internal node list table has overflowed.
(node_list_overflow) >nodetbl

Meaning: The table in which the graphic I/O module notes which nodes are resident in the remote processor is full. This limit is arbitrary and may be changed; therefore, occurrences of this message should be reported.

The node is not resident in the graphic processor.
(node_not_active) nodeinac

Meaning: An operation was requested to be performed upon a node assumed to be in the graphics terminal memory. The Multics-resident portion of the graphics system denies that this node is resident in the terminal memory.

The node value specified is not a valid node value.
(invalid_node_ob) badnode#

Meaning: A node value, passed as an input argument to a graphics system entry, does not correspond to any graphic element in the current WGS.

The node returned by the graphic terminal was not the node requested.
(mode_mismatch) wrongnod

Meaning: The terminal responded to a control request by controlling and/or returning the wrong node. This signifies problems with the internal terminal software.

The node value supplied is out of bounds.
(lsm_node_ob) lsm_oob

Meaning: The node value provided was negative, less than the smallest permissible node value, or greater than the greatest current node value. This error can occur when operations are attempted using node variables that have never been set.

The null node cannot be used to replace an existing node.
(null_replacement) replnull

Meaning: The user attempted to replace a node with the null node. The null (zero) node is not an actual node in that it possesses no contents. Rather, it is a list structure convention whose value (not contents) is significant. The user can obtain results similar to what is desired by replacing the node with (for example) a shift of zero.

The number of iterations to be performed is negative.
(bad_no_iter) bad_iter

Meaning: The user specified a negative number of iterations to take place within an increment command.

The specified graphic structure is recursive.
(recursive_structure) recursiv

Meaning: An attempt was made to compile a structure possessing an array that contained itself, possibly through some number of levels of indirection. Such a structure is not compilable. The entry graphic_compiler_$tree_ptr (described earlier in this section) can provide information on where the structure is in error.

This operation only permitted for a graphic I/O switch.
(nongraphic_switch) ^graphsw

Meaning: The graphic I/O module detected an attempt to perform a graphic operation (e.g., the association of a GDT) on a nongraphic I/O switch.

Too many elements supplied to create a single graphic list or array.
(lsm_blk_len) >blksize

Meaning: An attempt was made to create a list or array that exceeds the limitations of the graphics system (currently 4095 elements). The array or list should be split into a structure of smaller arrays or lists.

Too many node ends encountered.
(too_many_node_ends) >nodends

Meaning: Incorrect MSGC was passed to a graphics system entry. If this message is reflected by the graphic compiler or graphic operator, then this occurrence should be reported as a bug.

Unrecognized graphic effector encountered.
(unrecognized_effector) unreceff

Meaning: Incorrect MSGC was passed to a graphics system entry. If this message is reflected by the graphic compiler or graphic operator, then this occurrence should be reported as a bug.

Unrecognized special format character specified in graphic character table.
(gct_bad_special_char) gctspchr

Meaning: A graphic character table contains an explicit definition for a special format character (e.g., CR, NL, space, underscore). These format characters cannot be defined in a GCT as their representations are computed and constructed at run-time by the graphics system.

The following is a cross-index by definition name of graphic_error_table_ definitions listed above. The error message or "name of the definition" is given for each long message to assist in locating the information contained in the first section.

abs_pos_in_clipping
An absolute effector appears in an array within the scope of an extent element.
bad_align
The alignment provided for a text node is undefined.
bad_device_type
Graphic input device number is not defined.
bad_no_iter
The number of iterations to be performed is negative.
bad_node
Node is not a defined graphic datum.
clipping_unimplemented
Graphic clipping is not yet implemented.
compiler_error
Internal graphic compiler error.
gct_bad_special_char
Unrecognized special format character specified in graphic character table.
gdt_missing
Graphic device table was not specified or is internally inconsistent.
impossible_effector_length
Encountered effector has an impossible length.
incomplete_structure
Not enough node ends encountered.
inv_node_type
The graphic effector type specified is invalid for this operation.
invalid_node_no
The node number specified is not a valid node number.
list_oob
Supplied list index is outside the bounds of the list or array.
lsm_blk_len
Too many elements supplied to create a single graphic list or array.
lsm_invalid_op
The directed operation is invalid.

lsm_node_ob
The node value supplied is out of bounds.
lsm_seg_full
The graphic segment is full.
lsm_sym_search
Symbol not found in symbol table.
malformed_input
The graphic input received was malformed.
neg_delay
A negative delay between increments has been specified.
no_wgs_yet
No working graphic segment exists.
node_list_overflow
The internal node list table has overflowed.
node_mismatch
The node returned by the graphic terminal was not the node requested.
node_not_active
The node is not resident in the graphic processor.
nongraphic_switch
This operation only permitted for a graphic I/O switch.
not_a_gdt
Segment is not a graphic device table.
not_a_structure
Data is not a graphic structure.
null_replacement
The null node cannot be used to replace an existing node.
recursive_structure
The specified graphic structure is recursive.
struc_duplication
A name duplication has occurred in moving a graphic structure.
term_bad_effector
Terminal encountered an unimplemented graphic effector.
term_bad_err_message
Terminal returned a garbled error message.
term_bad_err_no
Terminal returned an invalid error code.
term_bad_increment_eff
Terminal reported unimplemented effector in increment command.
term_bad_increment_node
Terminal cannot increment requested node.
term_bad_input_device
Terminal does not implement requested input device
term_bad_message
Terminal reported error in graphic message contents.

graphic_error_table_

term_bad_parity
     Terminal reported parity error in graphic message.
term_no_active_structure
     Terminal reported that no structure was active.
term_no_room
     Terminal graphic buffer full.
term_node_not_found
     Terminal cannot locate requested node.
term_node_too_large
     Terminal reported replacement node was too large.
term_too_many_ends
     Terminal encountered too many node ends.
term_too_many_levels
     Terminal reported stack depth overflow.
too_many_node_ends
     Too many node ends encountered.
unimplemented_effector
     Effector not implemented by this graphic device.
unrecognized_effector
     Unrecognized graphic effector encountered.

graphic_gsp_utility_

Name: graphic_gsp_utility_

This module contains utility subroutines useful (but not limited) to Graphic Support Procedure (GSP) programmers. Entries in this program can perform common operations such as clipping displays to fit the screen.

Entry: graphic_gsp_utility_$clip_line

This entry performs two-dimensional clipping on positional elements. The caller specifies the starting and ending coordinates of the element and the clipping limits (e.g., the coordinates of the screen edges). Values are returned specifying the (possibly) clipped starting and ending coordinates, plus some output optimization information.

Usage

```
declare graphic_gsp_utility_$clip_line entry (float bin dimension (2),
     float bin dimension (2), float bin dimension (2,2), bit(1), bit(1),
     float bin dimension (2), bit(1), float bin dimension (2));

call graphic_gsp_utility_$clip_line (from, to, limits, shifting,
     goto_new_from, new_from, goto_new_to, new_to);
```

where:

1. from (Input)
   are the absolute coordinates of the starting point of the element.

2. to (Input)
   are the absolute coordinates of the final point of the element.

3. limits (Input)
   specify the desired clipping limits in the order: low X, low Y, high X, high Y.

4. shifting (Input)
   is "1"b if this element is invisible or nondrawing; otherwise, it is "0"b.

5. goto_new_from (Output)
   is "1"b if the caller must explicitly generate movement commands to the point designated by new_from; otherwise, it is "0"b. It is assumed that the argument from represents the caller's current graphic position.

6. new_from (Output)
   specifies the (possibly clipped) absolute coordinates of the desired starting point.

7. goto_new_to (Output)
   is "1"b if the caller must explicitly generate movement commands to the point designated by new_to; otherwise, it is "0"b.

8. new_to (Output)
specifies the (possibly clipped) absolute coordinates of the desired ending point.

Entry: graphic_gsp_utility_$clip_text

This entry performs two-dimensional clipping on text elements.

Usage

```
declare graphic_gsp_utility_$clip_text entry (char(*), fixed bin, float bin
     dimension (3), float bin dimension (2), float bin dimension (2,2),
     fixed bin, float bin dimension (2), fixed bin, fixed bin);

call graphic_gsp_utility_$clip_text (string, alignment, char_sizes, curpos,
     limits, hw_alignment, initial_shift, string_origin, string_len);
```

where:

1. string (Input)
   is the character string to be clipped.

2. alignment (Input)
   is the alignment of string (see "Notes" below).

3. char_sizes (Input)
   are the size parameters (in points) of a single character, in the order: height, width, spacing.

4. curpos (Input)
   is the absolute current graphic position.

5. limits (Input)
   specify the desired clipping limits in the order: low X, low Y, high X, high Y.

6. hw_alignment (Input)
   specifies the alignment by which the terminal normally aligns its hardware character set (see "Notes" below).

7. initial_shift (Output)
   specifies the relative shift that must be performed before displaying the (possibly clipped) string (see "Notes" below).

8. string_origin (Output)
   specifies the index of the first visible character of string.

9. string_len (Output)
   specifies the number of visible characters in string.

graphic_gsp_utility_

Notes

Values for alignment and hw_alignment are chosen from the list of allowable alignment values that appears in the description of the text element (see Section 3).

The caller must remain aware that the text element must not affect the current graphic position. In particular, it is the caller's responsibility to ensure this by compensating properly for both the initial shift given and the action of displaying the string, after which, the beam or pen position most likely will not coincide properly with the current graphic position.

Name: graphic_macros_, gmc_

This subroutine easily creates common graphic objects that are not directly representable as primitive graphic elements. All entities created are two-dimensional figures, at the position and in the orientation specified by the user. Each entry point returns a graphic node value that consists of an array of vectors.

Declarations for all the user-callable entry points in graphic_macros_ are contained in the PL/I include file "gmc_entry_dcls.incl.pl1" (see Section 8). Users may include this file (using the PL/I "%include" facility) in their source programs to save typing and syntax errors.

Each of the figures produced originates at the current graphic position. The current graphic position is left at the termination point of the figure. For simple closed curves (polygons, circles, ellipses, boxes) this is the same as the point of origin. For other figures (arcs, partial ellipses) it is not.

FORTRAN programmers should check "Programming Considerations" in Section 2 for special instructions about entries that return fixed bin quantities.

Entry: graphic_macros_$arc

This entry creates an arc using the same criteria used by the circle entry point.

Usage

```
declare graphic_macros_$arc entry (float bin, float bin, float bin,
     fixed bin(35)) returns (fixed bin(18));

node = graphic_macros_$arc (x_dist, y_dist, fraction, code);
```

where:

1. node (Output)
   is the returned graphic node.

2. x_dist (Input)
   is the x dimension of the relative distance from the current graphic position to the desired center of the circle.

3. y_dist (Input)
   is the y dimension of the relative distance from the current graphic position to the desired center of the circle.

4. fraction (Input)
   represents the fraction of a complete circle desired. If fraction = 1e0, a complete circle is drawn.

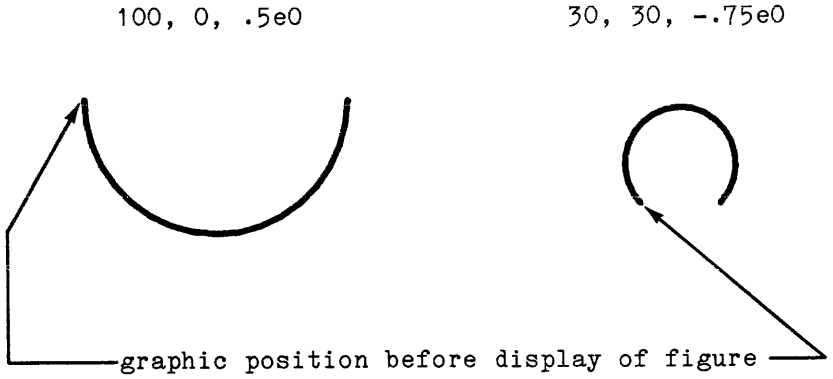
5. code (Output)
   is a standard status code.

Notes

Arcs are drawn counterclockwise, in the direction of increasing angle. If a clockwise arc is desired, a negative value for fraction may be used.

Examples

Values of x_dist, y_dist, and fraction:



Entry: graphic_macros_$box

This entry creates a rectangular box.

Usage

```
declare graphic_macros_$box entry (float bin, float bin, fixed bin (35))
     returns (fixed bin(18));

node = graphic_macros_$box (x_side, y_side, code);
```

where:

1. node (Output)
   is the returned graphic node.

2. x_side (Input)
   is the x dimension of the box desired.

3. y_side (Input)
   is the y dimension of the box desired.

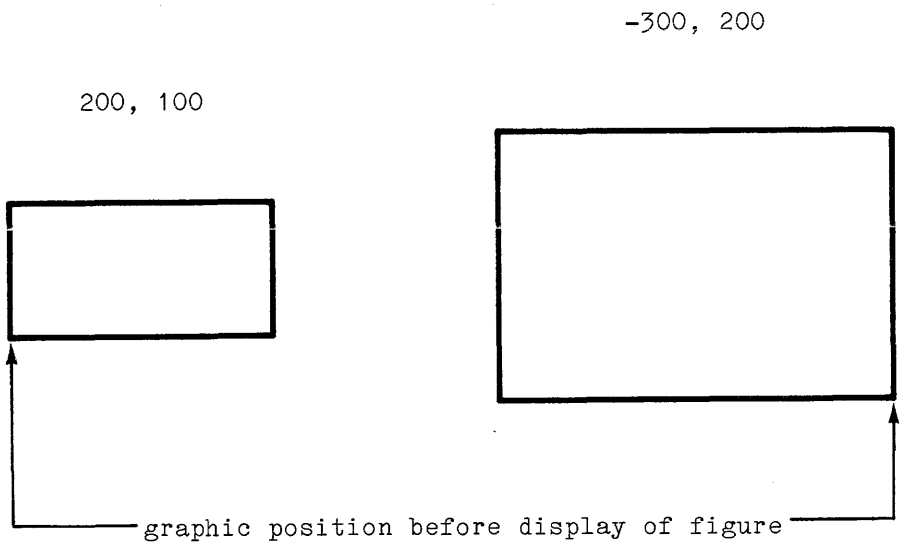
4. code (Output)
   is a standard status code.

Notes

The first two vectors of the box created are a horizontal line of length (x_side) and a vertical line of length (y_side). Therefore, if x_side and y_side are both negative, the box is drawn to the left and down from the current graphic position.

Examples

Values of x_side and y_side:



Entry: graphic_macros_$circle

This entry creates a circle. The rim of the circle originates at the current graphic position. The radius and orientation of the circle is determined by the given distances to the desired center point of the circle.

Usage

```
declare graphic_macros_$circle entry (float bin, float bin, fixed bin(35))
     returns (fixed bin(18));

node = graphic_macros_$circle (x_dist, y_dist, code);
```

where:

1. node (Output)
   is the returned graphic node.

2. x_dist (Input)
   is the x dimension of the relative distance from the current graphic position to the desired center of the circle.

3. y_dist (Input)
   is the y dimension of the relative distance from the current graphic position to the desired center of the circle.

4. code (Output)
   is a standard status code.

Entry: graphic_macros_$ellipse

This entry creates an ellipse, given the location of its epicenter (geographical center) and information about its eccentricity.

Usage

```
declare graphic_macros_$ellipse entry (float bin, float bin, float bin,
     fixed bin, float bin, fixed bin(35)) returns (fixed bin(18));

node = graphic_macros_$ellipse (x_dist, y_dist, eccentricity,
     eccentricity_angle, fraction, code);
```

where:

1. node (Output)
   is the returned graphic node.

2. x_dist (Input)
   is the x dimension of the distance from the current graphic position to the epicenter (geographical center) of the desired ellipse.

3. y_dist (Input)
   is the y dimension of the distance from the current graphic position to the epicenter (geographical center) of the desired ellipse.

4. eccentricity (Input)
   is the desired ratio of major axis to minor axis.

5. eccentricity_angle (Input)
   is the desired angle between the normal x-axis and the major axis of the ellipse.

6. fraction (Input)
   represents the fraction of the ellipse desired. If fraction = 1e0, an entire ellipse is drawn.

7. code (Output)
   is a standard status code.

Notes

Like arcs, fractional ellipses are drawn counterclockwise. If a clockwise portion of an ellipse is desired, a negative value for fraction may be used.

Fractional ellipses are computed on the basis of angle subtended by the elliptical portion, not by circumferential measurement. Therefore, depending on the location of the current graphic position and the angle of eccentricity, fractions such as 0.25e0 and 0.75e0 may not produce the expected result.

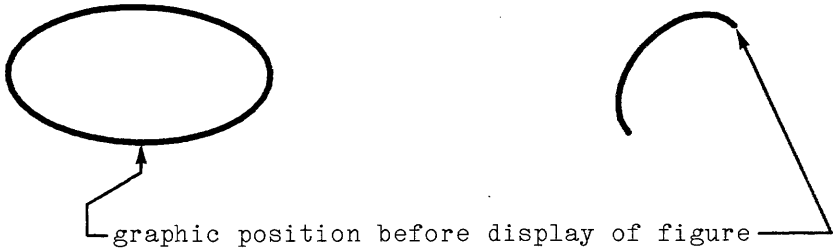
The definition of eccentricity presented does not bear any relation to the mathematical property also called eccentricity by which ellipses are sometimes described.

Examples

Values of x_dist, y_dist, eccentricity, eccentricity_angle, and fraction:

0, 50, 2, 0, 1

-40, -40, 1.5, 45, .5e0



Entry: graphic_macros_$ellipse_by_foci

This entry creates an ellipse given the locations of its two foci with respect to the current graphic position.

Usage

```
declare graphic_macros_$ellipse_by_foci entry (float bin, float bin,
     float bin, float bin, float bin, fixed bin(35)) returns (fixed bin(18));

node = graphic_macros_$ellipse_by_foci (x_dist1, y_dist1, x_dist2, y_dist2,
     fraction, code);
```

where:

1. node (Output)
   is the returned graphic node.

2. x_dist1 (Input)
   is the x dimension of the distance between the current graphic position and the first focus of the desired ellipse.

3. y_dist1 (Input)
   is the y dimension of the distance between the current graphic position and the first focus of the desired ellipse.

4. x_dist2 (Input)
   is the x dimension of the distance between the current graphic position and the second focus of the desired ellipse.

5. y_dist2 (Input)
   is the y dimension of the distance between the current graphic position and the second focus of the desired ellipse.

6. fraction (Input)
   represents the fraction of a complete ellipse desired. If fraction = 1e0, a complete ellipse is drawn.

7. code (Output)
   is a standard status code.

Notes

Like arcs, fractional ellipses are drawn counterclockwise. If a clockwise portion of an ellipse is desired, a negative value for fraction may be used.

Fractional ellipses are computed on the basis of angle subtended by the elliptical portion, not by circumferential measurement. Therefore, depending on the location of the current graphic position and the angle of eccentricity, fractions such as 0.25e0 and 0.75e0 may not produce the originally expected result.

The definition of eccentricity presented does not bear any relation to the mathematical property also called eccentricity by which ellipses are sometimes described.

Entry: graphic_macros_$polygon

This entry creates n-sided polygons.

Usage

```
declare graphic_macros_$polygon entry (float bin, float bin, fixed bin,
     fixed bin(35)) returns (fixed bin(18));

node = graphic_macros_$polygon (x_dist, y_dist, n_sides, code);
```

where:

1. node (Output)
   is the returned graphic node.

2. x_dist (Input)
   is the x dimension of the relative distance from the current graphic position to the desired center of the polygon.

3. y_dist (Input)
   is the y dimension of the relative distance from the current graphic position to the desired center of the polygon.

4. n_sides (Input)
   is the number of sides desired.

5. code (Output)
   is a standard status code.

Notes

One vertex of the polygon locates itself at the current graphic position.

Name: graphic_manipulator_, gm_

This subroutine contains entry points for creating, examining, and modifying graphic structures in the user's WGS in the process directory, and in one or more PGSs. (Refer to Section 3 for a complete discussion of graphic structures and structure manipulation.)

Because of the multitude of entry points in graphic_manipulator_, declarations for all the user-callable entry points are contained in the PL/I include file "gm_entry_dcls.incl.pl1" (see Section 8). Users may include this file (using the PL/I "%include" facility) in their source programs to save typing and syntax errors. In addition, because user programs normally call many entry points repeatedly, many entry points are given two names: a descriptive long name, and an abbreviation, both of which may be referenced externally.

Entries that create graphic elements usually take an integer argument to determine which type of element (e.g., setposition, vector, shift) is to be created. A PL/I include file, "graphic_etypes.incl.pl1", that declares mnemonic variables (e.g., Setposition, Vector, Shift) as representing these integers is available for inclusion in any source program, through the PL/I "%include" facility.

Entries that take arrays of dimension (*) as arguments can accept arrays of other than 1-origin (e.g., arrays may be declared "dimension (5:10)").

Unless otherwise stated, all entry points manipulate graphic structures resident in the current WGS.

FORTRAN programmers should check "Programming Considerations" in Section 2 for special instructions about entries that return fixed bin quantities.

Entry: graphic_manipulator_$init

This entry must be called before any other entry point in the graphics system. It initiates a WGS in the user's process directory.

Usage

```
declare graphic_manipulator_$init entry (fixed bin(35));

call graphic_manipulator_$init (code);
```

where code (Output) is a standard status code.

Notes

Subsequent calls to graphic_manipulator_$init reinitialize the WGS and destroy all previous structure.

Returned error codes may be from error_table_ (the system standard error table) or from graphic_error_table_ (errors particular to the graphics system). Both kinds of error codes may be sent directly to the com_err_ subroutine to obtain the corresponding error message.

Structure Creation Entry Points

The following entry points create the various terminal and nonterminal elements in a graphic structure. The newly created elements are free-standing in the current WGS, and are not incorporated in any structure. They may be incorporated in higher level structures through the create_list and create_array entry points (defined later in this description).

The graphic structure creation entry points return two generic arguments:

1. code (fixed bin(35))
   is a standard status code, either from error_table_ (the system standard error table) or from graphic_error_table_. The system subroutine "com_err_" may be used in any case to obtain explanations of error codes. The codes that may be returned are:

   graphic_error_table_$no_wgs_yet
   error_table_$lsm_node_ob
   error_table_$lsm_index_type
   error_table_$lsm_invalid_type
   error_table_$lsm_seg_full
   error_table_$lsm_blk_len

2. node_no (fixed bin(18))
   is the unique ID of the graphic element created, and may be used in subsequent calls to graphic structure manipulation entry points. This value is valid only for the current WGS. A node_no of zero may be used as input to any entry or as one of the elements of a list to signify a null element. A call to graphic_manipulator_$init or moving a structure between the WGS and a PGS generally invalidates a node value. If code is nonzero, node_no is zero.

Entry: graphic_manipulator_$assign_name

This entry creates a symbol nonterminal graphic element. This element assigns a name to a graphic construct and enters that name in the graphic symbol table in the WGS. Only graphic constructs so named may be saved in a PGS.

Usage

```
declare graphic_manipulator_$assign_name entry (char(*), fixed bin(18),
     fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$assign_name (name, value_n, code);
```

where:

1. node_no and code (see generic arguments above).

2. name (Input)
   is a character string containing the name to be assigned to the graphic construct subordinate to "value_n". It is truncated at the first blank character.

3. value_n (Input)
   is the node value of the graphic construct being named.

Entry: graphic_manipulator_$create_array

This entry creates an array nonterminal graphic element. Although lists and arrays may be included in an array, all structure subordinate to an array is lost when the graphic structure is compiled. An array may be manipulated as a list by the Multics-resident portion of the graphics system, but not inside an intelligent graphic terminal.

Usage

```
declare graphic_manipulator_$create_array entry (dimension(*) fixed bin(18),
     fixed bin, fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$create_array (array, array_l, code);
```

where:

1. node_no and code (see generic arguments above).

2. array (Input)
   is an array of node values of terminal or nonterminal graphic elements.

3. array_l (Input)
   is the number of elements in the array to be used in creating the list.

Entry: graphic_manipulator_$create_color

This entry creates a graphic element that specifies color composition.

Usage

```
declare graphic_manipulator_$create_color entry (fixed bin(6), fixed bin(6),
     fixed bin(6), fixed bin(35) returns (fixed bin(18));

node_no = graphic_manipulator_$create_color (red_intensity, green_intensity,
     blue_intensity, code);
```

where:

1. node_no and code (see generic arguments above).

2. red_intensity (Input)
   is the intensity of red in the color resulting from the composition of three primary additive colors in the specified intensities.

3. green_intensity (Input)
   is the intensity of green.

4. blue_intensity (Input)
is the intensity of blue.

Notes

Intensities are truncated to 6-bit positive integers upon graphic structure compilation. Minimum intensity is 0; maximum intensity is 63. Because of the nonlinear nature of the additive color spectrum, and the differences in CRT phosphors, colors resulting from the same proportions of the primary additive colors but different absolute intensities may be of different hues.

Entry: graphic_manipulator_$create_data

This entry creates a data block terminal graphic element. This element allows arbitrary bit strings representing user data or special user program-to-terminal conventions to be embedded in a graphic structure, and to be stored in the structure and sent to the terminal.

Usage

```
declare graphic_manipulator_$create_data entry (fixed bin, bit(*)
     unaligned, fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$create_data (n_bits, string, code);
```

where:

1. node_no and code (see generic arguments above).

2. n_bits (Input)
is the number of bits in "string" that are to be included in the bit string.

3. string (Input)
is a bit string containing data to be stored in the structure and sent to the terminal in the first "n_bits" bits.

Notes

Users may wish to use the nonstandard Multics PL/I "size" built-in function or the standard "length" and "unspec" PL/I built-in functions (e.g., "length (unspec (item))") as an easy method of supplying "n_bits".

Entry: graphic_manipulator_$create_list

This entry creates a list nonterminal graphic element.

Usage

```
declare graphic_manipulator_$create_list entry (dimension(*) fixed bin(18),
     fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$create_list (array, array_l, code);
```

where:

1. node_no and code (see generic arguments above).

2. array (Input)
   is an array of node values of terminal or nonterminal graphic elements.

3. array_l (Input)
   is the number of elements in the array to be used in creating the list.

Notes

The size of the maximum list that may be created is 4094 elements.

Any elements of the array may contain zeroes as placeholders for future replacements.

Entry: graphic_manipulator_$create_mode

This entry creates all mode terminal elements.

Usage

```
declare graphic_manipulator_$create_mode entry (fixed bin(6)), fixed bin,
     fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$create_mode (type, mode, code);
```

where:

1. node_no and code (see generic arguments above).

2. type (Input)
is the type of mode element to be created, and is one of the following values:

16 intensity
17 linetype
18 sensitivity
19 blinking

3. mode (Input)
is the mode value for the particular mode element to be created. The defined mode values are:

intensity
0 invisible
[and intervening integers to]
7 full intensity (default)

linetype
0 solid line (default)
1 dashed line
2 dotted line
3 dashed-dotted line
4 long-dashed line

sensitivity (light pen)
0 insensitive (default)
1 sensitive

blinking
0 steady (default)
1 blinking

Notes

Although only a small fraction of possible mode values are defined, no checking is performed to verify that specified modes are defined. This allows for future expansion to additional mode values.

Upon translation to MSGC (see Section 2 of this manual), mode values are truncated to 6-bit positive integers, limiting mode values to the range (0, 63).

Entry: graphic_manipulator_$create_position

This entry creates all positional graphic elements.

Usage

```
declare graphic_manipulator_$create_position entry (fixed bin, float bin,
     float bin, float bin, fixed bin(35)) returns (fixed bin(18));
```

```
node_no = graphic_manipulator_$create_position (type, x, y, z, code);
```

where:

1. node_no and code (see generic arguments above).

2. type (Input)
   indicates which type of positional element is to be created among the following values:

   0 setposition (absolute position)
   1 setpoint (absolute position, visible point)
   2 vector (visible line from current position)
   3 shift (relative position)
   4 point (relative position, visible point)

3. x (Input)
   is the x dimension of the positional element to be created.

4. y (Input)
   is the y dimension of the positional element to be created.

5. z (Input)
   is the z dimension of the positional element to be created.

Notes

Although the visible portion of the virtual screen is bounded by -512e0 < (x,y,z) < 511e0, the virtual screen allows coordinates bounded by -2048e0 < (x,y,z) < 2047e0. This allows a user to display a picture that exceeds the bounds of the visible screen "window" without wraparound.

Entry: graphic_manipulator_$create_rotation

This entry creates a three-dimensional rotation terminal graphic element.

Usage

```
declare graphic_manipulator_$create_rotation entry (float bin, float bin
     float bin, fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$create_rotation (x_angle, y_angle, z_angle,
     code);
```

where:

1. node_no and code (see generic arguments above).

2. x_angle (Input)
   is the number of degrees a graphic structure is to be rotated around the x-axis.

3. y_angle (Input)
   is the number of degrees a graphic structure is to be rotated around the y-axis.

4. z_angle (Input)
   is the number of degrees a graphic structure is to be rotated around the z-axis.

Notes

Rotation is performed around the x-axis first, then the y-axis, then the z-axis. Angles are transformed into positive angles in the range 0e0 ≤ angle < 360e0. If scaling, rotation, and extent elements appear in the same list or array, succeeding graphic constructs in the list are scaled first, then rotated, then clipped and masked.

Entry: graphic_manipulator_$create_scale

This entry creates a three-dimensional scaling terminal graphic element.

Usage

```
declare graphic_manipulator_$create_scale entry (float bin, float bin,
     float bin, fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$create_scale (x_scale, y_scale, z_scale,
     code);
```

where:

1. node_no and code (see generic arguments above).

2. x_scale (Input)
   is the factor by which all dimensions parallel to the stationary x-(left-to-right) axis are to be scaled.

3. y_scale (Input)
   is the factor by which all dimensions parallel to the stationary y-(bottom-to-top) axis are to be scaled.

4. z_scale (Input)
   is the factor by which all dimensions parallel to the stationary z-(back-to-front) axis are to be scaled.

Notes

Scale factors may be negative to produce mirror images. Scaling is performed independently in each dimension, relative to the stationary axes. If scaling, rotation, and extent elements appear in the same list or array, succeeding graphic constructs in the list are scaled first, then rotated, then clipped and masked.

Entry: graphic_manipulator_$create_text

This entry creates a text terminal graphic element. This allows graphic constructs to be labeled and captioned.

Usage

```
declare graphic_manipulator_$create_text entry (fixed bin, fixed bin,
     char(*) unaligned, fixed bin(35)) returns (fixed bin(18));

node_no = graphic_manipulator_$create_text (alignment, n_chars, string,
     code);
```

where:

1. node_no and code (see generic arguments above).

2. alignment (Input)
   indicates which portion of the character string is displayed at the current screen position as follows:

   1 top left
   2 top center
   3 top right
   4 middle left
   5 dead center
   6 middle right
   7 bottom left
   8 bottom center
   9 bottom right

3. n_chars (Input)
   is the number of characters in string to be taken as the text.

4. string (Input)
   is a character string containing the string to be used in the text element in its first "n_chars" characters.

Notes

Character strings are not rotated, scaled, partially clipped, or masked, but are subject to shifts of position produced by rotation and scaling. This allows the use of character-generating facilities of a graphics terminal and keeps labels readable.

Any printing ASCII character and newline may appear in string.

Structure Manipulation Entry Points

Entry: graphic_manipulator_$add_element

This entry inserts new elements in a list.

Usage

```
declare graphic_manipulator_$add_element (fixed bin(18), fixed bin,
     fixed bin(18), fixed bin(35));

call graphic_manipulator_$add_element (list_n, index, new_n, code);
```

where:

1. list_n (Input)
   is the node value of the list or array to which an element is added.

2. index (Input)
   is the index in the list after which the new element is added. It may have the following values:

0	the new element is added at the head of the list
-1	the new element is added at the end of the list
other	the new element is added after the specified element

3. new_n (Input)
   is the node value of the new graphic construct added to the list.

4. code (Output)
   is a standard status code.

Notes

An error occurs if the addition of an element to a list causes that list to exceed 4094 elements.

Entry: graphic_manipulator_$remove_symbol

This entry allows the user to "undefine" some symbol in the WGS. The symbol node itself is replaced by a reference to whatever was its value. This allows all graphics structures in the WGS that used that symbol by name to continue to work, via a direct reference to whatever was the value of that symbol before removal. All sharing relationships are preserved.

Usage

```
declare graphic_manipulator_$remove_symbol entry (char(*), fixed bin(35));

call graphic_manipulator_$remove_symbol (name, code);
```

where:

1. name (Input)
   is the name of the symbol which is removed.

2. code (Output)
   is a standard status code.

Entry: graphic_manipulator_$replace_element

This entry replaces list or array elements.

Usage

```
declare graphic_manipulator_$replace_element entry (fixed bin(18), fixed bin,
     fixed bin(18), fixed bin(35)) returns (fixed bin(18));

old_n = graphic_manipulator_$replace_element (list_n, index, new_n, code);
```

where:

1. old_n
   is the node value of the graphic construct that was replaced.

2. list_n (Input)
   is the node value of the list or array in which an element is replaced.

3. index (Input)
   is the index of the list element replaced.

4. new_n (Input)
   is the node value of a graphic construct that is to replace the element of the list or array pointed to by index.

5. code (Output)
is a standard status code.

Entry: graphic_manipulator_$replace_node

This entry replaces all shared instances of a graphic substructure with a given substructure.

Usage

```
declare graphic_manipulator_$replace_node entry (fixed bin(18),
     fixed bin(18), fixed bin(35));

call graphic_manipulator_$replace_node (old_n, new_n, code);
```

where:

1. old_n (Input)
is the node value of the old structure to be replaced.

2. new_n (Input)
is the node value of the new node.

3. code (Output)
is a standard status code.

Notes

The node "old_n" is destroyed in the process of being replaced.

Entry: graphic_manipulator_$replicate

This entry creates a new copy of a given substructure. No graphic elements in the old and new copies are shared.

Usage

```
declare graphic_manipulator_$replicate entry (fixed bin(18), fixed bin(35))
     returns (fixed bin(18));

new_n = graphic_manipulator_$replicate (template_n, code);
```

where:

1. new_n (Output)
   is the node value of the new copy.

2. template_n (Input)
   is the node value of the graphic substructure used as a template.

3. code (Output)
   is a standard status code.

Structure Examination Entry Points

Entry: graphic_manipulator_$examine_color

This entry locates a color element and returns the intensities of the three primary additive colors.

Usage

```
declare graphic_manipulator_$examine_color entry (fixed bin(18),
     float bin, float bin, float bin, fixed bin(35));

call graphic_manipulator_$examine_color (node_n, int_red, int_green,
     int_blue, code);
```

where:

1. node_n (Input)
   is the node value of the color element.

2. int_red (Output)
   is the intensity of red.

3. int_green (Output)
   is the intensity of green.

4. int_blue (Output)
   is the intensity of blue.

5. code (Output)
   is a standard status code.

Entry: graphic_manipulator_$examine_data

This entry returns the bit string contents of a datablock terminal graphic element.

Usage

```
declare graphic_manipulator_$examine_data (fixed bin(18), fixed bin
     bit(*), fixed bin(35));

call graphic_manipulator_$examine_data (node_n, n_bits, data, code);
```

where:

1. node_n (Input)
   is the node value of the data block element to be examined.

2. n_bits (Output)
   is the number of data bits.

3. data (Output)
   contains the user data.

4. code (Output)
   is a standard status code.

Entry: graphic_manipulator_$examine_list

This entry returns the contents of a list or array nonterminal graphic element.

Usage

```
declare graphic_manipulator_$examine_list entry (fixed bin(18), dimension(*)
     fixed bin(18), fixed bin, fixed bin(35));

call graphic_manipulator_$examine_list (node_n, array, array_l, code);
```

where:

1. node_n (Input)
   is the node value of the list or array node.

2. array (Output)
   is an array of node values representing the contents of the list or array.

3. array_l (Output)
is the number of element of the array that represent the list or array.

4. code (Output)
is a standard status code.

Notes

If the array is too small to hold the entire list, the error code "error_table_$smallarg" is returned. If this is the case, array_l contains the length of the array required to hold the entire list, and the entry may be called again with an array of that size or larger.

Entry: graphic_manipulator_$examine_mapping

This entry examines a mapping terminal graphic element.

Usage

```
declare graphic_manipulator_$examine_mapping entry (fixed bin(18),
     fixed bin, float bin dimension(*), fixed bin, fixed bin(35));

call graphic_manipulator_$examine_mapping (node_n, type, array, array_l,
     code);
```

where:

1. node_n (Input)
is the node value of the mapping element to be examined.

2. type (Output)
is the type of the mapping element, and is one of the following:

8	scaling
9	rotation
10	clipping

3. array (Output)
contains the argument values of the mapping element being examined.

4. array_l (Output)
is the number of elements in the array. This should be 3 for rotation and scaling, and 6 for clipping.

5. code (Output)
is a standard status code.

Notes

It is recommended that a user supply an array of at least dimension 6 to avoid problems.

Entry: graphic_manipulator_$examine_mode

This entry obtains a mode value of a mode graphic element. This entry applies only to mode elements with a single mode value, and not to mode elements with several values (such as color).

Usage

```
declare graphic_manipulator_$examine_mode entry (fixed bin(18), fixed bin,
     fixed bin, fixed bin(35));

call graphic_manipulator_$examine_mode (node_n, etype, mode, code);
```

where:

1. node_n (Input)
   is the node value of the mode element whose mode value is desired.

2. etype (Output)
   is the type of element, and is one of the following values:

   -1 not a mode element
   16 intensity
   17 line type
   18 sensitivity
   19 blinking

3. mode (Output)
   is the mode value.

4. code (Output)
   is a standard status code.

Entry: graphic_manipulator_$examine_position

This entry locates a positional graphic element and returns the x, y, and z coordinates.

Usage

```
declare graphic_manipulator_$examine_position entry (fixed bin(18),
     fixed bin, float bin, float bin, float bin, fixed bin(35));

call graphic_manipulator_$examine_position (node_n, etype, x, y, z, code);
```

where:

1. node_n (Input)
   is the node value of a positional graphic element whose coordinates are desired.

2. etype (Output)
   is the type of element, and is one of the following values:

   -1 not a positional element
    0 setposition
    1 setpoint
    2 vector
    3 shift
    4 point

3. x (Output)
   is the x value of the element.

4. y (Output)
   is the y value of the element.

5. z (Output)
   is the z value of the element.

6. code (Output)
   is a standard status code.

Entry: graphic_manipulator_$examine_symbol

This entry returns the name and node value of a named graphic substructure associated with a particular symbol node in the graphic symbol table.

Usage

```
declare graphic_manipulator_$examine_symbol entry (fixed bin(18), fixed bin(18),
     fixed bin, char(*), fixed bin(35));

call graphic_manipulator_$examine_symbol (node_n, value_n, n_chars, name,
     code);
```

where:

1. node_n (Input)
   is the node value of a symbol node.

2. value_n (Output)
is the node value of the graphic substructure named by this symbol.

3. n_chars (Output)
is the number of characters in the name.

4. name (Output)
is the name associated with the substructure.

5. code (Output)
is a standard status code.

Notes

The normal use of this entry is to examine individual symbols after listing the entire graphic symbol table with the graphic_manipulator_$examine_symtab entry point.

Entry: graphic_manipulator_$examine_symtab

This entry lists the symbols in the symbol table of the WGS.

Usage

```
declare graphic_manipulator_$examine_symtab entry (dimension(*)
      fixed bin(18), fixed bin, fixed bin(35));

call graphic_manipulator_$examine_symtab (array, array_l, code);
```

where:

1. array (Output)
contains the node values of the symbols in the symbol table.

2. array_l (Output)
is the number of symbols in the symbol table.

3. code (Output)
is a standard status code.

Notes

If array is too small, the error code "error_table_$smallarg" is returned, and "array_l" contains the size of array needed to hold the entire symbol table.

Entry: graphic_manipulator_$examine_text

This entry returns the alignment and character string value of a text graphic element.

Usage

```
declare graphic_manipulator_$examine_text entry (fixed bin(18), fixed bin,
     fixed bin, char(*), fixed bin(35));

call graphic_manipulator_$examine_text (node_n, alignment, n_chars, text,
     code);
```

where:

1. node_n (Input)
   is the node value of a text element.

2. alignment (Output)
   is the alignment of the text string.

3. n_chars (Output)
   is the number of characters in the text string.

4. text (Output)
   is the actual text.

5. code (Output)
   is a standard status code.

Entry: graphic_manipulator_$examine_type

This entry locates a graphic element and returns its type.

Usage

```
declare graphic_manipulator_$examine_type entry (fixed bin(18), bit(1)
     aligned, fixed bin, fixed bin(35));

call graphic_manipulator_$examine_type (node_n, t_nt, type, code);
```

where:

1. node_n (Input)
   is the node value of a graphic element whose type is desired.

2. t_nt (Output)
   is "0"b if the node is a terminal graphic element, "1"b if it is a nonterminal graphic element (list, array, or symbol).

3. type (Output)
is the type of the node, and is one of the following values:

-2	bad type
-1	null node
0	setposition
1	setpoint
2	vector
3	shift
4	point
8	scale
9	rotate
10	clip
16	intensity
17	line type
18	sensitivity
19	blinking
20	color
24	symbol
25	text
26	data
32	list
33	array

4. code (Output)
is a standard status code.

Notes

The introductory paragraphs of this subroutine description discuss PL/I include file "graphic_etypes.incl.pl1" which is useful in decoding the "type" argument, and similar arguments in various entries.

Entry: graphic_manipulator_$find_structure

This entry returns the node values of the graphic structure and symbol node of a named structure.

Usage

```
declare graphic_manipulator_$find_structure entry (char(*), fixed bin(18),
     fixed bin(35)) returns (fixed bin(18));

sym_n = graphic_manipulator_$find_structure (name, value_n, code);
```

where:

1. sym_n (Output)
is the node value of the symbol node naming the structure, and is 0 if the name is not found.

2. name (Input)
is the name of the structure to be located, and is truncated at the first blank.

3. value_n (Output)
is the node value of the graphic structure with the specified name, and is 0 if the name is not found in the graphic symbol table.

4. code (Output)
is a standard status code

Graphic Structure Storage Entry Points

The following entry points move portions of graphic structures between the WGS and one or more PGSs.

There are several generic arguments:

1. dname (char(*)) (Input)
is the name of the directory containing the desired PGS. If dname is the null string, the PGS specified by ename is located via the graphics search list (see "Notes" below).

2. ename (char(*)) (Input)
is the name of a PGS in the directory specified by dname. If the suffix ".pgs" is not explicitly provided, it is appended.

3. name (char(*)) (Input)
is the name of a named substructure in the WGS or PGS that is specified by dname and ename.

4. code (Output)
is a standard status code.

Notes

The graphics search list is described fully in Section 3.

graphic_manipulator_

Entry: graphic_manipulator_$get_struc

This entry moves a named graphic substructure from a PGS into the WGS.

Usage

```
declare graphic_manipulator_$get_struc entry (char(*), char(*), char(*),
     fixed bin(2), fixed bin(35));

call graphic_manipulator_$get_struc (dname, ename, name, merge, code);
```

where:

1. dname, ename, name, and code (see generic arguments above).

2. merge

determines the disposition of named substructures in the structure being moved from the PGS, and is one of the following values:

0 the entire graphic substructure in the PGS is moved into the WGS. Names of named substructures are entered into the graphic symbol table of the WGS as they are moved. If a name already exists in the WGS symbol table, the error code "graphic_error_table_$struc_duplication" is returned, and structure movement is aborted.

1 identical to 0, except that on naming conflicts, the old symbol and its substructure in the WGS are replaced by the new substructure.

2 instances of named substructures in the structure being copied from the PGS are replaced by identically-named substructures already in the WGS. If a name in the PGS is not found in the graphic symbol table of the WGS, a symbol with that name and no associated substructure is created in the WGS.

3 identical to 2, except that the PGS substructures with names not in the WGS symbol table are copied into the WGS.

Notes

The merge argument provides the flexibility of global replacements of named graphic entities. Normal usages of get_struc are:

Value of "merge"	Operation
0	moving a structure into an empty WGS.
1	moving a structure into a nonempty WGS to restore named substructures to their previously saved state (e.g., when one has been editing a graphic structure and has decided to start again with a fresh copy of the original).
2 or 3	newly designed substructures in the WGS are to replace identically named substructures in a graphic structure in a PGS when the structure is moved into the WGS.

Node values are not preserved across this operation.

Entry: graphic_manipulator_$put_struc

This entry moves a named graphic substructure from the WGS into a PGS.

Usage

```
declare graphic_manipulator_$put_struc entry (char(*), char(*), char(*),
     fixed bin(2), fixed bin(35));

call graphic_manipulator_$put_struc (dname, ename, name, merge, code);
```

where:

1. dname, ename, name, and code (see generic arguments above).

2. merge
   determines the disposition of named substructures in the structure being moved from the WGS, and is one of:

   0 the entire graphic substructure in the WGS is moved into the PGS. Names of named substructures are entered into the graphic symbol table of the PGS as they are moved. If a name already exists in the PGS symbol table, the error code "graphic_error_table_$struc_duplication" is returned, and structure movement is aborted.

   1 identical to 0, except that on naming conflicts, the old symbol and its substructure in the PGS are replaced by the new substructure.

   2 instances of named substructures in the structure being copied from the WGS are replaced by identically named substructures already in the PGS. If a name in the WGS is not found in the graphic symbol table of the PGS, a symbol with that name and no associated substructure is created in the PGS.

   3 identical to 2, except that the WGS substructures with names not found in the PGS symbol table are copied into the PGS.

Notes

The merge argument provides the flexibility of global replacements of named graphic entities. Normal usages of put_struc are:

Value of "merge"	Operation
0	moving a structure into an empty PGS.
1	moving a structure into a nonempty PGS when one wishes to replace old named substructures in the PGS with new ones of the same name.
2 or 3	newly designed substructures in the WGS are to replace identically named substructures in a PGS.

Node values are not preserved over this operation.

Entry: graphic_manipulator_$save_file

This entry saves an entire graphic structure in the WGS in a PGS whose name is specified.

Usage

```
declare graphic_manipulator_$save_file entry (char(*), char(*),
     fixed bin(35));

call graphic_manipulator_$save_file (dname, ename, code);
```

where dname, ename, code (see generic arguments above).

Notes

The PGS specified by dname and ename is reinitialized (all previous contents are destroyed) before the entire graphic structure resident in the WGS is copied. Node values are not preserved over this operation.

Entry: graphic_manipulator_$use_file

This entry moves an entire graphic structure saved in a PGS into the WGS.

Usage

```
declare graphic_manipulator_$use_file entry (char(*), char(*),
     fixed bin(35));

call graphic_manipulator_$use_file (dname, ename, code);
```

where dname, ename, code (see generic arguments above).

Notes

The WGS is reinitialized (all previous contents are destroyed) before the graphic structure resident in the PGS is copied. Node values are not preserved over this operation.

Name: graphic_operator_, go_

This subroutine contains entry points for performing animation on a graphics terminal, obtaining graphic input from a user, initiating graphic interactions between a user and the terminal, and controlling special terminal functions.

All entry points generate MSGC for the operations they perform, and output this code over the I/O switch named graphic_output. Those entry points that look for graphic input from a terminal expect it over the I/O switch named graphic_input. Entries are provided to allow users to specify their own I/O switches for these operations, if desired.

Complete descriptions of the various dynamic graphic operators may be found in Section 3 of this document.

Because of the multitude of entry points in graphic_operator_, declarations for all the user-callable entry points are contained in the PL/I include file "go_entry_dcls.incl.pl1" (see Section 8). Users may include this file (using the pl1 "%include" facility) in their source programs to save typing and syntax errors.

FORTRAN programmers should check "Programming Considerations" in Section 2 for special instructions about entries that return fixed bin quantities.

Generic Entries

All of the entries in graphic_operator_ that perform output have counterparts that perform the same operation over a user-specified I/O switch. Each of these entries has the same calling sequence as its counterpart, but takes one additional argument in the last argument position:

switch_ptr (Input)
is a pointer to the I/O switch on which the output is desired. If this is null, switch "graphic_output" is assumed.

The "variable switch" entry points are named similarly to their counterparts, with the suffix "_switch". They are:

ENTRY	VARIABLE SWITCH COUNTERPART
control	control_switch
delete	delete_switch
dispatch	dispatch_switch
display	display_switch
erase	erase_switch
increment	increment_switch
pause	pause_switch
replace_element	replace_element_switch
synchronize	synchronize_switch

Entry points that perform both input and output have counterparts that perform the same operation over user-specified I/O switches. Each of these entries uses the same calling sequence as its counterpart, but takes two additional arguments in the last argument positions:

input_switch_ptr (Input)
is a pointer to the switch on which the input is desired. If this is null, switch "graphic_input" is assumed.

output_switch_ptr (Input)
is a pointer to the I/O switch on which the output is desired. If this is null, switch "graphic_output" is assumed.

The "variable switch" counterparts follow the same naming convention described above. They are:

ENTRY	VARIABLE SWITCH COUNTERPART
what	what_switch
where	where_switch
which	which_switch

Animation Entry Points

Entry: graphic_operator_$increment

This entry increments the parameter values of a single positional, modal, or mapping terminal graphic element a specified number of times, with a specified delay between increments.

Usage

```
declare graphic_operator_$increment entry (fixed bin(18), fixed bin,
      float bin, fixed bin(18), fixed bin(35));

call graphic_operator_$increment (node_n, no_iter, delay, incr_n, code);
```

where:

1. node_n (Input)
   is the node value of the positional, modal, or mapping element incremented.

2. no_iter (Input)
   is the number of times the incrementation is performed.

3. delay (Input)
   is the number of seconds of real time the graphics terminal is to delay before performing each increment (including the first increment).

4. incr_n (Input)
is the node value of a graphic element, of the same type as node_n, that contains the increments for each parameter in node_n.

5. code (Output)
is a standard status code.

Notes

The parameters of the graphic element whose node value is "node_n" are updated to reflect their new values after incrementation.

The terminal delays the specified number of seconds (rounded to the nearest 1/64th of a second) before each increment, including the first. The number of seconds is truncated to 11 bits of integer precision (2047e0).

Any number of increment operations may be performed in parallel.

Entry: graphic_operator_$replace_element

This entry replaces an element in a list, resident in terminal memory, with another element, also resident in terminal memory.

Usage

```
declare graphic_operator_$replace_element entry (fixed bin(18), fixed bin,
     fixed bin(18), fixed bin(35)) returns (fixed bin(18));

old_n = graphic_operator_$replace_element (list_n, index, new_n, code);
```

where:

1. old_n (Output)
is the node value of the element replaced.

2. list_n (Input)
is the node value of the list containing the element to be replaced.

3. index (Input)
is the index in the list of the element to be replaced.

4. new_n (Input)
is the node value of the node to replace the element of list "list_n" pointed to by index. It must already be resident in terminal memory.

5. code (Output)
is a standard status code.

Entry: graphic_operator_$synchronize

This entry causes the graphics terminal to complete all previously received graphic operators before processing any following graphic operators. Its primary use is to synchronize operations going on in parallel.

Usage

```
declare graphic_operator_$synchronize entry (fixed bin(35));

call graphic_operator_$synchronize (code);
```

where code (Output) is a standard status code.

Input and User Interaction Entry Points

Entry: graphic_operator_$control

This entry places a positional graphic element in terminal memory under control of the user. The new absolute or relative coordinates of the element are updated in the WGS.

Usage

```
declare graphic_operator_$control entry (fixed bin(18), fixed bin(35));

call graphic_operator_$control (node_n, code);
```

where:

1. node_n (Input)
   is the node value of a positional node to be placed under user control.

2. code (Output)
   is a standard status code.

Notes

The exact nature of the interaction between the user and the terminal during the operation is left to the terminal programming.

Entry: graphic_operator_$pause

This entry causes the graphics terminal to pause before performing any subsequent graphic operation. The terminal remains in this state until the user indicates via some local interaction a readiness to proceed.

Usage

```
declare graphic_operator_$pause entry (fixed bin(35));

call graphic_operator_$pause (code);
```

where code (Output) is a standard status code.

Notes

The exact nature of the interaction between the user and the terminal during this operation is left to the terminal programming.

Entry: graphic_operator_$what

This entry is used to obtain a "what" graphic input. The structure sent by the terminal is decompiled and turned into an equivalent structure in the WGS.

Usage

```
declare graphic_operator_$what entry (fixed bin, fixed bin, fixed bin(35))
     returns (fixed bin(18));

node_no = graphic_operantor_$what (device, device_used, code);
```

where:

1. node_no (Output)
   is a node value specifying the structure in the WGS that was created from the input returned.

2. device (Input)
   tells the graphics terminal which type of graphic input device the user will use to give a graphic input. It is one of the following values:

   -1 any device (user chooses at run time)
    0 terminal processor or program
    1 keyboard
    2 mouse
    3 joystick

4 tablet and pen
5 light pen
6 trackball

3. dev_used (Output)
is the type of graphic input device actually used to produce the graphic input.

4. code (Output)
is a standard status code.

Notes

The exact nature of the interaction between the user and the terminal to produce a "what" input is left to the terminal programming.

Entry: graphic_operator_$where

This entry returns a "where" graphic input consisting of three absolute coordinate positions.

Usage

```
declare graphic_operator_$where entry (fixed bin, float bin, float bin,
     float bin, fixed bin(35));

call graphic_operator_$where (device, x, y, z, code);
```

where:

1. device (Input)
same as in "what" entry above.

2. x (Output)
is the x coordinate of the position indicated by the user.

3. y (Output)
is the y coordinate of the position indicated by the user.

4. z (Output)
is the z coordinate of the position indicated by the user.

5. code (Output)
is a standard status code.

Notes

The protocol by which a user lets the terminal know which input device to use (if there is a choice) is left to the terminal programming. Terminals not implementing a requested device have two options: they may map the device into an equivalent implemented device, or they may return an error to the graphics system. This may be done either by the terminal (if intelligent) or by the terminal's support procedure (if a static display).

Entry: graphic_operator_$which

This entry returns a "which" graphic input consisting of a unique path in the tree-structured graphic structure resident in the graphics terminal of the graphic substructure indicated by the user.

Usage

```
declare graphic_operator_$which entry (fixed bin, fixed bin(18), fixed bin,
     dimension(*) fixed bin, fixed bin(35));

call graphic_operator_$which (device, top_n, depth, path_array, code);
```

where:

1. device (Input)
   same as in "where" entry above.

2. top_n (Output)
   is the node value of the top-level node of a graphic structure resident in terminal memory.

3. depth (Output)
   is the number of structure levels in the path to the substructure indicated by the user.

4. path_array (Output)
   is an array of list indexes comprising a unique path through the structure to the indicated substructure.

5. code (Output)
   is a standard status code.

Notes

If path_array is too small to hold the entire path, the error code error_table_$smallarg is returned. In this case, depth contains the size of array needed to hold the entire tree pathname, and the input is saved in internal static storage, where it may be obtained by a subsequent call to graphic_operator_$which with an array of sufficient size. The input is saved

until a successful call to graphic_operator_$which returns it, or until freed by a call to graphic_operator_$reset.

The interaction between the user and the terminal to produce a "which" input is left to the terminal programming.

Terminal Control Entry Points

Entry: graphic_operator_$delete

This entry deletes all graphic structure subordinate to and including a given node from graphics terminal memory.

Usage

```
declare graphic_operator_$delete (fixed bin(18), fixed bin(35));

call graphic_operator_$delete (node_no, code);
```

where:

1. node_no (Input)
   is the node value of a graphic structure already resident in terminal memory that is to be deleted. If zero, all graphic structures in terminal memory are deleted.

2. code (Output)
   is a standard status code.

Notes

If node_no is not resident in terminal memory, the error message "graphic_error_table_$node_not_active" is returned.

This operation has no effect on a static device.

This operation deletes whole subtrees of graphic structure. Any other structure that references the deleted node or any of its subordinates ceases to function. Therefore, it is not recommended that the delete function be used without first issuing an erase command.

Entry: graphic_operator_$dispatch

This entry dispatches to the terminal any graphic operators buffered while running in nonimmediate mode.

Usage

```
declare graphic_operator_$dispatch entry (fixed bin(35));

call graphic_operator_$dispatch (code);
```

where code (Output) is a standard status code.

Entry: graphic_operator_$display

This entry causes the graphic structure subordinate to and including a given node, already resident in the terminal memory, to be displayed on the terminal screen.

Usage

```
declare graphic_operator_$display entry (fixed bin(18), fixed bin(35));

call graphic_operator_$display (node_n, code);
```

where:

1. node_n (Input)
   is the node value of the top-level node of a graphic structure resident in terminal memory that is to be displayed.

2. code (Output)
   is a standard status code.

Notes

The node "node_n" may be any node resident in terminal memory (e.g., it may be a substructure in a larger structure in terminal memory).

Entry: graphic_operator_$erase

This entry erases all graphics currently displayed on the terminal screen.

Usage

```
declare graphic_operator_$erase entry (fixed bin(35));

call graphic_operator_$erase (code);
```

where code (Output) is a standard status code.

Notes

On intelligent, refresh terminals, this entry should not erase any information on the screen that is not part of a graphic structure (e.g., text from using the terminal as an alphanumeric terminal).

Entry: graphic_operator_$reset

This entry destroys any graphic operators buffered but not yet sent to the terminal when immediacy = "0"b. Additionally, it causes any saved "which" type input that has not yet been obtained to be discarded.

Usage

```
declare graphic_operator_$reset entry;

call graphic_operator_$reset;
```

there are no arguments.

Entry: graphic_operator_$set_immediacy

This entry sets an internal "immediacy" mode for the buffering of dynamic graphic operators to be sent to the terminal.

Usage

```
declare graphic_operator_$set_immediacy entry (bit(1) aligned,
     bit(1) aligned, fixed bin(35));

call graphic_operator_$set_immediacy (immediacy, prev_immediacy, code);
```

where:

1. immediacy (Input)
   determines whether dynamic graphic operators are sent as they are generated, or buffered until sent (explicitly by a call to graphic_operator_$dispatch, or implicitly by a call to an input operator). If immediacy = "0"b, then operators are buffered. If immediacy = "1"b (the initial default), then operators are sent as they are generated.

2. prev_immediacy (Output)
   is the value of the immediacy mode as it was prior to the set_immediacy call.

3. code (Output)
   is a standard status code.

Notes

Any operators already buffered whenever immediacy is set to "1"b are output to the terminal.

Name: graphic_terminal_status_

This subroutine interprets error messages sent from a remote programmable graphics terminal. An entry may be used to gain further information concerning the exact cause of a terminal error that has occurred. This entry should be used in conjunction with the information in the include file "graphic_terminal_errors.incl.pl1" (see Section 8).

Entry: graphic_terminal_status_$decode

This entry interprets the character string status message sent by an intelligent graphics terminal.

Usage

```
declare graphic_terminal_status_$decode entry (char(*), fixed bin(35));

call graphic_terminal_status_$decode (err_string, code);
```

where:

1. err_string (Input)
   is the acknowledgement message received from an intelligent terminal.

2. code (Output)
   is a standard status code.

Notes

Only status codes from graphic_error_table_ are returned by this entry. If the status code is nonzero, it is guaranteed to be one of the "terminal" errors in graphic_error_table_, distinguishable by a name of the form "graphic_error_table_$term_...". (See description of graphic_error_table_ in this section.)

Entry: graphic_terminal_status_$interpret

This entry extracts detailed information about the last graphics terminal error that has occurred.

Usage

```
declare graphic_terminal_status_$interpret entry (fixed bin(35), char(1),
     fixed bin, fixed bin, (*) fixed bin, fixed bin(35));

call graphic_terminal_status_$interpret (status_code, err_char, node, depth,
     path, code);
```

where:

1. status_code (Output)
   is the status code last returned by graphic_terminal_status_$decode.

2. err_char (Output)
   is the SPI representation of the status code returned by the terminal.

3. node (Output)
   is the top-level node of the structure resident in the graphics terminal that was being operated upon at the time of the error. If this node is zero, no structure figured in the error.

4. depth (Output)
   is the depth in the list structure, from the top-level node, at which the error occurred.

5. path (Output)
   the i_th element of path corresponds to the index of the element of the i_th level list that was active on the graphics terminal's display stack at the moment of the error. (Each element, except the last one, refers to a subroutine call. The whole array represents a kind of pathname of the exact node that caused the error and includes the history of its invocation.)

6. code (Output)
   is a standard status code.

Notes

This entry only returns one nonzero code, error_table_$smallarg. This signifies that the dimension of the path argument was less than the value of depth. The information is saved, however, and may be obtained on a subsequent try with a larger array area.

Name: gui_

The gui_ subroutine provides a means by which casual graphics users can communicate with the MGS using PL/I or FORTRAN.

The user should be aware that graphic item calls do not transmit picture fragments directly to the screen, but create a list structure of picture description elements in the WGS. The list is then transmitted to the screen by a call to gui_$gdisp.

Declarations for all the user-callable entry points in gui_ are contained in the PL/I include file "gui_entry_dcl.incl.pl1" (see Section 8). Users may include this file (using the PL/I "%include" facility) in their source programs to save typing and syntax errors.

Entry: gui_$garc

This entry creates a sequence of items directing that an arc be generated that runs from the current position.

Usage

```
declare gui_$garc entry (float bin, fixed bin, fixed bin);

call gui_$garc (q, dx, dy);
```

where:

1. q (Input)
   is the length (in radians/pi) of the circle to be drawn (e.g., q=2 specifies a complete circle.) If q is positive, the arc is drawn counterclockwise. If q is negative, the arc is drawn clockwise.

2. dx (Input)
   is the relative distance, in the x direction, from the current position to the center of the desired circle.

3. dy (Input)
   is the relative distance, in the y direction, from the current position to the center of the desired circle.

Entry: gui_$gbox

This entry creates a sequence of items directing that a box (of size dx by dy) be displayed starting from the current position. The current position defines one corner of the box. The first two sides of the box to be drawn are of lengths (dx, 0) and (0, dy) respectively. This allows the user to position the current graphic position at any corner of the box by controlling the signs of the arguments dx and dy.

This page intentionally left blank.

Usage

```
declare gui_$gbox entry (fixed bin, fixed bin);

call gui_$gbox (dx, dy);
```

where:

1. dx (Input)
   is the size of the box in the x direction.

2. dy (Input)
   is the size of the box in the y direction.

Entry: gui_$gcirc

This entry creates a sequence of items directing that a complete circle be displayed starting from and ending with the current position.

Usage

```
declare gui_$gcirc entry (fixed bin, fixed bin);

call gui_$gcirc (dx, dy);
```

where:

1. dx (Input)
   is the relative distance, in the x direction, from the current position to the center of the desired circle.

2. dy (Input)
   is the relative distance, in the y direction, from the current position to the center of the desired circle.

Entry: gui_$gdisp

This entry directs that everything currently on the display list be displayed. The initial call to gdisp erases the screen. Subsequent calls append fragments to the existing picture without erasing the screen unless gui_$geras has been called in the meanwhile.

Usage

```
declare gui_$gdisp entry;

call gui_$gdisp;
```

there are no arguments.

Entry: gui_$gdot

This entry directs that subsequent lines and figures should be drawn with solid or dotted lines.

Usage

```
declare gui_$gdot entry (fixed bin);

call gui_$gdot (type);
```

where type (Input) is an integer specifying the type of line to use and is one of the following values:

0 solid line
1 dashed line
2 dotted line
3 dashed-dotted line
4 long-dashed line

A PL/I include file, "graphic_etypes.incl.pl1", which declares mnemonic variables (e.g., solid, dotted) as representing these integers, is available for inclusion in the user's source program, through the PL/I "%include" facility.

Entry: gui_$geras

This entry ensures that the next call to gui_$gdisp causes the screen to be erased, and the entire display list to be redisplayed.

Usage

```
declare gui_$geras entry;

call gui_$geras;
```

there are no arguments.

Entry: gui_$geqs

This entry creates a sequence of items directing that a regular polygon of n sides be displayed starting from and ending with the current position. The center of this polygon is at a distance (dx, dy) from the current position. The current position defines one vertex of the polygon.

Usage

```
declare gui_$geqs entry (fixed bin, fixed bin, fixed bin);

call gui_$geqs (n, dx, dy);
```

where:

1. n (Input)
   is the desired number of sides. It must be less than 200.

2. dx (Input)
   is the relative distance, in the x direction, from the current position to the center of the desired polygon.

3. dy (Input)
   is the relative distance, in the y direction, from the current position to the center of the desired polygon.

Entry: gui_$ginit

This entry initializes the WGS and creates an empty display list with the name "gui_display_list_". This entry must be called prior to issuing any other calls to gui_. Subsequent calls to this entry reinitialize (i.e., destroy the contents of) the WGS.

Usage

```
declare gui_$ginit entry;

call gui_$ginit;
```

there are no arguments.

Entry: gui_$gpnt

This entry creates an item that directs that the current position be shifted by the specified increment in three-dimensions and directs that a visible point be displayed at this position.

Usage

```
declare gui_$gpnt entry (fixed bin, fixed bin, fixed bin);

call gui_$gpnt (dx, dy, dz);
```

where:

1. dx (Input)
   is the number of points by which the current position is shifted in the x direction.

2. dy (Input)
   is the number of points by which the current position is shifted in the y direction.

3. dz (Input)
   is the number of points by which the current position is shifted in the z direction.

Entry: gui_$grmv

This entry directs that everything in the display list be removed. This has no effect upon the current picture displayed, which remains until erased.

Usage

```
declare gui_$grmv entry;

call gui_$grmv;
```

there are no arguments.

Entry: gui_$gsft

This entry creates an item that directs that the current position be shifted by the specified increment in three-dimensions. This shift is not visible.

Usage

```
declare gui_$gsft entry (fixed bin, fixed bin, fixed bin);

call gui_$gsft (dx, dy, dz);
```

where:

1. dx (Input)
   is the number of points by which the current position is shifted in the x direction.

2. dy (Input)
   is the number of points by which the current position is shifted in the y direction.

3. dz (Input)
   is the number of points by which the current position is shifted in the z direction.

Entry: gui_$gsps

This entry creates an item that sets the current position in the display space to a specified point in three-dimensions and appends this item to the display list.

Usage

```
declare gui_$gsps entry (fixed bin, fixed bin, fixed bin);

call gui_$gsps (x, y, z);
```

where:

1. x (Input)
   is the absolute x coordinate of the desired point.

2. y (Input)
   is the absolute y coordinate of the desired point.

3. z (Input)
   is the absolute z coordinate of the desired point.

Entry: gui_$gspt

This entry is similar to gsps above, but directs that a visible point be displayed at this position.

Usage

```
declare gui_$gspt entry (fixed bin, fixed bin, fixed bin);

call gui_$gspt (x, y, z);
```

arguments are as above.

Entry: gui_$gtxt

This entry creates an item directing that a character string be displayed starting from the current position in a horizontal direction. The current position remains unchanged.

Usage

```
declare gui_$gtxt entry (char(*), fixed bin);

call gui_$gtxt (cstring, alignment);
```

where:

1. cstring (Input)
   is the character string to be displayed.

2. alignment (Input)
   specifies by which alignment point the string is to be aligned and is one of the following values:

   1 top left
   2 top center
   3 top right
   4 middle left
   5 dead center
   6 middle right
   7 bottom left
   8 bottom center
   9 bottom right

A PL/I include file, "graphic_etypes.incl.pl1", which declares mnemonic variables (e.g., Upper_left, Center) as representing these integers, is available for inclusion in the user's source program through the PL/I "%include" facility.

Entry: gui_$gvec

This entry creates an item that directs that a straight line of specified dimensions be constructed in three-dimensions starting at the current position.

Usage

```
declare gui_$gvec entry (fixed bin, fixed bin, fixed bin);

call gui_$gvec (dx, dy, dz);
```

where:

1. dx (Input)
   is the number of points by which the current position is shifted in the x direction.

2. dy (Input)
   is the number of points by which the current position is shifted in the y direction.

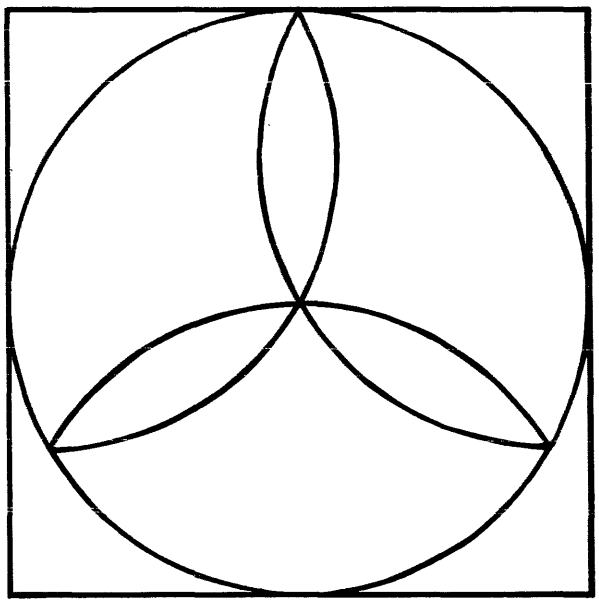
3. dz (Input)
   is the number of points by which the current position is shifted in the z direction.

Example of gui_

The following is an example of a program using gui_:

```
gui_demo: proc;
%include graphic_etypes;

dcl  gui_$gtxt entry (char(*), fixed bin),
     gui_$gcirc entry (fixed bin, fixed bin),
     gui_$gbox entry (fixed bin, fixed bin),
```

```
gui_$gsps entry (fixed bin, fixed bin, fixed bin),
gui_$gdisp entry,
gui_$ginit entry,
gui_$garc entry (float bin, fixed bin, fixed bin);

     call gui_$ginit;
     call gui_$gsps (300, 300, 0);
     call gui_$gbox (-600, -600);
     call gui_$gsps (0, 300, 0);
     call gui_$gcirc (0, -300);
     call gui_$garc (2/3, sqrt (3.000e0) * 150, -150);
     call gui_$garc (2/3, -sqrt (3.000e0) * 150, -150);
     call gui_$garc (2/3, 0, 300);

     call gui_$gsps (0, -400, 0);
     call gui_$gtxt ("MULTICS GRAPHICS SYSTEM," Upper_center);

     call gui_$gdisp;

end gui_demo;
```



MULTICS GRAPHICS SYSTEM

Name: plot_

This subroutine is a user interface that creates a two-dimensional graph from input data for use with Multics display terminals. The graph created is a cartesian graph, scaled so as to permit maximum coverage of the screen, and labeled in convenient increments to facilitate reading. This routine can be made to plot with either vectors connecting the data points, with a specified character displayed at each point plotted, or both. It also has facilities that enable the user to append a new plot over the one being currently displayed (in which case the new plot is scaled to match the old one), to suppress the grid (in which case only the left-most and lowest lines are displayed, with tick marks at increments), and to direct that the graph be scaled equally in both directions.

Declarations for all the user-callable entries in plot_ are contained in the PL/I include file "plot_entry_dcls.incl.pl1" (see Section 8). Users may include this file (using the PL/I "%include" facility) in their source programs to save typing and syntax errors. This include file also contains declarations of mnemonically-named "constant" variables that may be used to specify the options and modes accepted by the various entries of plot_.

FORTRAN programmers should be familiar with the requirements described in Section 2 under "Programming Considerations".

Usage

```
declare plot_ entry ((*) float bin, (*) float bin, fixed bin, fixed bin,
     char(1));

call plot_ (x, y, xydim, vec_sw, symbol);
```

where:

1. x (Input)
   is an array of x coordinates of points to be plotted.

2. y (Input)
   is an array of y coordinates of points to be plotted.

3. xydim (Input)
   is the number of elements in the x and y array pairs.

4. vec_sw (Input)
   can be one of the following values:

   1 if the vectors, but no symbol are desired
   2 if the symbol and connecting vectors are desired
   3 if the symbol, but no connecting vectors are desired

5. symbol (Input)
   is the symbol to be plotted at each point.

Notes

It is possible, by repetitive calls to plot_, to display any set of graphs on top of one another. All graphs after the first graph are scaled to the scale of the first. A call to plot_ erases the screen only if there was a call to plot_$setup prior to it. The only exception is that the first call to plot_ in a process always erases the screen whether or not plot_$setup has been called.

Default values for options are dotted grid, automatic scaling, no labels, and linear-linear plot.

The display list produced by plot_ is attached to the graphic symbol "plot_display_list_".

No clipping of data to fit the plot grid is performed.

The data given to plot_ is not sorted in any manner. This allows the plotting of relations as well as functions.

Entry: plot_$scale

This entry explicitly specifies plot scaling by specifying the extent of the axes in the x and y directions. If this scaling feature is desired, this entry must be called before any call to plot_ (i.e., immediately after a call to plot_$setup); otherwise, it is ignored.

Usage

```
declare plot_$scale entry (float bin, float bin, float bin, float bin);

call plot_$scale (xmin, xmax, ymin, ymax);
```

where:

1. xmin (Input)
   is the desired low bound of the x-axis.

2. xmax (Input)
   is the desired high bound of the x-axis.

3. ymin (Input)
   is the desired low bound of the y-axis.

4. ymax (Input)
   is the desired high bound of the y-axis.

Entry: plot_$setup

This entry sets parameters controlling the type of of plotting performed. The parameters specify the type of graph desired (log-log, linear, etc.), the type of grid desired (if any), and whether or not plot_ is to scale both axes equally. A call to this entry also ensures that the next call to plot_ erases the screen.

Usage

```
declare plot_$setup entry (char(*), char(*), char(*), fixed bin,
     float bin, fixed bin, fixed bin);

call plot_$setup (title, xlabel, ylabel, type, base, grid_sw, eq_scale_sw);
```

where:

1. title (Input)
   is the title of the graph. It is placed above the grid.

2. xlabel (Input)
   is the label desired along the x-axis.

3. ylabel (Input)
   is the label desired down the y-axis.

4. type (Input)
   can be one of the following values:

   1 linear-linear plot
   2 log-linear plot (log on x-axis)
   3 linear_log plot (log on y_axis)
   4 log-log plot

5. base (Input)
   is the logarithm base (for logarithmic plots).

6. grid_sw (Input)
   can be one of the following values:

   0 if tick marks and values are desired
   1 if dotted grid and values are desired
   2 if solid grid and values are desired
   3 if no grid or values are desired

7. eq_scale_sw (Input)
   can be one of the following values:

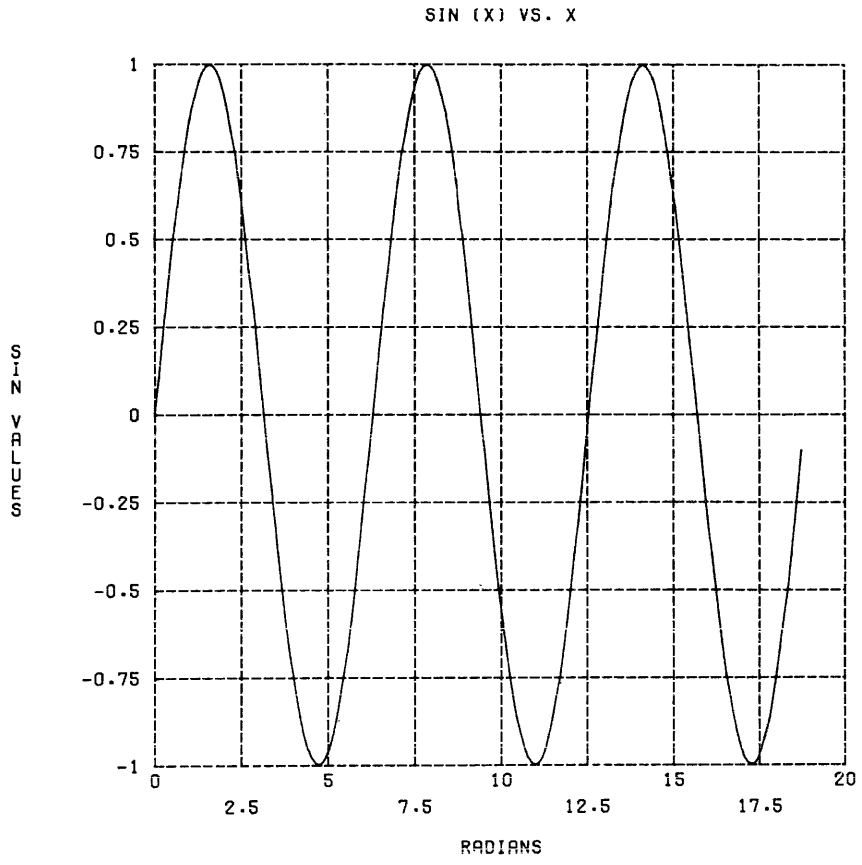
   0 if normal scaling is desired
   1 if the plot is to be scaled equally in both directions

Notes

Coordinates of linear axes are represented in simple numerical form (e.g., "5," "20.25"). Coordinates of logarithmic axes are represented in the form "eN," where N is the appropriate power of the base. For example, the coordinate value "e3" appearing on a log axis with a base of 10 represents 10**3 or 1000.

Example of plot

```
plot_example:  proc;

  declare x(180) float bin,
          y(180) float bin,
          i fixed bin,
          pi float bin static internal initial (3.14159e0),
          three_cyc float bin;

  %include plot_entry_dcls;
  declare (sin, float) builtin;

  three_cyc = 6e0*pi/180e0;

  do i = 1 to 180;
          x(i) = three_cyc * float (i-1);
          y(i) = sin (x(i));
          end;

  call plot_$setup ("SIN (X) VS. X", "RADIANS",
       "SIN VALUES", Linear_linear, 0e0, Dotted_grid, Normal_scaling);

  call plot_ (x, y, 180, Vectors_only, "");

  return;
  end;
```



SECTION 6

GRAPHIC DEVICE TABLES

This section contains descriptions of GDTs. Use of a GDT is described in Section 3, setup_graphics command.

The GDTs described are as follows:

ards	- Advanced Remote Display Station Terminal, Adage, Inc.
calcomp_915	- CalComp 915/1036 Plotter combination
rg512	- RetroGraphics 512 enhancement for ADM3A Terminals
tek_4002	- Tektronix Models 4002 and 4002A Terminals
tek_4012	- Tektronix Models 4012 and 4013 Terminals
tek_4014	- Tektronix Models 4014 and 4015 Terminals
tek_4662	- Tektronix Model 4662 Plotter

Name: ards

This GDT contains a description of the capabilities of the Advanced Remote Display Station. It has the added name "ARDS".

Description

This is a static terminal. Dynamic operators are not accepted.

Sensitivity, blinking, color, and extent are not implemented.

Any line_type other than solid is translated into the ARDS-provided dashed line.

Intensity of zero is recognized as invisible. Other values are translated to visible.

The query effector is not implemented.

Name: calcomp_915

This GDT contains a description of the capabilities of a CalComp 915/1036 Plotter combination.

Description

This is a static device. Dynamic operators are not accepted.

Sensitivity, blinking, and extent are not implemented.

Only the solid, dashed, and dotted line_types are implemented.

Intensity of zero is recognized as invisible. Other values are translated to visible.

The query effector is not implemented.

A limited implementation of the color facility is available. Use of this facility assumes that plotter pen "one" (rightmost) is loaded with blue ink, pen "two" with green ink, and pen "three" with red ink. Since only one of these pens may be active at a time, the pen corresponding to the color possessing the greatest intensity in a given color effector is used. If two or more colors in a particular effector possess equal intensity, the darker pen (the lower-numbered pen) is used.

Tapes produced are in the 6-bit format, regardless of whether they are 7-, or 9-track tapes.

Notes

This GDT cannot be used in the online mode (i.e., to the terminal). When specifying this GDT to the setup_graphics command, the user must also specify the "-offline" control argument.

Name: rg512

This GDT contains a description of the capabilities of the RetroGraphics 512 graphic enhancement for ADM3A terminals.

Description

This is a static terminal. Dynamic operators are not accepted.

Line_type, sensitivity, blinking, color, and extent are not implemented.

Intensity of zero is recognized as invisible. Other values are translated to visible.

Supported modes:

This GDT supports the following special modes:

baud=nnnn
specifies the baud rate at which the device is being used. Explicit specification of this mode is necessary only if the device is connected to the Multics system in such a manner as to make its baud rate indiscernible to the system (e.g., via a multihost network).

Name: tek_4002

This GDT contains a description of the capabilities of the Tektronix models 4002 and 4002A terminals. It has the added name "tek_4002A".

Description

This is a static terminal. Dynamic operators are not accepted.

Line_type, sensitivity, blinking, color, and extent are not implemented.

Intensity of zero is recognized as invisible. Other values are translated to visible.

The query effector is implemented only for the "where" operation, using crosshairs. After positioning the crosshairs to the desired point, the user * should press "return" to enter the desired point.

Supported modes:

This GDT supports the following special mode:

baud=nnnn
specifies the baud rate at which the device is being used. Explicit specification of this mode is necessary only if the device is connected to the Multics system in such a manner as to make its baud rate indiscernible to the system (e.g., via a multihost network).

tek_4012

Name: tek_4012

This GDT contains a description of the capabilities of the Tektronix models 4012 and 4013 terminals. It has the added name "tek_4013".

Description

This is a static terminal. Dynamic operators are not accepted.

Line_type, sensitivity, blinking, color, and extent are not implemented.

Intensity of zero is recognized as invisible. Other values are translated to visible.

The query effector is implemented only for the "where" operation, using the built-in crosshairs. After positioning the crosshairs to the desired point, the user should press "return" to enter the desired point. *

Supported modes:

This GDT supports the following special modes:

baud=nnnn
: specifies the baud rate at which the device is being used. Explicit specification of this mode is necessary only if the device is connected to the Multics system in such a manner as to make its baud rate indiscernible to the system (e.g., via a multihost network).

Name: tek_4014

This GDT contains a description of the capabilities of the Tektronix models 4014 and 4015 terminals. It has the added name "tek_4015".

Description

This is a static terminal. Dynamic operators are not accepted.

Sensitivity, blinking, color, and extent are not implemented.

Intensity of zero is recognized as invisible. Other values are translated to visible.

When used with a terminal possessing the Extended Graphics Module option, this GDT implements all values of line_type.

The variable character size (a feature of models 4014 and 4015) is forced to a known value when performing graphic text output, and reset to the smallest available size (the normal operating mode) after output is finished.

The query effector is implemented only for the "where" operation, using the built-in crosshairs. After positioning the crosshairs to the desired point, the * user should press "return" to enter the desired point.

Supported modes:

This GDT supports the following special modes:

baud=nnnn
specifies the baud rate at which the device is being used. Explicit specification of this mode is necessary only if the device is connected to the Multics system in such a manner as to make its baud rate indiscernible to the system (e.g., via a multihost network).

extaddr, ^extaddr
specifies whether the extended addressing feature is to be used. (The Extended Graphics Option module must be installed in the terminal for this mode to be effective.) Extended addressing allows the screen to be addressed to four times the usual precision but can add as much as 25% to character-transmission time and volume. By default, this mode is off.

Name: tek_4662

This GDT contains a description of the capabilities of the Tektronix model 4662 table-top plotter.

Description

This is a static terminal. Dynamic operators are not accepted.

Sensitivity, blinking, color, and extent are not implemented.

Intensity of zero is recognized as invisible. Other values are translated to visible.

The query effector is not implemented.

The erase effector does not pause to allow the paper to be changed. As on a normal terminal, it is expected that the user has provided some protocol to pause after the display of one picture before it is erased and replaced by another.

Supported modes:

This GDT supports the following special modes:

baud=nnnn
: specifies the baud rate at which the device is being used. Explicit specification of this mode is necessary only if the device is connected to the Multics system in such a manner as to make its baud rate indiscernible to the system (e.g., via a multihost network).

device=ch
: specifies the plotter addressing code. The character ch must be A, B, C, or D. The addressing code for a given plotter is specified via the user-settable switches on the rear of the plotter. If this mode is not specified, it defaults to device=A.

extaddr, ^extaddr
: specifies whether the extended addressing feature is to be used. Extended addressing allows the graphic area on the device to be addressed to four times the usual precision but can add as much as 25% to character-transmission time and volume. By default, this mode is off.

Notes

The Tektronix 4662 plotter contains user-settable switches that control the baud rate of the plotter, the addressing codes used, and several other parameters.

tek_4662

The following option settings are required:

Low plotting speed (A/8):	off (0)
Terminal mute (A/4):	on (0)
Copy mode (A/2):	on (1)
CR effect (A/1):	CR -> CR (0)
Delete interpretation (B/8):	DEL -> LOY (0)
RS232-C select (C/2):	on (1)

The device address setting should agree with the setting of the "device=" mode:

Device address (C/1+D/8):	A (00)
	B (01)
	C (10)
	D (11)

The setting of the GIN terminator (B/4+B/2) is unimportant, since graphic input is not implemented for this device.

The remaining settings (stop bits, parity, and baud) should be set according to the communication facilities to be used:

Stop bits (B/1):	1 bit (1)
	2 bits (0)
Parity (C/8+C/4):	odd (11)
	even (10)
	none (01 or 00)
Baud rate (D/4+D/2+D/1):	110 (100)
	150 (000)
	300 (001)
	600 (010)
	1200 (011)

SECTION 7

GRAPHIC CHARACTER TABLES

This section contains displays of the following GCTs:

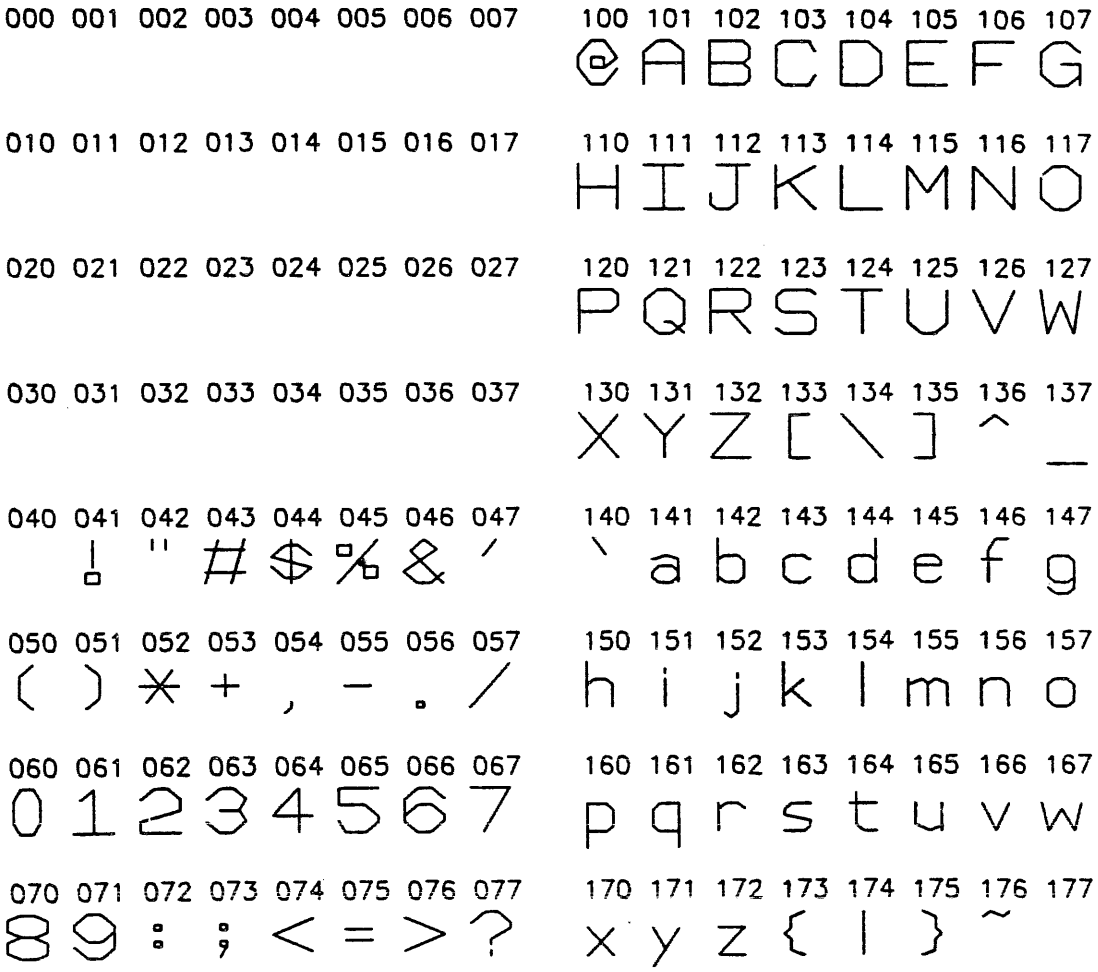
gct_block_roman_
gct_complex_italic_
gct_complex_roman_
gct_complex_script_
gct_duplex_roman_
gct_gothic_english_
gct_gothic_german_
gct_gothic_italian_
gct_simplex_roman_
gct_simplex_script_
gct_triplex_italic_
gct_triplex_roman_

Graphic Character Tables are fully described in Section 3.

gct_block_roman_

Name: gct_block_roman_

✱ The gct_block_roman_ graphic character set is displayed below.



The Quick Brown Fox Jumps over the Lazy Dog.

gct_block_roman_

Name: gct_complex_italic_

The gct_complex_italic_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.

000	001	002	003	004	005	006	007
							🔔
010	011	012	013	014	015	016	017
020	021	022	023	024	025	026	027
030	031	032	033	034	035	036	037
040	041	042	043	044	045	046	047
	!	"	#	$	%	&	'
050	051	052	053	054	055	056	057
(	)	*	+	,	−	.	/
060	061	062	063	064	065	066	067
0	1	2	3	4	5	6	7
070	071	072	073	074	075	076	077
8	9	:	;	<	=	>	?

100	101	102	103	104	105	106	107
@	A	B	C	D	E	F	G
110	111	112	113	114	115	116	117
H	I	J	K	L	M	N	O
120	121	122	123	124	125	126	127
P	Q	R	S	T	U	V	W
130	131	132	133	134	135	136	137
X	Y	Z	[	\	]	^	_
140	141	142	143	144	145	146	147
`	a	b	c	d	e	f	g
150	151	152	153	154	155	156	157
h	i	j	k	l	m	n	o
160	161	162	163	164	165	166	167
p	q	r	s	t	u	v	w
170	171	172	173	174	175	176	177
x	y	z	{	|	}	~	

The Quick Brown Fox Jumps over the Lazy Dog.

gct_complex_italic_

gct_complex_roman_

Name: gct_complex_roman_

The gct_complex_roman_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.

The Quick Brown Fox Jumps over the Lazy Dog.

gct_complex_roman_

Name: gct_complex_script_

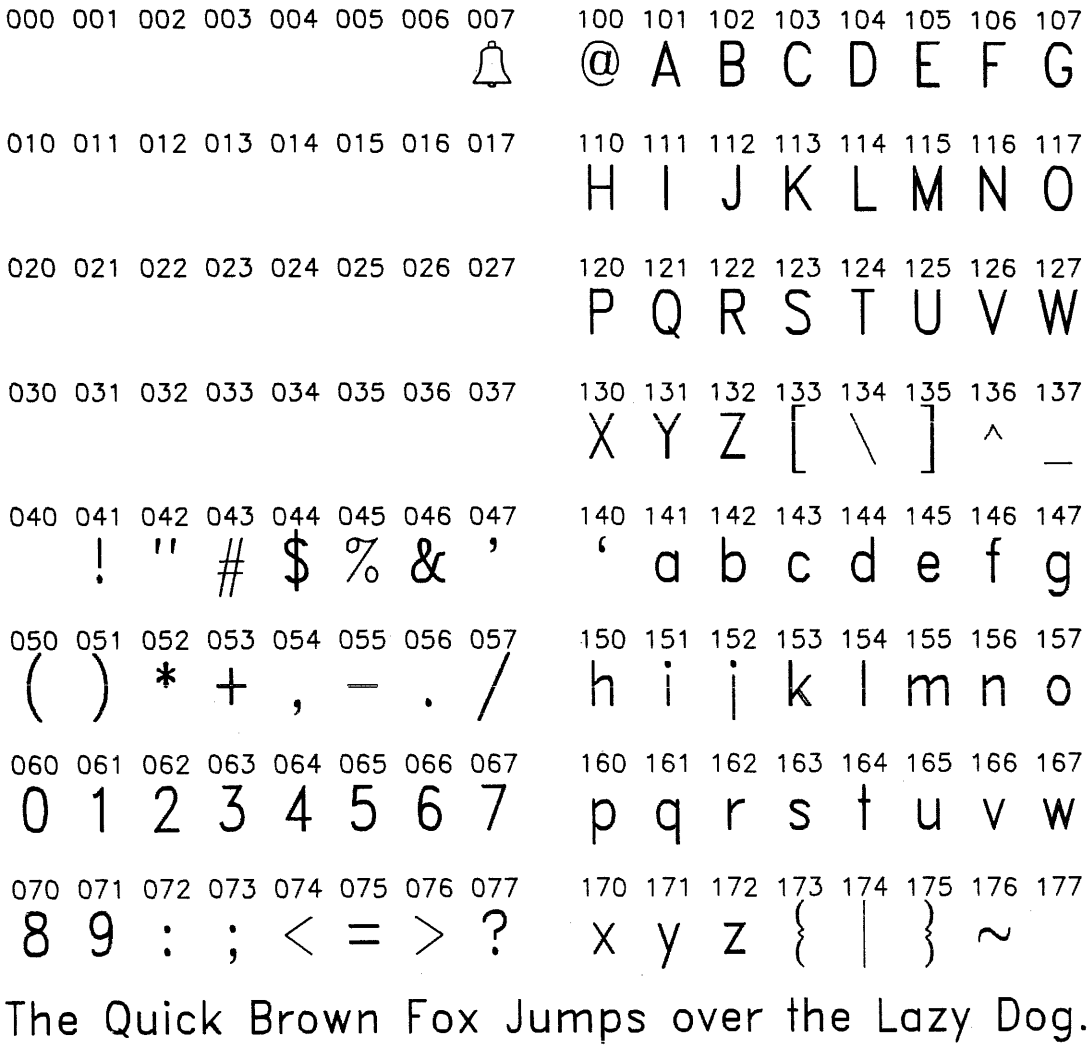
The gct_complex_script_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.

gct_complex_script_

gct_duplex_roman_

Name: gct_duplex_roman_

The gct_duplex_roman_ character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.



gct_duplex_roman_

gct_gothic_english_

Name: gct_gothic_english_

The gct_gothic_english_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.

gct_gothic_english_

gct_gothic_german_

Name: gct_gothic_german_

The gct_gothic_german_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.

Notes

The German alphabet from which this character set is derived contains three distinct versions of the letter "s" which are used variously according to well-defined rules of typography. The "s" which is used most often occupies the position usually occupied by "s" in the collating sequence. The other two versions occupy otherwise unused positions. Users for whom the strict correctness of the typography is important must perform some translation of their input strings before submitting them to graphic_chars_.

000	001	002	003	004	005	006	007	100	101	102	103	104	105	106	107
							🔔	@	𝔄	𝔅	ℭ	𝔇	𝔈	𝔉	𝔊
010	011	012	013	014	015	016	017	110	111	112	113	114	115	116	117
								ℌ	ℑ	𝔍	𝔎	𝔏	𝔐	𝔑	𝔒
020	021	022	023	024	025	026	027	120	121	122	123	124	125	126	127
								𝔓	𝔔	ℜ	𝔖	𝔗	𝔘	𝔙	𝔚
030	031	032	033	034	035	036	037	130	131	132	133	134	135	136	137
				ß	ß	ſ		𝔛	𝔜	ℨ	[	\	]	^	_
040	041	042	043	044	045	046	047	140	141	142	143	144	145	146	147
	!	"	#	$	%	&	'	`	a	b	c	d	e	f	g
050	051	052	053	054	055	056	057	150	151	152	153	154	155	156	157
(	)	*	+	,	-	.	/	h	i	j	k	l	m	n	o
060	061	062	063	064	065	066	067	160	161	162	163	164	165	166	167
0	1	2	3	4	5	6	7	p	q	r	s	t	u	v	w
070	071	072	073	074	075	076	077	170	171	172	173	174	175	176	177
8	9	:	;	<	=	>	?	x	y	z	{	|	}	~	

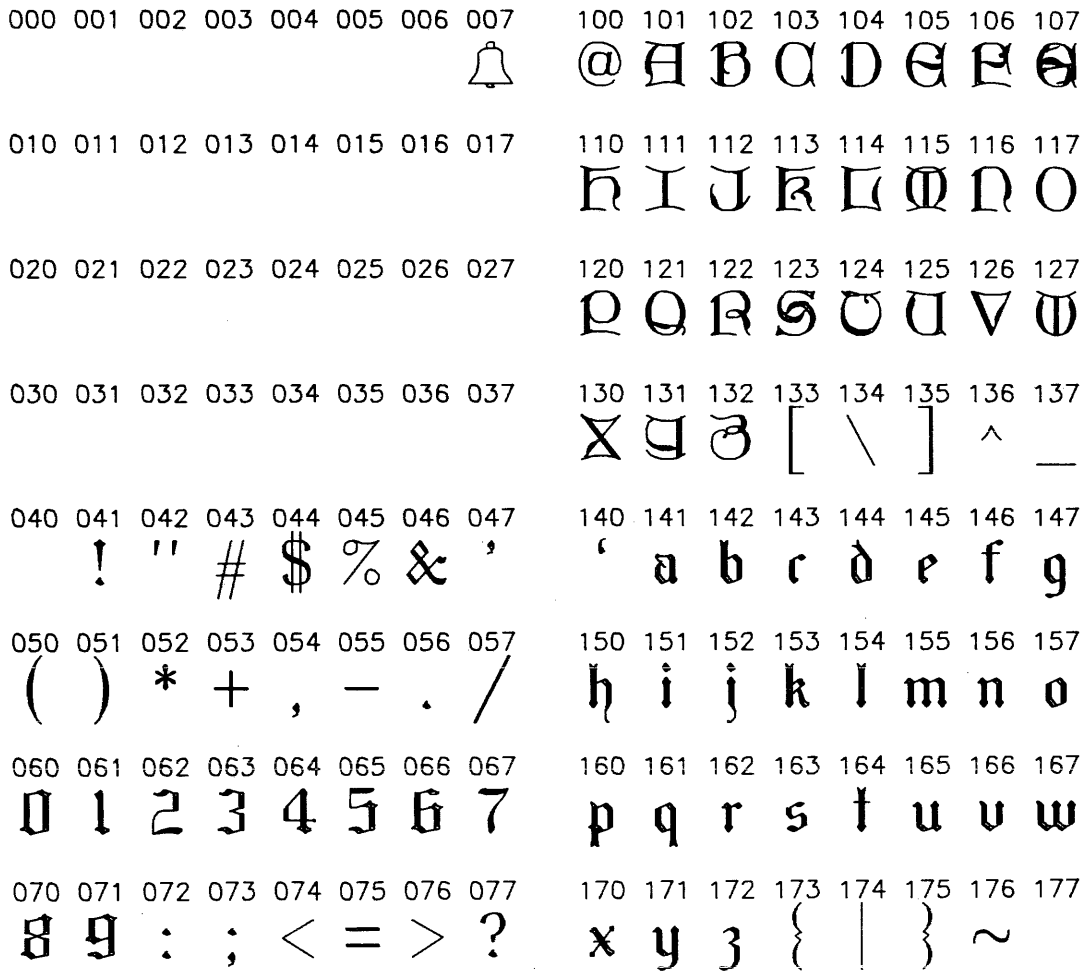
The Quick Brown Fox Jumps over the Lazy Dog.

gct_gothic_german_

gct_gothic_italian_

Name: gct_gothic_italian_

The gct_gothic_italian_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.



The Quick Brown Fox Jumps over the Lazy Dog.

gct_gothic_italian_

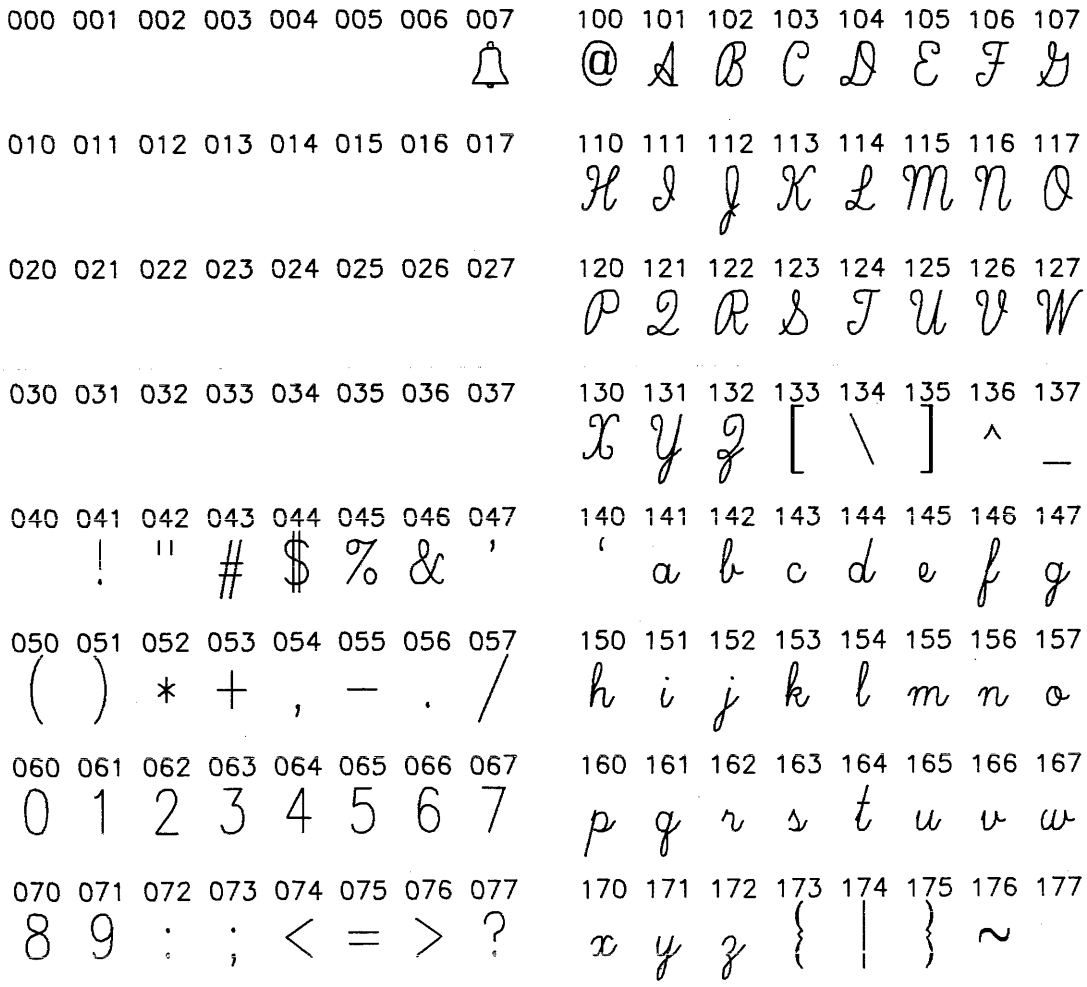
gct_simplex_roman_

Name: gct_simplex_roman_

The gct_simplex_roman_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.

Name: gct_simplex_script_

The gct_simplex_script_ character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.



gct_simplex_script_

gct_triplex_italic_

Name: gct_triplex_italic_

The gct_triplex_italic_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.

000	001	002	003	004	005	006	007
							🔔
010	011	012	013	014	015	016	017
020	021	022	023	024	025	026	027
030	031	032	033	034	035	036	037
040	041	042	043	044	045	046	047
	!	"	#	$	%	&	'
050	051	052	053	054	055	056	057
(	)	*	+	,	-	.	/
060	061	062	063	064	065	066	067
0	1	2	3	4	5	6	7
070	071	072	073	074	075	076	077
8	9	:	;	<	=	>	?

100	101	102	103	104	105	106	107
@	A	B	C	D	E	F	G
110	111	112	113	114	115	116	117
H	I	J	K	L	M	N	O
120	121	122	123	124	125	126	127
P	Q	R	S	T	U	V	W
130	131	132	133	134	135	136	137
X	Y	Z	[	\	]	^	_
140	141	142	143	144	145	146	147
`	a	b	c	d	e	f	g
150	151	152	153	154	155	156	157
h	i	j	k	l	m	n	o
160	161	162	163	164	165	166	167
p	q	r	s	t	u	v	w
170	171	172	173	174	175	176	177
x	y	z	{	|	}	~	

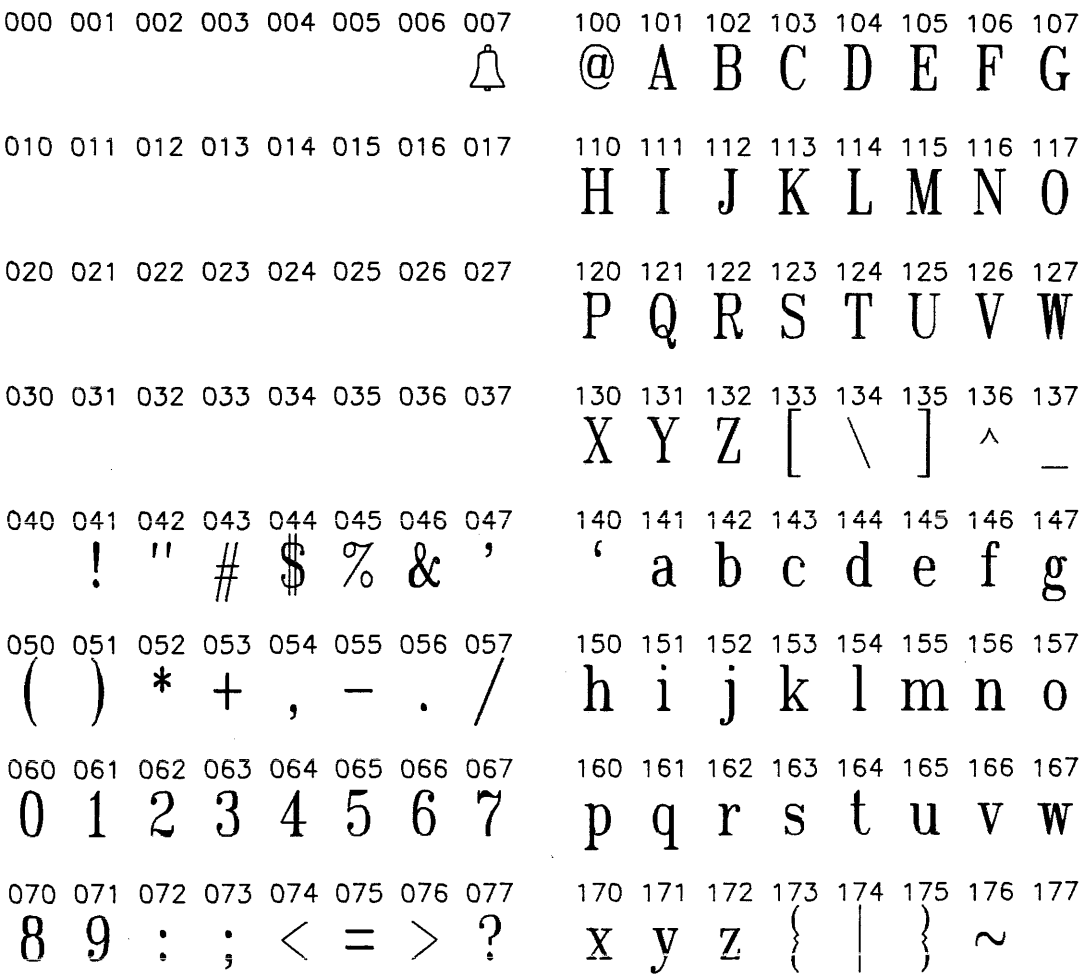
The Quick Brown Fox Jumps over the Lazy Dog.

gct_triplex_italic_

gct_triplex_roman_

Name: gct_triplex_roman_

The gct_triplex_roman_ graphic character set is displayed below. This character set is extracted from the standard Hershey Occidental character set.



SECTION 8

GRAPHIC INCLUDE FILES

The include files (found in the default translator search path) described here are generally useful to the MGS user. This documentation is provided to encourage the graphics system user to use common coding practices, and to avoid errors and duplication of effort in the production of system required declarations.

Name: gc_entry_dcls.incl.pl1

contains declarations for every entry point in graphic_compiler_, including declarations for every combination of multiple names by which an entry can be called.

Name: gct_char_names.incl.pl1

contains declarations of the specific character names required by compile_gct to specify characters in graphic character tables.

Name: gch_entry_dcls.incl.pl1

contains declarations for every entry point in graphic_chars_.

Name: gm_entry_dcls.incl.pl1

contains declarations for every entry point in graphic_manipulator_, including declarations for every combination of multiple names by which an entry can be called.

Name: gmc_entry_dcls.incl.pl1

contains declarations for every entry point in graphic_macros_, including declarations for every combination of multiple names by which an entry can be called.

Name: go_entry_dcls.incl.pl1

contains declarations for every entry point in graphic_operator_, including declarations for every combination of multiple names by which an entry can be called.

Name: graphic_char_dcl.incl.pl1

contains the declaration of the format of graphic character tables.

Name: graphic_code_dcl.incl.pl1

contains mnemonically named declarations of variables that have special meaning to programs manipulating MSGC. It includes:

1. a declaration of every defined graphic effector;

2. the default values for all graphic modes;

3. a table describing the expected lengths of graphic effectors; (Note: subroutine graphic_element_lengths_ is, in the general case, the only foolproof way to extract this information.)

4. miscellaneous small character constants with other meanings in MSGC.

Name: graphic_device_table.incl.pl1

contains the declaration of the format of a GDT. In addition, it contains declarations for mnemonically-named constants containing the values understood by a GDT (for example, prepare for text, expansion, etc.) other than those belonging to specific effectors in MSGC.

Name: graphic_enames.incl.pl1

contains declarations of constant arrays that may be indexed by numbers returned by the graphics system to yield printable representations of those values. (In effect, it is the inverse of the contents of include file graphic_etypes.) It contains printable values for:

1. element codes, as returned, for example, by graphic_manipulator_$examine_type;

2. intensity, sensitivity, blink, and line_type values;

3. text alignments;

4. graphic input device codes.

Name: graphic_etypes.incl.pl1

contains declarations for mnemonically named "constant" variables that contain numeric values that have meaning to the graphics system. It contains:

1. graphic effector values, as returned, for example, by graphic_manipulator_$examine_type;

2. values used as arguments to specific graphic modes, for example, line_types by name, intensity, blink, and sensitivity values;

3. variables containing text alignment codes;

4. merge codes for graphic_manipulator_ entries "put_struc" and "get_struc";

5. variables containing codes for graphic input devices.

Name: graphic_input_formats.incl.pl1

contains declarations describing the structure of well formed graphic input messages for the "what," "where," and "which" types of query. It also contains the declarations of the characters that follow the "query" character to cause specific types of input.

Name: graphic_terminal_errors.incl.pl1

contains the declaration of an array of error codes. Status codes returned by intelligent graphics terminals may be used to index into this array to extract the standard Multics error code corresponding to the error. The declarations of the standard Multics error codes used are also included.

Name: gui_entry_dcls.incl.pl1

contains declarations for every entry point in gui_.

Name: plot_entry_dcls.incl.pl1

contains declarations for every entry point in plot_. In addition, it contains declarations for mnemonically named "constant" variables that contain numeric values that may be used to select the various modes and functions of plot_ at several of its entry points.

APPENDIX A

SUBROUTINE ABBREVIATIONS

The following list identifies all of the graphics subroutines and their associated entry points that have software supported abbreviations, but are not otherwise documented in this manual.

subroutine/entry point name	abbreviation
calcomp_compatible_subrs_	ccs_
graphic_compiler_	gc_
display	d
display_append	da
display_name	dn
display_name_append	dna
load	l
load_name	la
graphic_macros_	gmc_
graphic_manipulator_	gm_
create_array	carray
create_color	ccolor
create_data	cdata
create_list	clist
create_mode	cmode
create_position	cpos
create_rotation	crot
create_scale	cscale
create_text	ctext
examine_color	ecolor
examine_data	edata
examine_list	elist
examine_mapping	emap
examine_mode	emode
examine_position	epos
examine_symbol	esym
examine_symtab	esymtab
examine_text	etext
examine_type	etype
find_structure	fstruc
graphic_operator_	go_

INDEX

MISCELLANEOUS

A

B

C

D

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 68)
MULTICS GRAPHICS SYSTEM
ADDENDUM A

ORDER NO. AS40-01A

DATED AUGUST 1981

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ **DATE** _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 496, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

26377, 5C1279, Printed in U.S.A.

AS40-01