To: Distribution

From: Lee J. Scheffler

Date: November 14, 1974

Subject: New Multics Graphics Package

Attached are preliminary versions of Sections I and
II of what is intended to become the Graphics Users'
Supplement (GUS) to the MPM. These sections describe
the design goals and structure of a new Multics
Graphics System.

Several questions important to the Multics development
community not covered inside are answered below.

- This graphics system runs entirely in the user
  ring, depending only on the existence of raw
  input and output modes.

- A working version of it is available on the MIT
  Multics, through the SIPB. See me for details;
  I would like to know who is using it.

- There is not yet a working version of it for use
  on the Phoenix Multics.

- Command and subroutine writeups are available.
  A Users' Guide (to become Section III of the
  GUS) should be available about mid-December.

- There is not yet a write-around module for the
  old graphics system. Unless there is a great
  demand for one, one will probably never be written.

- There is a conversion program for converting old
  graphic data bases into new format.

Questions, comments to Scheffler.MPLS on MIT or
Phoenix Multics.

HONEYWELL INFORMATION SYSTEMS INCORPORATED

DATA SYSTEMS OPERATIONS

The Multics Programmers' Manual
Graphics Users' Supplement
Sections I and II only

Revision 2

Date: 06/11/74

I.   Overview of the Multics General Graphics System

The Multics Graphics System provides a general-purpose termi-
nal-independent interface through which user or application pro-
grams can create, edit, store, display and animate graphic con-
structs. There are three major objectives behind the design of
this interface.

1. It should be easy and natural to write a graphic program. The
    set of graphic primitives and operations available should be
    sufficiently flexible and general that a user need not "bend
    over backwards" to program common operations.

    Primitives are provided for manipulating a structured picture
    description composed of lines, points, screen modes, rota-
    tions, translations (position shifts) and scalings, in three
    dimensions. Primitives are also provided for displaying parts
    of such a graphic structure, for animating an already dis-
    played structure, for obtaining graphic input, and for con-
    trolling special terminal functions (such as screen erase).
    These primitives are suitable for direct use by a knowledge-
    able programmer.

2. It should be amenable to a wide range of applications, while
    retaining efficiency and ease of use. The motivations behind
    this goal are to avoid creating and maintaining a multiplicity
    of systems, each oriented towards a separate application or
    terminal; and to avoid the necessity of graphics users having
    to master the idiosyncracies of entirely separate systems.

    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 for application modules to provide simpler or more ap-
    plication-oriented interfaces. Efficiency is enhanced by pro-
    viding 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.

3. It should be highly terminal-independent. That is, as far as
    possible, a graphic program written for one type of graphic
    terminal should be operable on another graphic terminal of
    similar capabilities without modification. A wide variety of

graphic terminal types may be connected to Multics, and this
terminal mix changes with time. By making the graphic system
interface independent of any particular terminal type, we
avoid several problems that arise from terminal-dependent pro-
gramming.

This has several desirable consequences:

a) User fragmentation is prevented. Users are not isolated by
   the particular type of graphic terminals they use, but can
   make use of graphic programs developed on different termi-
   nals by other users.

b) Terminal immobility is avoided. Users are not restricted
   by their programs to using only particular terminal types,
   but can make use of whatever graphic terminal is available.
   More importantly, graphic subsystems written for specific
   terminals can be easily transferred to new and better ter-
   minal types.

c) Software duplication is greatly reduced. Most graphics
   utility routines need be written only once to be usable
   with most or all of the graphic terminal types on the sys-
   tem.

Terminal-independence is achieved in the Multics Graphics Sys-
tem in the following way. The programming interface of the
Multics Graphics System incorporates the union of most useful
features of existing terminals and is extensible to allow the
addition of new features as graphic terminal capabilities
evolve. A user tailors his program to use the features of the
terminal types he intends to 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, the user has a reasonable guarantee
that his graphic programs will produce a recognizable picture
on most any type of graphic terminal connected to Multics. Of
course, not all graphic programs will operate equally well at
any type of graphic terminal (e.g., animation is not possible
on a storage-tube terminal.)

II. Structure of the System

The Multics Graphics System is organized into two distinct func-
tional parts: the terminal-independent or "central" graphics
system, and the terminal interfaces, as shown in Figure 1.

User and application programs communicate almost exclusively with
the central graphics system. The central graphics system manipu-
lates a database containing a structured representation of a
graphic picture. When a user or application program decides to
display a portion of the graphic structure, the structure is
transformed into a character string representation known as "Mul-
tics Standard Graphics Code," which is suitable for transmission
through a Multics I/O stream. 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 Stand-
ard Graphics Code, consulting a tabular description of the capa-
bilities of the graphic terminal currently being used to decide
if any operations need to be performed on the graphics code be-
fore it is sent to the graphic terminal. Typical operations in-
clude discarding structure information for static terminals and
redundant information for intelligent terminals, performing rota-
tions and scalings for terminals lacking these features, attempt-
ing compromise operations where necessary, and translating the
standard code 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 Multics
Standard Graphics Code which is interpreted by the central graph-
ics system and returned to user or application programs as return
arguments from a request for input.

This particular organization was chosen for reasons of generality
and efficiency in performing many operations common to graphic
subsystems. The structured database allows graphic pictures to
be represented naturally (e.g., a screw as part of 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 Multics
Standard Graphics Code 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 stand-
ard background scene on any terminal type. Also, in many cases,
the terminal-dependent graphics code produced by a particular
terminal interface can also be stored and played back to that
particular terminal type at negligible computational overhead.
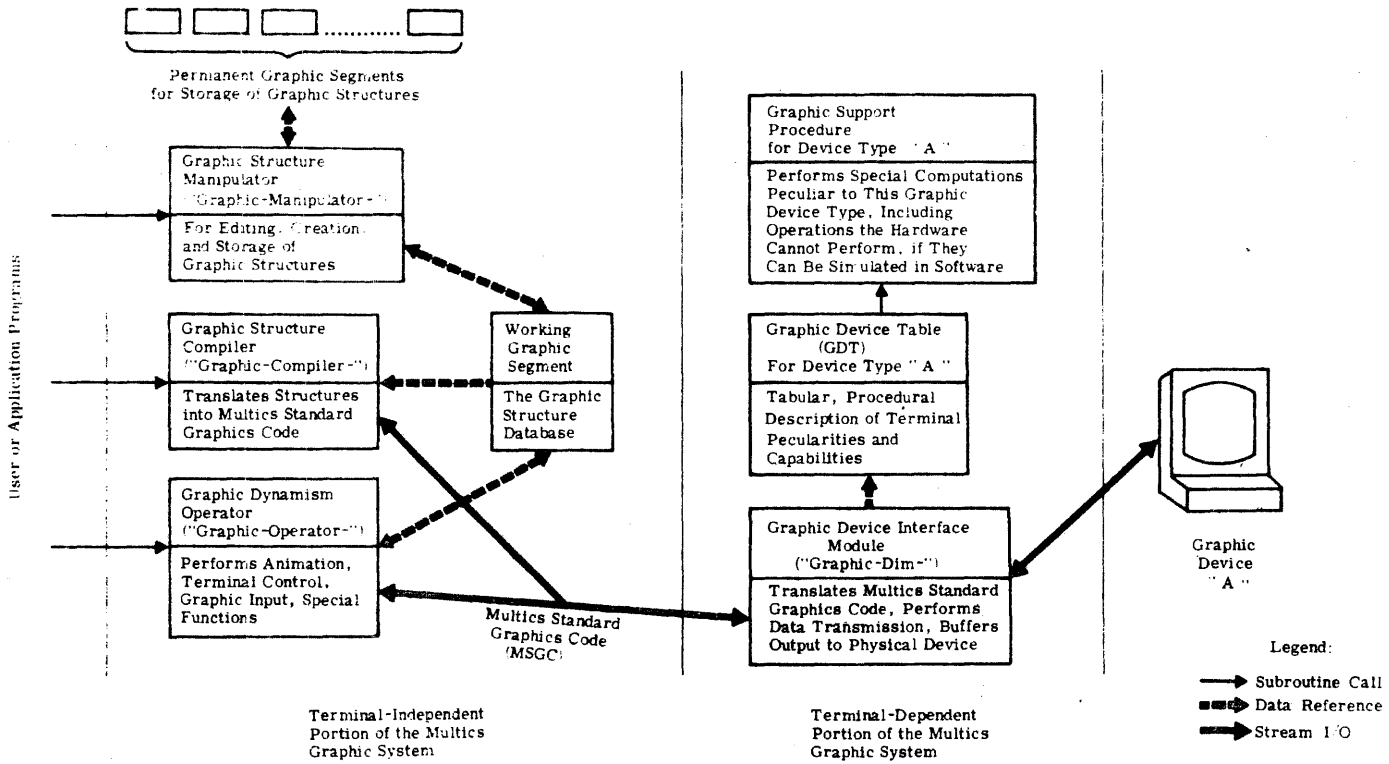
Figure 1
Functional parts of the Multics Graphics System

(c) Copyright 1974, Honeywell Information Systems Inc.

User or Application Programs

Permanent Graphic Segments
for Storage of Graphic Structures

Graphic Structure
Manipulator
("Graphic-Manipulator-")

For Editing, Creation,
and Storage of
Graphic Structures

Graphic Structure
Compiler
("Graphic-Compiler-")

Translates Structures
into Multics Standard
Graphics Code

Graphic Dynamism
Operator
("Graphic-Operator-")

Performs Animation,
Terminal Control,
Graphic Input, Special
Functions

Working
Graphic
Segment

The Graphic
Structure
Database

Multics Standard
Graphics Code
(MSGC)

Terminal-Independent
Portion of the Multics
Graphic System

Graphic Support
Procedure
for Device Type "A"

Performs Special Computations
Peculiar to This Graphic
Device Type, Including
Operations the Hardware
Cannot Perform, if They
Can Be Simulated in Software

Graphic Device Table
(GDT)
For Device Type "A"

Tabular, Procedural
Description of Terminal
Pecularities and
Capabilities

Graphic Device Interface
Module
("Graphic-Dim-")

Translates Multics Standard
Graphics Code, Performs
Data Transmission, Buffers
Output to Physical Device

Terminal-Dependent
Portion of the Multics
Graphic System

Graphic
Device
"A"

Legend:

→ Subroutine Call
⇢ Data Reference
► Stream I/O

The tabular description of a graphic terminal capabilities and
pecularities allows new terminal types to be added to the system,
with a minimum of overhead. And the ability to specify system-
or user-supplied utility routines to aid graphic code translation
promotes terminal independence, and provides a handle for extend-
ing the basic capabilities of the Multics Graphics System.

II.1 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
Multics Graphics System deals instead with tree-structured de-
scriptions 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
of note. First, it allows for fairly natural representation of
graphic data. Recognizable objects (automobiles, doors, houses)
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 sub-structures 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 structures: ter-
minal elements and non-terminal elements. Terminal elements rep-
resent simple graphic operations most often interpreted directly
by graphic terminal hardware. These include screen positioning,
line and point drawing operations, screen modes (such as blink-
ing, intensity, dotted, dashed or solid lines, and sensitivity to
a light pen), and coordinate rotations and scalings in three di-
mensions. Non-terminal elements are lists which impose ordering
on the elements they contain. Levels of structure are represent-
ed by including non-terminal elements within other non-terminal
elements. Figure 2 depicts a graphic structure describing a
simple house. At the highest level (House-Display), the House is
placed in proper screen position by a setpoint, given full screen
intensity, and made sensitive to light pen "hits". At the next
level, the house is composed of a House-Outline, a Door, and
Windows. The House-Outline itself is made up of a Roof, a
Foundation, a Chimney and an antenna. The single Window design
is shared in two places in the Windows substructure.

Each graphic element in a Multics segment representing a graphic
structure is uniquely identified within the segment by a node
number which is used to reference that element within the struc-
ture and in later operations. Non-terminal elements are simply
linear lists of the node numbers of all the elements they con-
tain.

Figure 2
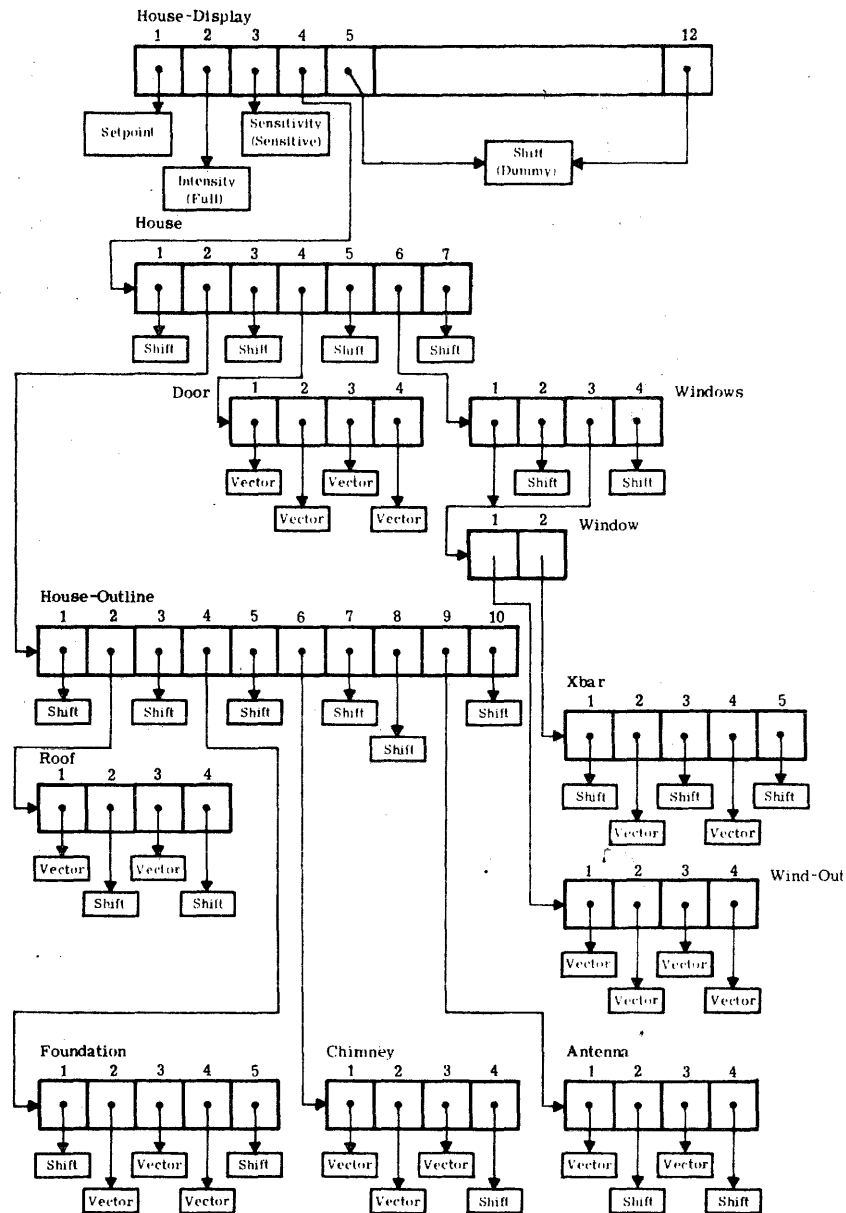A Typical Graphic Structure Organization

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

        element_type   (argument1, argument2, ... argument$n$)

is used to convey the essential meaning of each element. The ac-
tual semantics of subroutine calls for creating and editing
graphic elements is described in the section on Graphic Structure
Manipulation.

II.1.1 Non-Terminal Graphic Elements

There are three types of non-terminal graphic elements used in structuring a graphic picture. They are:

         lists
         arrays
         symbols

Lists

Lists are the most straightforward and most often used non-terminal graphic elements. A list is specified by

     list (element1, element2, ... element$\underline{n}$)

where element n is the node number 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 non-terminal elements. However, circular or recursive lists (those that contain themselves or are part of a chain of list containment that eventually leads back to themselves) have undefined meaning and are therefore illegal. Note that 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

An array is structually equivalent to a list, but causes all information about the structure of its elements to be lost when the structure is compiled into Multics Standard Graphic Code. 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 which do not support dynamic graphics and thus have no use for structural information, and for those substructures which the user does not intend to alter dynamically (e.g., background scenes).

Symbols

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

         symbol (element, name)

where element is the node number of the terminal or non-terminal graphic element, and name is the node number of a terminal text

element (see next section) containing the text of the name of the
element.   Symbols serve several purposes, the primary one being
to uniquely identify graphic constructs in a mnemonic way,   that
may be moved between several Multics segments.

II.1.2 Terminal Graphic Elements

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

There are four categories of terminal elements in the Multics
Graphic System.

        positional elements
        modal elements
        mapping elements
        miscellaneous elements

Positional Elements

Positional elements affect the screen position (in three dimen-
sions) 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. Positions are computed within a virtual
screen of 1024 x 1024 x 1024 points, with the point (0,0,0) cor-
responding 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 (-512e0 < x,y,z < 511e0).

The coordinate system is a right-handed Cartesian coordinate sys-
tem, 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 rela-
tive. Absolute positional elements force the graphic cursor to a
specific point in the virtual screen. Relative positional ele-
ments move the graphic cursor to a new position relative to its
current position. The elements are:

        setposition, setpoint - absolute positioning
        vector, shift, point - relative positioning

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 posi-
   tion 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 posi-
tion, although in general, shared substructures should have a net
relative shift of (0,0,0) (i.e., the sum of the relative posi-
tioning elements in a shared list or array should normally add up
to (0,0,0)).

Modal Elements

Modal elements produce no effects on the screen of themselves,
but affect the properties of successive graphic elements in de-
fined manners. The appearance of a modal element in a list over-
rides 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)

intensity (value)

   This element affects the brightness of succeeding graphic ele-
   ments 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 zero is
   defined as solid (the default), type one as dashed, type two
   as dotted. The remaining type codes are reserved for future
   expansion.

steady/blinking (value)

   This element causes succeeding graphic elements to be dis-
   played steadily (the default), or to blink.

insensitive/sensitive (value)

   This element causes succeeding graphic elements to be sensi-
   tive 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 dis-
   played in the color specified by the intensities of the three
   primary colors in the additive color spectrum.

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

1) When a modal element occurs in a list, it effects all succes-
   sive 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 graphic environment (mode settings, rotations, scal-
   ings, 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 dis-
   carded at the end of the substructure and the modes are re-
   stored 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 modes.)

Mapping Elements

Mapping elements cause no visible effect by themselves, but af-
fect 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 modulus 360 de-
   grees.

scaling (*x, *y, *z)

   This element causes succeeding graphic elements to undergo
   scaling in the three separate directions defined by the sta-
   tionary coordinate system.   Scalings may be negative to pro-
   duce mirror images.

clipping (left, right, bottom, top, back, front)

   This element causes all succeeding normally visible graphic
   elements to be clipped (become invisible) if they fall outside
   of a rectangular solid defined by its parameters. (The parame-
   ters are relative displacements from the current screen posi-
   tion of each of six planes defining a rectangular solid.) If a
   graphic element straddles the boundary, only the part within
   the rectangular solid will be visible.

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

1) Successive mapping elements override previous mapping elements
   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 cur-
   rently active in the parent list.

3) Mapping elements in a sub-list have no effect on the mappings
   in a parent list.

Because mappings are non-commutative vector operations, the order
of application of mapping elements to constructs in a list is im-
portant.   A scene that is first scaled and then rotated will in
general appear different than one that is first rotated and then
scaled. Within a list, scaling is performed first, then rotation
then clipping.   This order may be overridden by using several
levels of structure to achieve the desired order of application.
The modes "closest to the object" (on the lowest structural
level) are "more 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 graphic
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 posi-
tions 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

      data block - for storing user data within the graphic struc-
      ture, or extension of the basic capabilities of the Multics
      Graphic System

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)

      string is a text string of any length (although in general it
      will be smaller than the text line length of most graphic ter-
      minals).

      alignment is a number from 1 to 6 which specifies that the
      text string is to be aligned in one of 6 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	Lower left
5	Lower center
6	Lower right

The string is subject to active screen modes, but not to map-
pings. However, the initial position of the string is subject to
mappings.

datablock

The datablock graphic element allows arbitrary user-defined bit
strings representing user data to be stored as part of a graphic
structure. The data is passed to the graphic terminal just as
any graphic effector is, which makes it possible for a user with
special applications to use a datablock to contain terminal-de-
pendent information or commands. This provides a straightforward
and powerful facility for extending the basic capabilities of the
Multics Graphic System by allowing user program-to-graphic termi-
nal conventions.

The datablock is defined by:

    datablock (user_data)

    where user_data is any string of any length. There are no
    system-defined type codes for marking the user_data as repre-
    senting integers, characters, etc., although the user may in-
    clude his own such description as part of his data.

Datablocks have no system-defined effect on other graphic ele-
ments in the same list or in subordinate graphic structures.

II.2   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 a pro-
gram called the Graphic Manipulator (graphic_manipulator_) 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 numbers, valid
only within the current WGS. When a new graphic element is crea-
ted, the node number of the created element is returned to the
user program as a sort of "receipt". This node number is used in
all later references to this element. Lists of graphic elements
are simply PL/I- or FORTRAN-like arrays of node numbers 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 Permanent Graphic Segments (PGS) anywhere in the
Multics storage hierarchy.

Node numbers are used for graphic structure creation and editing
for efficiency.   The node number of an element presently corre-
sponds to its word offset in the WGS.   (This correspondence is
not guaranteed to remain valid.) Names are used for permanent
storage because they are more mnemonic, and because the operation
of copying a graphic structure into a PGS performs an implicit
storage compaction and garbage collection function, thereby
changing the node numbers of most graphic elements copied.

See the writeup of graphic_manipulator_ for the details of the
various graphic structure manipulation entry points.

II.3   Graphic Structure Compilation

When a user has created and edited a graphic structure to his
satisfaction, he can then produce a character string representa-
tion of this structure for transmission through the Multics I/O
system by using the Graphic Compiler (graphic_compiler_).   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 number or name of its top-level list.   The compiler
transforms this structure into an equivalent representation in
Multics Standard Graphics Code, a standard intermediate form
which is terminal-independent. This code is written over the I/O
stream "graphic_output". This stream may be attached to a termi-
nal interface, thereby directing the code to a particular graphic
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 commonly necessary operations on the re-
mote 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).

II.3.1   Multics Standard Graphics Code

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 stream.   It allows
the representation of structural information useful to intelli-
gent terminals and redundant information necessary to display
shared substructures on non-intelligent terminals.

Multics Standard Graphics Code is terminal-independent in two
senses: it contains no specification of any particular terminal
type, and it contains all information necessary to produce graph-
ics on all supported terminals.

The MSGC for a graphic structure is produced by a left-most tree
walk of the structure in the current working graphic segment.
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 ... arg$\underline{n}$

Levels of list structure are represented by nestings of paired
parentheses, and include a list/array indicator and a node number
followed by the list elements, in order:

        (list_indicator node_no element1 element2 ... element$\underline{n}$)

The node number is retained to aid intelligent terminals in con-
structing their internal representations of graphic structures,
and to allow identification of shared substructures.   Other
graphic operations (animation, input, etc.) are also represented
by a single ASCII character operator code followed by arguments:

        operator_code arg1 arg2 ...arg$\underline{n}$

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.

There are four formats for transmission of argument values in
Multics Standard Graphics Code, depicted in the following pic-
tures:

1 Character

```
bits
  0   1   2   3           8
┌─────────┬───┬─────────────┐
│         │   │             │
│  0   0  │ 1 │             │
│         │   │             │
└─────────┴───┴─────────────┘
                └─────┬─────┘
              6 Bit Unsigned
              Binary Integer
```
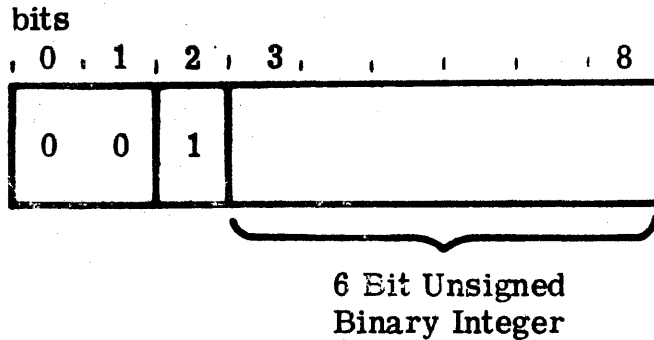
(First)

Single Precision Integer (SPI) format is used for transmission of
small non-negative values from 0 to 63.

2 Characters   Char 1   Char 2

```
Bits  0   1   2   3           8   0   1   2   3           8
    ┌─────────┬───┬─────────────┬─────────┬───┬─────────────┐
    │         │   │             │         │   │             │
    │  0   0  │ 1 │             │  0   0  │ 1 │             │
    │         │   │             │         │   │             │
    └─────────┴───┴─────────────┴─────────┴───┴─────────────┘
                  └──────┬──────┘             └──────┬──────┘
                    High-order                   Low-order
                     6 Bits                        6 Bits
              └──────────────────────┬──────────────────────┘
                          12 Bit 2's Complement
                          Binary Integer
```
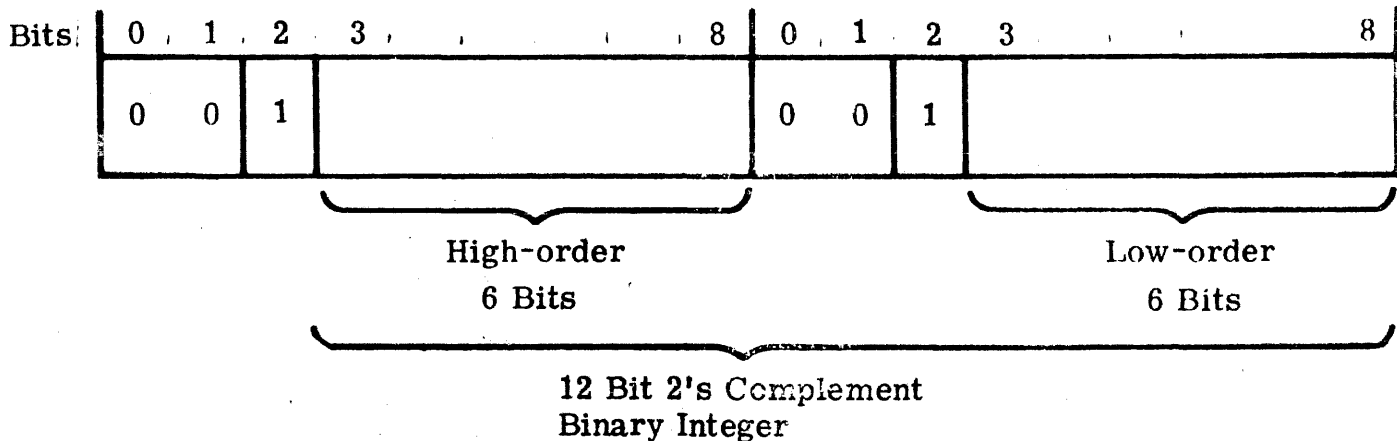
(Second)

Double Precision Integer (DPI) format is used for signed integers
ranging from -2048 to 2047.

```
3 Characters         Char 1                      Char 2                      Char 3
Bits  | 0  1  2  3              8 | 0  1  2  3              8 | 0  1  2  3              8 |
      |─────────────────────────|─────────────────────────|─────────────────────────|
      | 0  0  1  .               | 0  0  1                  | 0  0  1                  |
      |─────────────────────────|─────────────────────────|─────────────────────────|
             └────┬────┘                  └────┬────┘                  └──────┬──────┘
              High-order                  Low-order                  6 Bit Unsigned
               6 Bits                      6 Bits                    Binary Fraction
                                                                    (Implicit Binary
                     └──────────┬──────────┘                        Point to Left of
                        12 Bit 2's Complement                        First Bit)
                         Binary Integer
                              └──────────────────────┬──────────────────────┘
                                            18 Bit 2's Complement
                                            Fixed Point Real Binary
                                                  Number
```
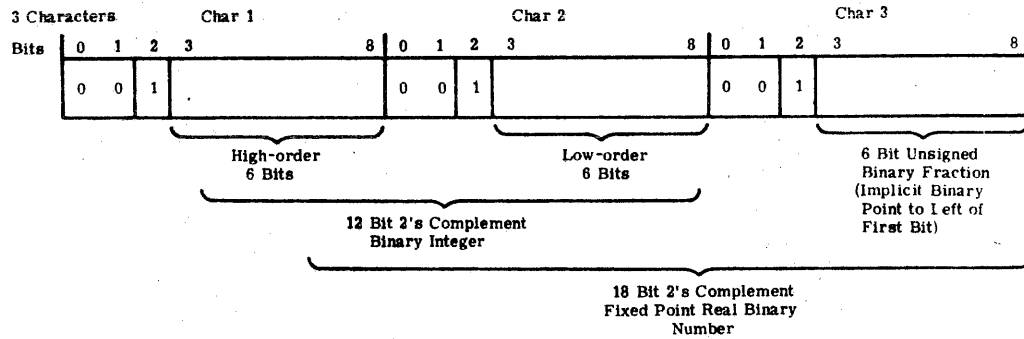
Scaled Fixed Point (SCL) format is used for numbers with frac-
tional parts. It has the same range as the DPI format, but is
accurate to fractional parts of 1/64.

```
3 Characters         Char 1                      Char 2                      Char 3
Bits  | 0  1  2  3              8 | 0  1  2  3              8 | 0  1  2  3              8 |
      |─────────────────────────|─────────────────────────|─────────────────────────|
      | 0  0  1                  | 0  0  1                  | 0  0  1                  |
      |─────────────────────────|─────────────────────────|─────────────────────────|
             └────┬────┘                  └────┬────┘                  └────┬────┘
              High-order                   Middle                    Low-order
               6 Bits                      6 Bits                     6 Bits
                     └───────────────────────┬───────────────────────┘
                                      18 Bit Unsigned
                                      Binary Integer
```
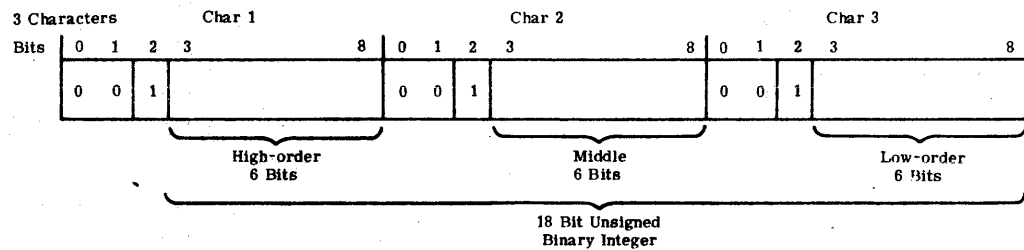
Unique ID (UID) format is used to transmit 18-bit node numbers.

Following are the character codes and argument list formats for
the operators in Multics Standard Graphics Code. (See Section
II.4 on Dynamic Graphic Operations for descriptions of the func-
tions of the animation, input and terminal control operators.)

Positional Operators

```
                        \
setposition ("0") !
setpoint    ("1") !
vector      ("2") }  xpos      ypos      zpos
shift       ("3") !  (DPI)     (DPI)     (DPI)
point       ("4") !
                     /
```

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

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, in the SCL format.

```
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, in DPI for-
    mat.

```
clip        ("7")    right   left    top    bottom    front    back
                     (DPI)   (DPI)   (DPI)  (DPI)     (DPI)    (DPI)
```

    where the arguments are the relative displacements of six
    planes forming a rectangular solid "clipping box", all in DPI
    format.

Modal Operators

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

    where value is a number from 0 (invisible) to 7 (fully visi-
    ble) in SPI format.

line_type ("9")   value
                     (SPI)

   where value is one of the following:

            0 - solid line
            1 - dashed line
            2 - dotted line
            3-63 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 repre-
   senting full intensity.

Miscellaneous Operators

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

   where alignment controls the positioning of the character
   string relative to the current screen position.   Values for
   the alignment are described earlier in this section.

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

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

   where length is the number of data bits to follow and 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 struc_type is either 0 (list) or 1 (array) and node_no
is the unique ID associated with the list or array.

```
node_end      (")")      (no arguments)
```

```
symbol        ("=")      length      name
                         (OPI)       (ASCII)
```

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

```
reference     ("%")      node_no
                         (UID)
```

where node_no is the unique ID of a node already resident in
terminal memory, and is used in successive references to
shared substructures. Users wishing to construct and output
their own graphic code should refrain from using this opera-
tor, as it will limit their graphic code to intelligent termi-
nals. This operator is normally inserted into the graphic
stream at run time by the graphic device interface module.

Animation Operators

```
increment    ("<")      node_no     times    delay     template_node
                        (UID)       (OPI)    (SCL)
```

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

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

delay is the amount of time the terminal is to delay before
each increment.

template_node is the graphic element containing the relative
increment to be performed, and includes the element code in
its own format.

synchronize   (".")   (no arguments)

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

     node_no is the unique ID of the list or array node being al-
     tered.

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

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

Input and User Interaction

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

     input_type is the type of graphic input desired (1 = where, 2
     = which, 3 = what)

     input_device is the graphic input device to be used to gener-
     ate the indicated input.

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

control   ("*")   node_no
                         (UID)

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

pause   ("t")   (no arguments)

Terminal Control

erase   ("-")   (no arguments)

display   ("+")   node_no
                                (UID)

   node_no is the unique ID of the top_level list node to be dis-
played.

delete   ("/")   node_no

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

II.4   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 Multics and those in
the graphic terminal memory should be isomorphic (structurally
equivalent). In other words, there are no provisions for the
user or his 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 or application program in Multics.

There are several reasons for adopting the philosophy. First, it
allows a simple and well-defined interface to a graphic 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 re-
quests information. Second, terminal-resident programming is
simplified, reducing the amount of memory required at the termi-
nal. Finally, the problems inherent in maintaining separate
copies of a database (in this case a graphic structure) are elim-
inated. The nature of the dynamic graphic operators is such that
both Multics-resident and terminal-resident structures are iden-
tical before and after each operation.

Dynamic graphic operations are initiated by calls to entry points
in the Graphic Dynamism Operator (graphic_operator_). These
entry points emit characters in Multics Standard Graphics Code to
cause a terminal to perform the desired operations, and return to
the user program any information returned by the terminal. The
details of these entry points may be found in the module writeup
on graphic_operator_.

II.4.1   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.

There are three dynamic operators which accomplish movement:

        increment
        synchronize
        alter

increment

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

    increment node_no no_times delay template

    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 succes-
    sive increments

    template is a positional mode or mapping element whose argu-
    ments are the increments to each of the parameters in the ele-
    ment being incremented.

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

synchronize

Because several constructs may be moving simultaneously, there is
a need to be able to coordinate movements to allow events to be
properly sequenced (e.g., balls bouncing off each other).   The
synchronize primitive has no arguments. It simply commands the
graphic terminal to complete all operations received before the
synchronize before beginning any subsequently received operators.

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

    list_id is the node number of a list already resident in ter-
    minal memory

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

    new_element is the node number of the new element, which must
    also be resident in terminal memory.

The indicated list is updated both in the working graphic segment
in Multics, and in the terminal-resident structure.

II.4.2   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 clas-
ses of graphic input built into the Multics Graphics System:

where (coordinate postion) - the user indicates one position in
the stationary x,y,z coordinate system.

which (structure specification) - the user indicates a particular
subtree of a displayed graphic structure.

what (new structure) - the user creates some new graphic struc-
tures at his terminal and returns them to Multics.

These three input types are all initiated with a single "query"
dynamic operator of the form

        query input_type device_type

    input_type is a code indicating which of the three inputs are
    desired.

    device_type indicates the graphic input device from which the
    input is desired. (It may also indicate that the user is to
    be given his 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 Multics.
This kind of operation is implemented by the "control" dynamic
operator:

        control device_type node_no

    device_type is the same as in the "query" operator.

node_no is the unique ID of a positional modal or mapping ele-
ment 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 orien-
tation (rotation) of a scene under control of a trackball. Upon
completion of a control interaction, the structure resident in
Multics is updated to mirror the changes made.

Pause

Occasionally it is desirable to allow a user to proceed step by
step through a sequence of displays at his own speed. If there
is no new computation required of Multics between steps, there is
no reason for an interaction with Multics between steps. The
"pause" operation causes the terminal to delay processing of sub-
sequently received graphic data until the user indicates that he
is ready to proceed. In this way, all graphic operations for
such a session can be pre-loaded into the terminal and operated
with a minimum of Multics interaction.

II.4.3 Terminal Control

There are various housekeeping functions that need to be per-
formed when dealing with graphic terminals.

>   screen control
>   terminal memory management
>   communications control and error handling

Screen Control

Screen control consists of erasing all graphics currently dis-
played on the screen, and selectively displaying graphic struc-
tures already resident in terminal memory. The former is accomp-
lished with the "erase" operator:

>   erase (no arguments)

The latter function is accomplished with the "display" operator

>   display node_no

node_no is the unique ID of the top-level of a graphic structure
already resident in terminal memory that is to be displayed.

Memory Management

Memory management deals with loading new graphic structures into
terminal memory and deleting structures that are no longer
needed. Loading is accomplished implicitly simply by sending a
new graphic structure to the terminal. The "delete" operator al-
lows individual structures to be deleted, presumably freeing
space in terminal memory.

>   delete node_no

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

Communications Control and Error Handling

There are several problems that fall under the heading of commun-
ications control. It is necessary to distinguish character
strings representing graphic structures and operations from nor-
mal text. Since most intelligent terminals are mini-computers
with limited memory, there will often be limits on the speed with
which the terminal can process incoming graphics and the size of
terminal communications buffers. And because fairly complex

structures all being transmitted, some high-level protocol for
discovering and reporting errors to Multics is necessary.

For dynamic terminals, two ASCII control characters are defined
to have the following meanings:

>   DC1 (octal 021)   Enter graphic mode
>   DC2 (octal 022)   Enter text mode

DC1 indicates that all subsequent characters should be interp-
reted as representing graphic structures and operators.

DC2 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 Graphic Device Table describing a terminal con-
tains an indication of the size of the terminal's input buffer.
The strategy is to dispatch no more than this number of charac-
ters 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 fol-
lowed by a right parenthesis and a new-line character (")<NL>")

Character	Format	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.)

Character	Format	Represents
2	ASCII	character code of graphic operator in which error occurred
3-5	UID	unique ID of top-level node in graphic structure in which error was detected
6	SPI	depth of error in list structure
7	SPI	list index of top level list element
8	SPI	list index of next level list element
9 on	SPI	list indices of succeeding elements until done

If the error code returned is 0, then the next buffer of charac-
ters is output to the terminal. Otherwise, the error is reflec-
ted back to the user program and the as yet unsent characters are
destroyed.

Many graphic operators must be sent immediately to the ter-
minal, because they require terminal response before more graphic
data is generated. However, it is desirable to keep the frequen-
cy of status request interactions to a minimum because half-du-
plex communications protocols insert rather substantial delays.
Control over when the Multics output buffer is sent is exercised
in two ways. First, in the Graphic Device Table describing a
terminal, one can specify for each graphic operator in Multics
Standard Graphics Code whether or not the buffer must be sent.
Normally, the buffer must be sent only on query and control oper-
ators, where input from the terminal is necessary. Secondly, an
entry point in the Graphic Operator (graphic_operator_) 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.