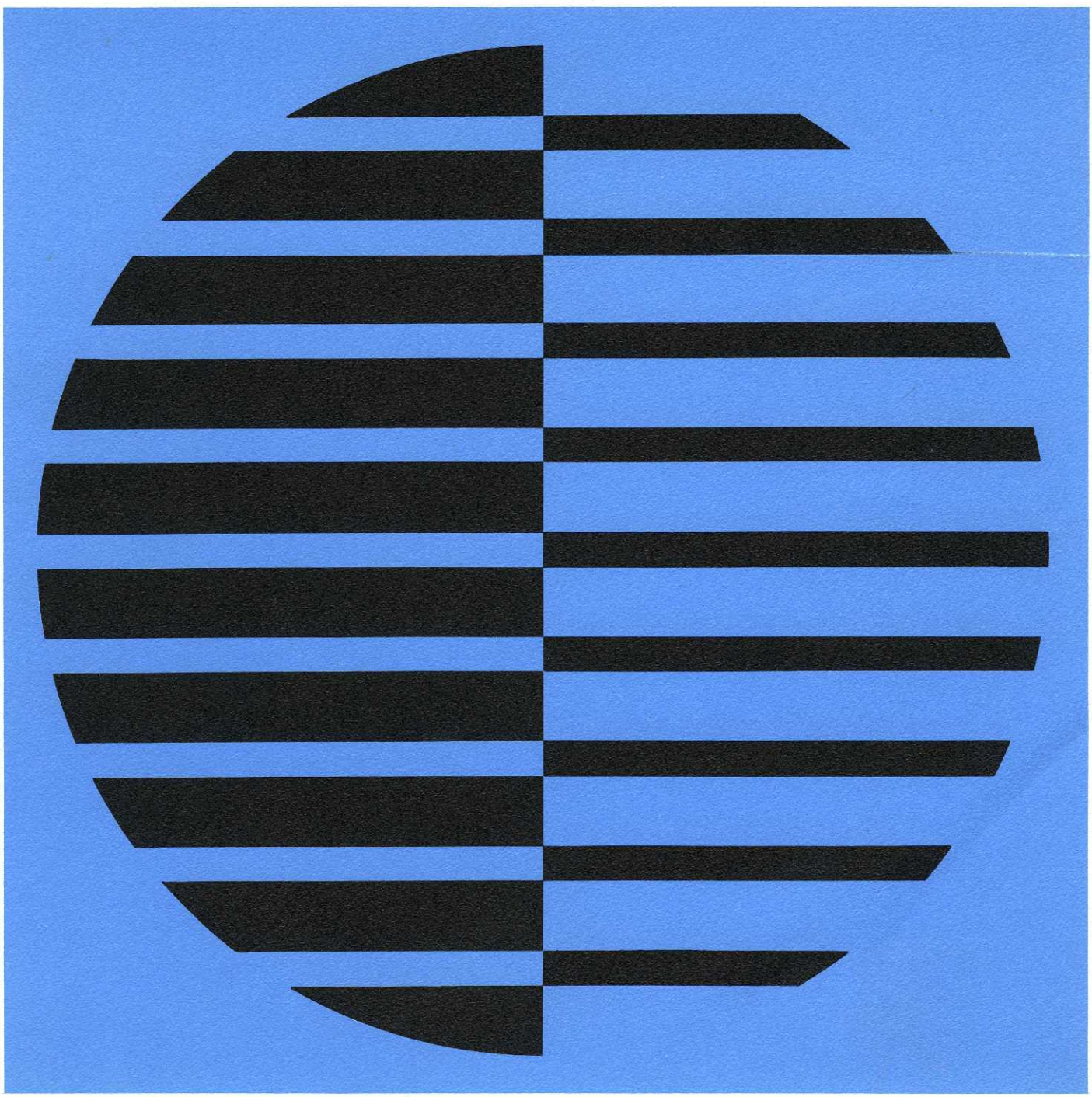
CONTROL DATA®
6000 SERIES COMPUTER SYSTEMS
274 INTERACTIVE GRAPHICS SYSTEM
Reference Manual



CONTROL DATA
CORPORATION

CONTROL DATA®
6000 COMPUTER SYSTEMS
274 INTERACTIVE GRAPHICS SYSTEM

REFERENCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A	Initial release. This manual describes the IGS software designed for use with the 6000 series
(2-1-71)	SCOPE 3.3 operating system.
B	Final release. Revisions made by authority of PCO CF 1120 on various pages; revision lines
(4-1-71)	omitted to avoid confusion produced by large-scale relocation of text within sections. Revision B
	describes the final released form of the Version 2 software.
C	Change to page 1-6 made by authority of PCO CF 1204 to correct an error regarding the reserva-
(5-15-71)	tion of central memory space for graphics control points.
D	Changes made to pages 2-21, 5-7, 6-8, 6-9, 6-24, 6-26, 6-67, 6-72, 6-74, and B-1 by authority of
(10-15-71)	PCO CF 1254 for minor editorial corrections and correction of errors in descriptions of mask
	processing.

Publication No.
17303600

Additional copies of this manual may be obtained from the nearest Control Data Corporation sales office.

6000 Series 274 Interactive Graphics System Reference Manual

Printed in the United States of America

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Systems Publications
215 Moffett Park Drive
Sunnyvale, California 94086

or use Comment Sheet in the back of this manual.

PREFACE

This manual is a general programming and reference guide for the CONTROL DATA® 6000 series 274 Interactive Graphics System, Version 2. It contains a summary of software operation and external characteristics. A background knowledge of Control Data 6000 series software and hardware is needed to properly use this manual.

Version 2 of Interactive Graphics functions in a 6000 series SCOPE 3.3 operating system environment.

This manual is organized for quick-reference to programming information. Detailed background material is provided in each subdivision of the system software; programming information is isolated within the section that describes the software involved in that particular system function.

The first section contains a general outline of software operation and appropriate hardware information. At the end of Section 6 is a subsection containing summaries and calling formats for the individual graphics routines of the system which are accessible to an applications programmer. For more information related to the system's software, see the following Control Data publications:

Title	Publication Number
6000 Series SCOPE 3.3 Reference Manual	60305200
6000 Series SCOPE 3.3 Operating Guide	60306400
6000 Series Systems Reference Manual	60100000
6000 Series Interactive Graphics System General Information Manual	44616700
6000 Series EXPORT/IMPORT High-Speed and 274 Interactive Graphics System Operator's Guide	17303100
6000 Series 274 Interactive Graphics System User's Guide	44629300
1700 Series 1744 Digigraphics Controller Reference/Customer Engineering Manual	60283300
General Description of the 274 Display Console and the 1744 Controller	17223000

CONTENTS

APPENDICES

FIGURES

TABLES

INTRODUCTION

The CONTROL DATA® 6000 series 274 Interactive Graphics System is designed to permit real-time use of a large computer by a graphics console operator – without significantly degrading the capabilities of the machine.

Interactive Graphics accomplishes this by using a small machine, a Control Data 1700 series computer, to control the basic functions of the graphics hardware; the system uses the 6000 series computer only to handle more difficult manipulations and to do the mathematical work required by the applications programmer or the console user.

The Digigraphics 274 Display Consoles connected to the smaller computer permit the user to create, display, store, retrieve, and modify any graphics forms necessary for the active analysis of a problem – as well as giving him a means of entering data directly. These graphic forms can then be expanded or changed by the user in a real-time environment through his application program and the Interactive Graphics System.

The system can process the types of programs usually run in batch-processing mode, but it eliminates the user waiting time of that mode and provides a user with much greater flexibility in his use of the computer than batch-processing permits.

The system handles problems that:

- Can best be repeated in symbolic, graphic, or geometric form (such as schematics, diagrams, layouts, lattice structures, geologic cross-sections, and paths of motion)
- Can best be described using mathematical functions (dynamic analyses)
- Require human intervention (such as transcribing data for digital processing, empirical problem-solving, and geographic studies)

MAJOR FEATURES

Interactive Graphics includes these unique features:

- Graphics programming is done only on the 6000 series computer – the 1700 series computer software operates without programmer intervention.

- Graphics programs can be written in standard FORTRAN Run or Extended, independent of display hardware characteristics. At installation time, either Run or Extended is specified; this manual assumes that Extended (FTN) is used. If an installation chooses Run, however, each FTN card must be replaced by a RUN control card in IGS jobs.
- Data files can be tailored to fit the specific needs of an application programmer's job.
- Batch and graphic processing is performed concurrently; both types of jobs can be entered through the 1700 series computer, as well as at the 6000 series site.
- Interactive Graphics can simultaneously service 24 independent graphics consoles through four 1700 series computers.
- The 1700 series computers are not dedicated to graphics work, but can perform other functions – even when graphics jobs are in the system.

HARDWARE ELEMENTS

The hardware configuration of the 6000 series Interactive Graphics System is very versatile; the system can be configured for either remote, local, or intermediate operation. Figure 1-1 shows a typical Interactive Graphics hardware system; a fully expanded system would include the following Control Data equipment:

- Any standard 6000 series hardware configuration, including a 6673 or 6674 Data Set Controller
- Four 1700 series computers
- Six 274 Graphics Consoles per 1700
- One 853 Disk Storage Unit per 1700
- One card reader per 1700†
- One card punch per 1700†
- One printer per 1700
- One 1713 Teletypewriter per 1700

A paper tape station (used with a 1711 Teletypewriter) is also optional at the 1700 series site.

†For simultaneous graphics and remote batch processing, a buffered card reader (1725 Card Reader Controller and 405 Card Reader) or buffered card reader/punch (1728 Card Reader/Punch Controller and 430 Card Reader/Punch) must be used. A 415 Card Punch and Controller may be used with the 405 Card Reader.

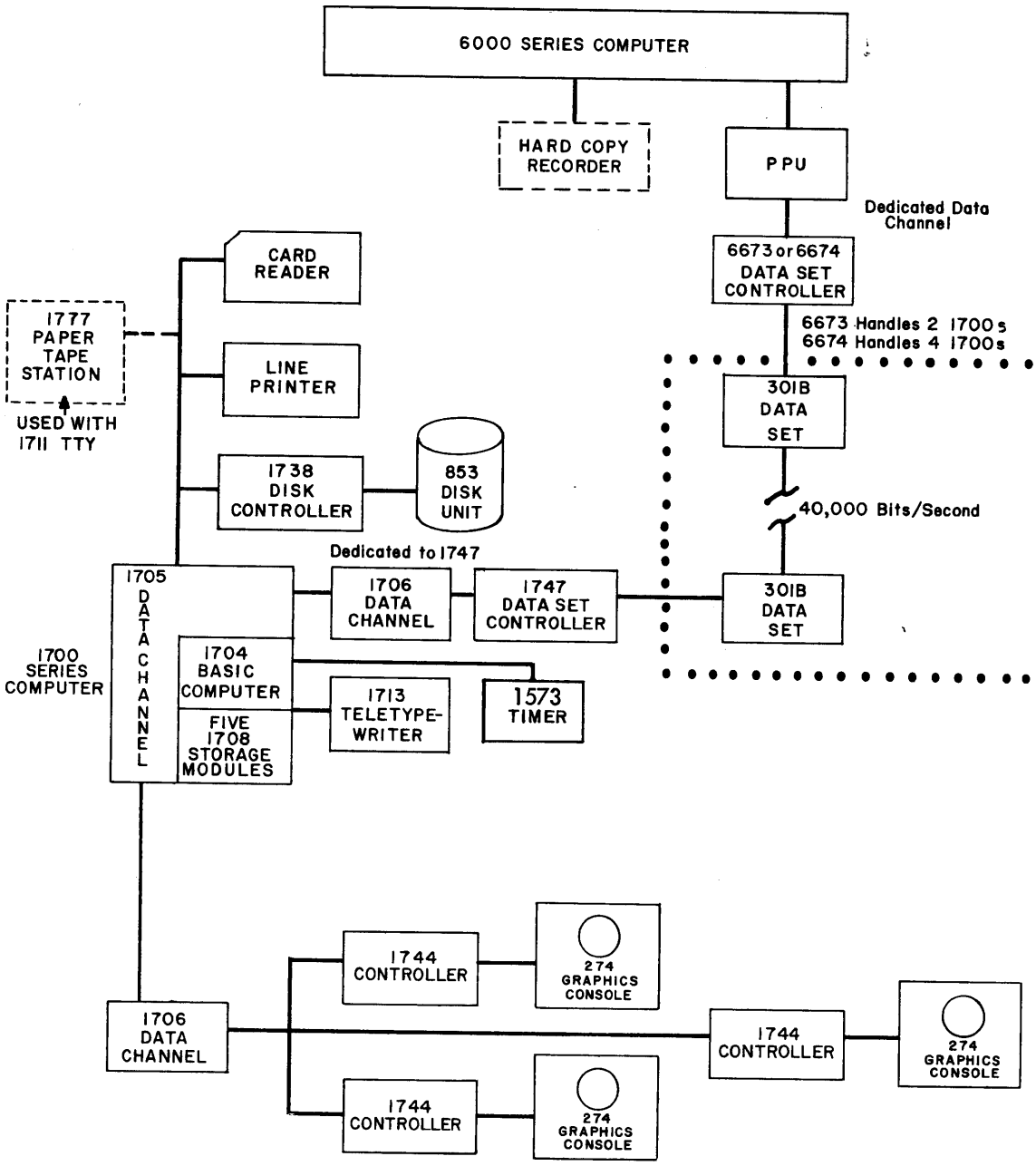


Figure 1-1. Typical Interactive Graphics Hardware System

Figure 1-1 is a generalized remote configuration diagram for the system. In a local configuration, the equipment within the dotted lines is replaced by a connecting line less than 200 feet long; in an intermediate configuration, the same equipment is replaced by two special equipment units, joined by a line up to 1,000 feet long.† If an SC1700 Computer system is used, the basic computer in the diagram is a 1774 and 1772 units are used for storage; a 1773 Storage Access Bus and a 1775 Data Channel replace the 1705.

GENERAL SOFTWARE OPERATION

Interactive Graphics software operates as two separate but communicating groups of routines – one in the 6000 series computer, the other in each of the 1700 series computers. Figure 1-2 shows the relationship between the groups.

6000 SERIES SOFTWARE

The 6000 series portion of the system software includes:

- The 6000 series SCOPE 3.3 operating system, with several added graphics features††
- A standard FORTRAN Run or FORTRAN Extended compiler
- The 6000 Basic Graphics Package, for actual graphics programming
- The Scheduler, to provide time-sharing for graphics programs
- An EXPORT High-Speed (HS) program, for communication between the 6000 series and the 1700 series computers

SCOPE FEATURES

Because graphics programs require a real-time environment, they cannot be allowed to compete with batch jobs for the use of central memory control points; instead, one or two of SCOPE's control points are dedicated to graphics use. The number can be varied as needed by the 6000 series' operator, depending on the ratio of the graphics job load to the batch-processing load. If graphics programs are not being run, all control points can be made available for batch use.

The real-time requirements of graphics jobs also prohibit them from competing with batch jobs for central memory storage space. Therefore, each graphics control point has

† Line speed for this configuration is memory speed (1.1 seconds per 12-bit word) minus a small factor for line length.

†† The operating system interface and some of the features vary with the version of SCOPE used; the revision level notes of this manual indicate the operating system interface.

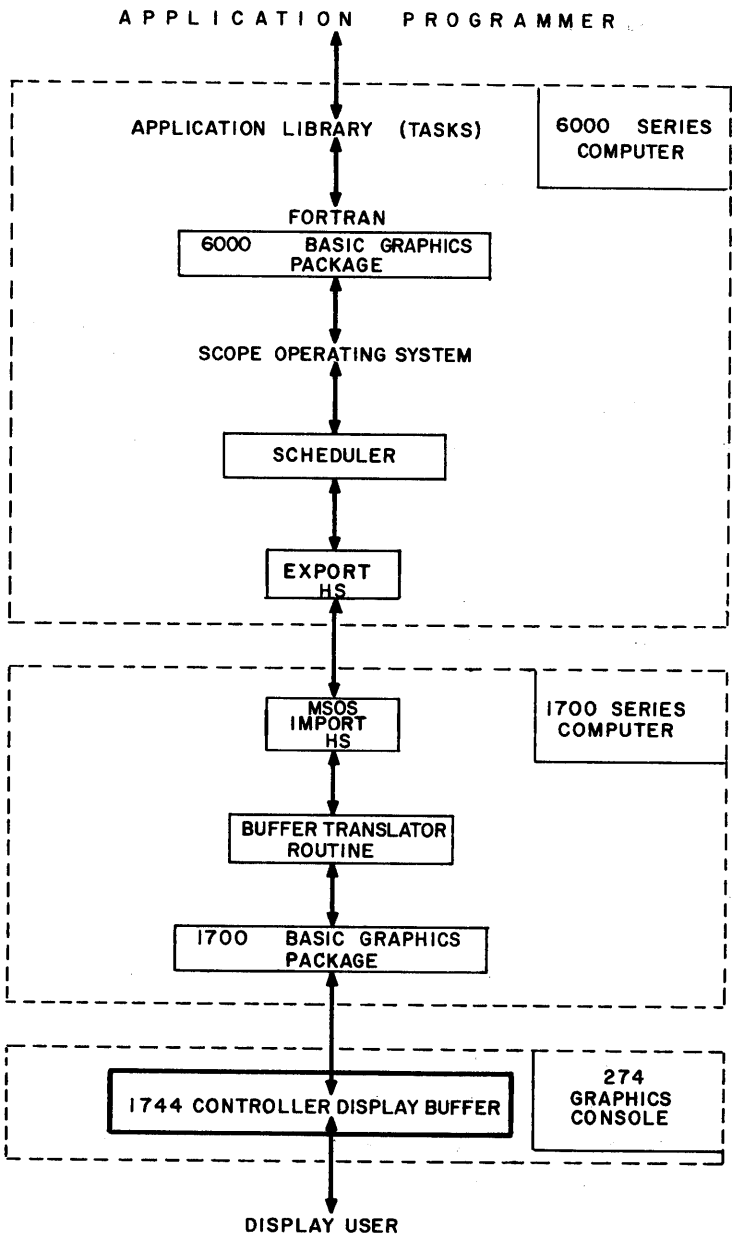


Figure 1-2. Software Interactions

the ability to roll out lower priority batch jobs to obtain central memory space. This rollout is performed automatically, without operator intervention, to minimize response time at the graphics control point.

The SCOPE library for Interactive Graphics includes three utility routines for the use of graphics programs:

- The task file creator, AEFILE
- The task file dump routine, AEDUMP
- The random-access file creator, AELOAD

Graphics programs are written as a series of overlays, each performing a task. The AEFILE routine places these overlays in mass storage as a random-access file with an index that can keep track of hundreds of overlays. The applications programmer can make additions to and deletions from this file; a task is located within the file and placed in central memory when it is needed (location and loading is performed by 6000 Basic Graphics Package routines).

The AEDUMP routine is used to remove unneeded information from the task file and to rewrite the file in a form that can be stored outside the system.

The AELOAD utility program is used to restructure the file produced by AEDUMP into a form that can be used as a task file.

PROGRAMMING FEATURES

The 6000 Basic Graphics Package allows the user to write programs in FORTRAN without worrying about the maintenance of a display-oriented graphics data base or the mechanics of communicating with the display. The Package contains an expandable library of subroutines that provide efficient and complete access to the graphics hardware (and two-way communication with it) without limiting application types or data structures. The Package is designed so that the programmer's only concern is communication with the graphics console operator and the computational requirements of the application; he is not aware of the internal functions of the package, since there are no system-specified data areas that must be manipulated.

REAL-TIME MULTIPROGRAMMING

If several graphics programs are in the system at the same time, a form of time-sharing must be used so that each graphics console user believes that his program has sole use of the 6000 series computer.

Graphics programs share their use of the 6000 series central memory through a mechanism controlled by the Scheduler. The Scheduler looks at the programs waiting for execution in its graphics input queue, the graphics input request of currently executing programs, and at the programs themselves. The Scheduler then decides whether to roll out a program and roll in a new one from the input queue, or to roll in an old program that was rolled out while waiting for an input request to be serviced.

The Scheduler determines how long each program should be allowed to remain at a graphics control point on the basis of the central and peripheral processor time the program used when it last resided at a control point; this gives short graphics programs priority over longer ones. A lower limit, chosen by each installation, is imposed on the Scheduler's determination of a program's permitted resident time.

EXPORT HS FEATURES

EXPORT HS performs all data communication between the 1700 series computers and the 6000 series computer. The Interactive Graphics version of EXPORT HS provides the same services for remote batch programs as the non-graphics version, and has several additional features:

- EXPORT HS is called to a control point by the 6000 series computer's operator.
- EXPORT HS monitors the resident time of each graphics program and calls the Scheduler into a peripheral processor when a program's permitted resident time has elapsed.
- EXPORT HS periodically scans each graphics control point for an input or output request and automatically transfers graphics output data to its own output buffers for transmission to the proper 1700.
- Graphics data from a 1700 series computer is queued by EXPORT HS when it is received for later use by an application program (remote batch data is turned over to SCOPE for processing, as in the non-graphics version).
- EXPORT HS overlays are stored in central memory resident, rather than in mass storage, to reduce the overhead time of data communication processing.
- EXPORT HS processes remote batch data and graphics data concurrently.

1700 SERIES SOFTWARE

The 1700 portion of the Interactive Graphics software consists of three groups of routines:

- An MSOS IMPORT HS program, to handle all communications between the 6000 series computer and the 1700 series computer.

- The buffer translator
- The 1700 Basic Graphics Package

MSOS IMPORT HS FEATURES

The Interactive Graphics version of MSOS IMPORT HS has all of the data communication features of the non-graphics version and interfaces with drivers to run a line printer, card reader and card punch, or card reader/punch.

DATA TRANSLATION

The buffer translator reformats the graphics data buffers received by MSOS IMPORT HS from the 6000 series computer into calls to the 1700 Basic Graphics Package. In this manner, data from the 6000 Basic Graphics Package is translated into a display-oriented data base. The buffer translator also formats data from the graphics consoles for transmission to the 6000 series computer.

1700 GRAPHICS ROUTINES

The 1700 software includes a group of graphics routines called the 1700 Basic Graphics Package. These routines act like drivers for the graphics consoles, sending display information to the 1744 Controllers according to instructions received from the 6000 Basic Graphics Package calls. The 1700 routines also process interrupts and data from the graphics consoles, queueing the information until the program in the 6000 series computer requests it. The application programmer does not use the 1700 Package routines when coding a job.

ADDITION OF SOFTWARE FUNCTIONS

Additional 1700 functions can be incorporated in the Interactive Graphics System without altering the existing software; the 1700 series computer can be used to drive remote devices for specific applications, without hardware modification (other than the addition of memory).

GENERAL PROCESS CHART

The general process chart in Figure 1-3 follows a user's program through the Interactive Graphics System and shows the relationships between the hardware and software at various stages in the program's processing.

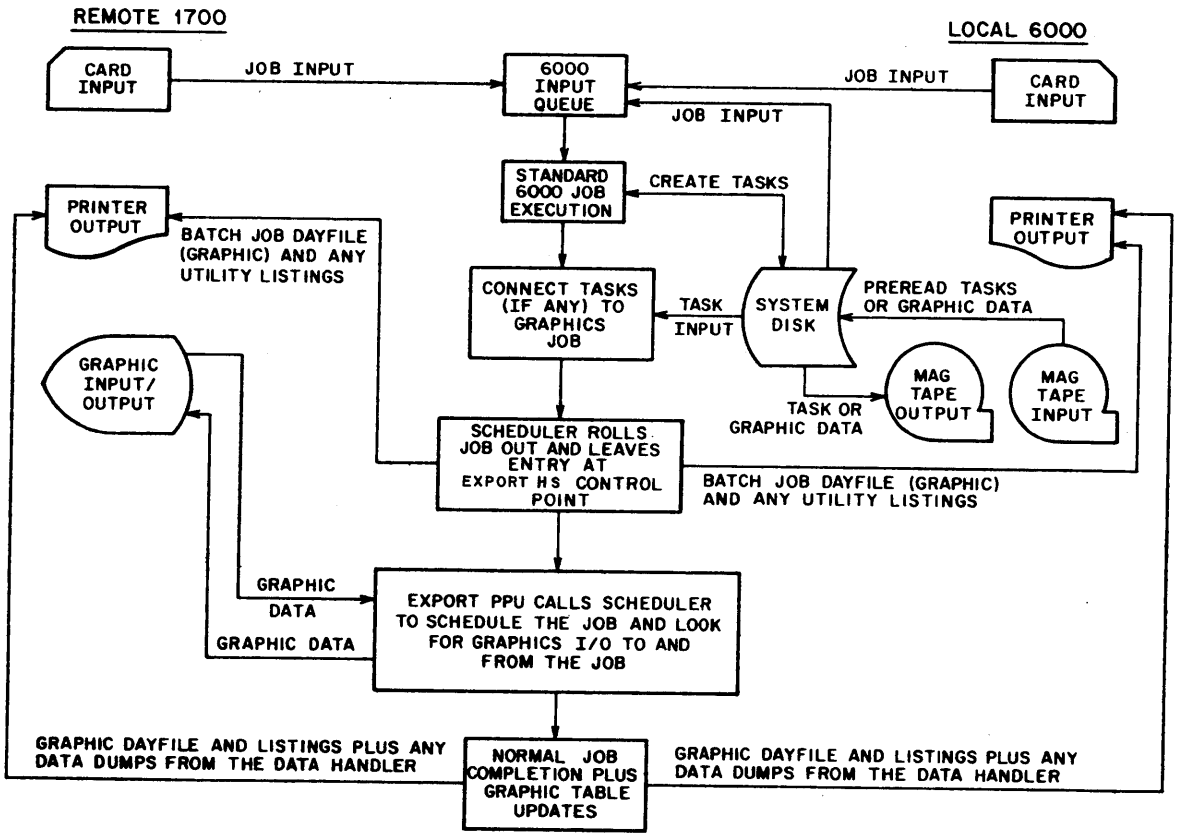


Figure 1-3. General Process Chart

SYSTEM PROCESS CHART

The system process chart in Figure 1-4 also follows a program through the system, and shows in more detail the interaction of the parts of the software with the hardware. This chart is a schematic of the flow of data through the system during graphics program processing.

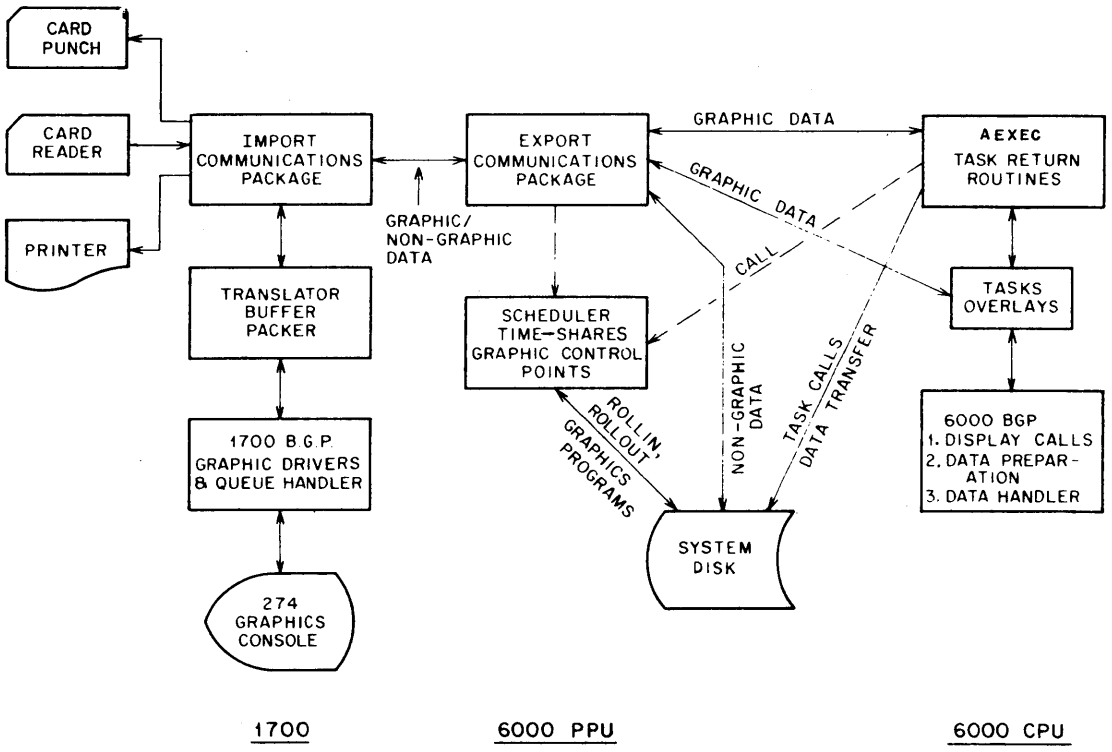


Figure 1-4. System Process Chart

INPUT/OUTPUT AND GENERAL PROCESSING 2

CONTROL POINTS

Two to three of the control points provided by the standard SCOPE operating system are reserved in the modified form of the system used by Interactive Graphics. One to two can be designated as graphics control points by the installation and are then used exclusively for graphics programs, although the system makes them available on command for batch use. The third control point is reserved for the use of EXPORT High-Speed.

SCHEDULER

The Scheduler is a PPU program; when called into its peripheral processor, it provides dynamic scheduling and time-sharing for graphics jobs running at graphics control points. Graphics jobs are queued; if a job is on console 1, another job for console number 1 can be read in before the current graphics job on the console is aborted and detached from the console.

Initially, a graphics job enters the Interactive Graphics System as a batch job and is assigned to a batch-processing control point for execution (batch job scheduling is done by SCOPE, not by the Scheduler). At some point in its execution as a batch job, the graphics job calls the graphics reformatter (see Application Executive, Section 6).

SCHEDULER SUBROUTINES

A set of Scheduler subprograms puts a graphics job into the graphics rolled out format so that the job can be scheduled at a graphics control point.

After a program's initial call to the Scheduler/reformatter, the Scheduler drops the CPU and clears the program's EXPORT HS communication word. The Scheduler then rolls out the program, its control point field, dayfile, and all of its associated local file name table entries. Permanent and COMMON files are assigned to EXPORT's control point while the job is rolled out, and the program is then assigned an initial priority and placed in a special graphics input queue. When the program is rolled back in, the file name table entries are replaced to reflect the new control point number.

SCHEDULING OF GRAPHICS CONTROL POINTS

The execution priority of each program in the Scheduler's graphics input queue is determined by the program's current field length and whether or not it has any unsatisfied graphics input requests; short programs with no unsatisfied requests have the highest priorities.

GRAPHICS CONTROL POINTS

INITIALIZATION

The operator of the 6000 series computer assigns one or two graphics control points manually, using the procedure given in the Interactive Graphics System Operating Guide (see Preface). The type-ins that he uses enter the control point number in a table (EXPORT HS must be assigned first); this table identifies which control points EXPORT must service for graphics processing.

STRUCTURE

Figure 2-1 shows the general structure of one graphics control point area. The uses of the various words and subdivisions are described in other sections of this manual. Minimum field length of a control point area is about 7000 octal words.

NUMBER

In order to best use the dedicated space available plus the idle time when graphics tasks are being rolled in or out to mass storage, two graphics control points should be used. While one control point is accessing the mass storage device, the other can be executing and/or performing input and output to the graphics console.

SIZE

The field length of graphics jobs should be kept to a minimum (10 to 20K). The suggested method of application programming and the random-access loading of tasks permit division of large code modules into smaller ones which may be rapidly accessed sequentially. The data handler provides the capability to maintain data without large in-core arrays or application concern with disk input or output.

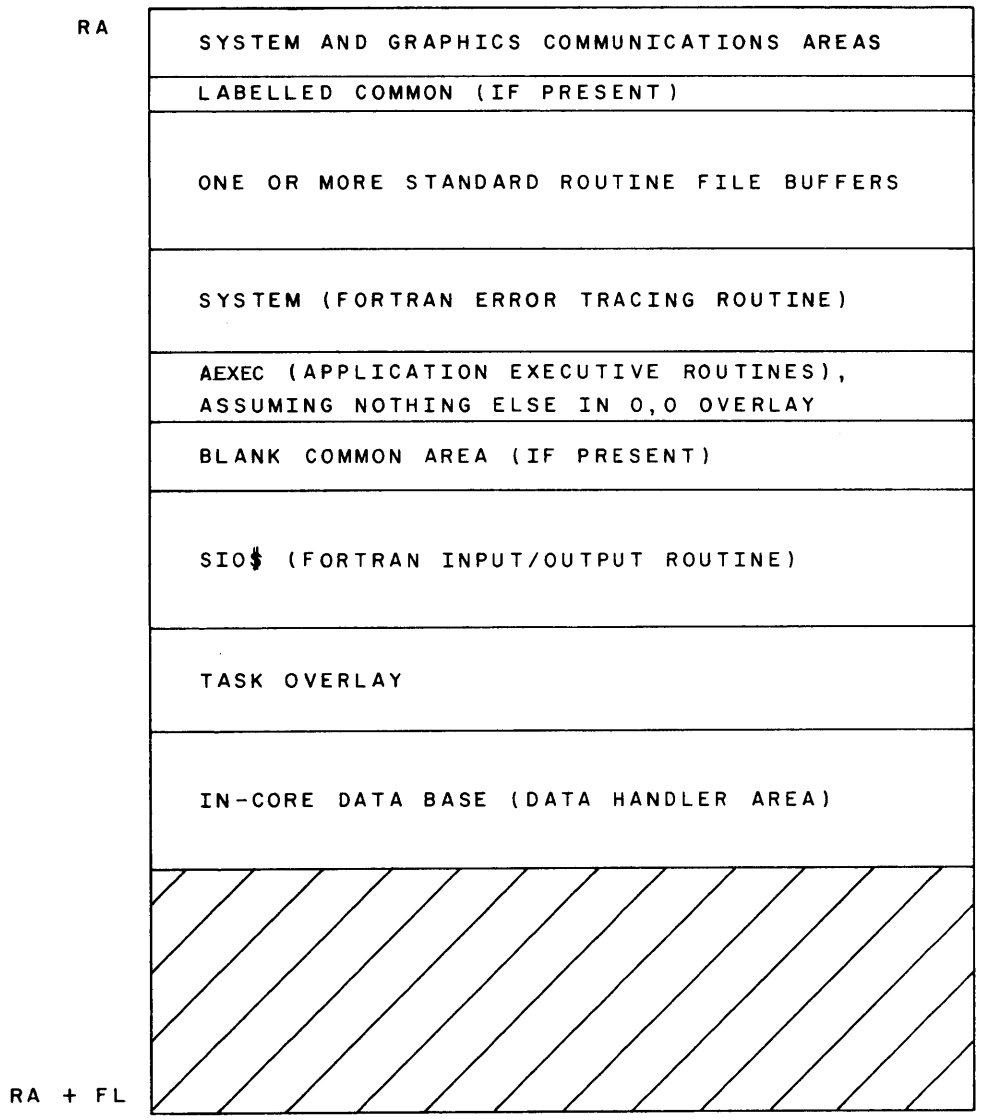


Figure 2-1. Graphics Control Point Field

GRAPHICS PROGRAM CARD DECK

User programs are compiled and executed on the 6000 series computer system. The user submits his application program as a normal FORTRAN batch job card deck; the following discussion assumes that the card deck is punched in standard 6000 Hollerith code.

The card deck consists of control cards, program cards, and data cards. The control cards specify how the job is to be processed; the FORTRAN program cards and the data cards follow the control cards in a deck. The deck ends with an end-of-file card (6-7-8-9 quadruple-punched in column one).

CONTROL CARDS

JOB CARD

The first control card, the job card, must indicate the job name, priority, central processor time limit, and memory requirements of the program. Fields are separated by commas; the last field is terminated by a period. Fields other than the job name may appear in any order. All capitalized letters must appear on the card; they are required by SCOPE.

```
n, Pp, Tt, CMfl, TPm, ECfl.
```

n	Alphanumeric job name, which begins with a letter and is 1 to 7 characters long.
Pp	Equals priority level in octal, with a 1 as the lowest priority; the upper limit on p is an installation option.
Tt	t equals central processor time limit for the whole job, including compilation and execution, in seconds and is 1 to 5 octal digits.
CMfl	fl equals total central memory field length of the job, with a maximum of 6 octal digits.
TPm	m is the number of tape drives required; it may not be omitted.
ECfl	fl equals total extended core storage field length required in terms of 1000_8 word blocks, with a maximum of 7777_8; this parameter may be omitted.

FTN CARD

The FTN card is usually the second control card in the deck. It calls the FORTRAN compiler and provides compiler mode, field length, and file names as follows:

```
FTN(I=fn, B=fn, E=fn, lp=fn, T)
```

I=fn	File name of source input, assumed INPUT if =fn is omitted.
B=fn	File name of binary output, assumed LGO if =fn is omitted.
E=fn	Prepare file name for input to UPDATE; assumed COMPS if =fn is omitted.

lp=fn Listing parameters; OUTPUT is assumed as the file name if =fn is omitted.
lp may be replaced by:

L	normal listing
X	non-ASA usage listing
R	assembler cross-reference table listing
O	object code listing

T Test mode for maximum error checking and traceback information of binary routines.

LGO CARD

This card calls the SCOPE general purpose system loader and begins program execution — regardless of the parameters on the FTN card. The format for this card is:

```
LGO.
```

AEFILE CARD

The AEFILE card calls the graphics task file creation utility routine, AEFILE, which restructures the program file identified by the second data card into an indexed random-access COMMON file or primary overlays, identified by the name on the first data card. AEFILE can also be used to add or replace overlays in the file and make changes within overlays.

This card has the format:

```
AEFILE.
```

NOTE

If AEFILE creates a local task file, use the creation/execution run deck shown in Figure 2-9 but insert a COMMON card between AEFILE and the source call card. The COMMON card is described on the following pages. The option to create a local file is provided so the user can catalog it as a permanent file.

SOURCE CALL CARD

This card is used in combined creation and execution runs. SCR. requests that the program beginning with OVERLAY (SCR, 0, 0) be placed in a file named KCB for execution (see Figure 2-9). This differs from separate creation and execution runs in that the creation run places the program on a COMMON file called OBJECT, where it can be called later by the execution run (see Figures 2-2 and 2-8).

The format for this card is:

```
SCR.
```

AEDUMP CARD

This card calls the AEDUMP utility routine, which reads a random-access file (the graphics task file), removes all rewritten records and indexes, and writes it as a serial-access file with an index as its first record.

The format for this card is:

```
AEDUMP (s, o)
```

- s — Name of the random-access file to be used as a source; this is the file created by AEFILE or AELOAD.
- o — Name of the serial-access file to be produced.

AELOAD CARD

The AELOAD card calls the AELOAD utility routine, which reads a serial-access file (the file produced by AEDUMP) with an index as its first record. AELOAD then writes the file as a random-access file with an index of disk addresses as its last logical record. The file produced by AELOAD can be used as a graphics COMMON file.

The AELOAD card has the format:

```
AELOAD (s, o)
```

- s — Name of the serial-access file to be read as a source; this is the output file of AEDUMP.
- o — Name of the random-access file to be produced.

COMMON CARD

This card attaches any existing COMMON file named in its parameter field to the program and changes its status in the file environment table so that no other program will have access to it while the current program is running. When the program the file is attached to terminates, the file is returned to the system and may be reassigned by another program's COMMON card. The COMMON card's format is:

```
COMMON, fn.
```

fn Name of the COMMON file (usually the graphics task file created by AEFILE or AELOAD) to be assigned to the program.

RELEASE CARD

The RELEASE card eliminates the COMMON file named in its parameter field from the system. When SCOPE encounters a RELEASE card, it changes the file's file name table/ file status table entry so that the file is reclassified as a local program file. When the program ends, all of its local files are automatically destroyed. This card has the format:

```
RELEASE, fn.
```

fn Name of the COMMON file (usually the graphics task file) to be destroyed.

EXIT CARD

When SCOPE detects a program error, it searches the program's control card record for an EXIT card. If it finds one, it performs any actions specified by the control cards following the EXIT card, then terminates the program.

If an error occurs and no EXIT card exists, SCOPE simply terminates the job with a dayfile message.

If no error occurs, an EXIT card (and any control cards following it) is ignored; SCOPE simply terminates the job with a dayfile message. The EXIT card format is:

```
EXIT.
```

or

```
EXIT (S)
```

If the S parameter is used, EXIT processing is also done when assembly or compilation errors cause termination.

PROGRAM CARDS

Several program cards are required by Interactive Graphics. Program cards, which are punched as standard FORTRAN cards, are separated from control cards and data cards by end-of- record cards (7-8-9 triple-punched in column one).

MAIN (ZERO-LEVEL) OVERLAY CARD

This card causes the FORTRAN compiler to translate the program overlay following it as a zero-level overlay. Zero-level overlays always reside in core when the program is at a control point, and serve to link blank COMMON areas between higher level overlays. The main overlay card has the format:

```
OVERLAY(lfn, 0, 0)
```

lfn Name to be assigned to the source file of overlays (produced by the SCOPE General Purpose System Loader).

CALL AEXEC CARD

This card calls the application executive AEXEC program; when encountered at compile and loading time, it causes the executive's AEXEC program to be loaded into the zero-level overlay as a subprogram from the SCOPE system library. This card has the format:

```
CALL AEXEC
```

If this card is not used, the programmer must supply his own executive zero-level overlay to setup a call to AEFILE, load tasks, fetch buttons, and so forth.

TASK LEVEL OVERLAY CARD

This card is used to begin each task overlay and serves as an end-of- record card for the overlay preceeding it.

The format for this card is:

OVERLAY (p, s)

p Primary overlay level number in octal; must be greater than zero and less than 100_8.

s Secondary overlay level number in octal; must be positive and less than 100_8.

Overlays need not be numbered sequentially in an input file.

DATA CARDS

If a graphics program uses the AEXEC program, there must be at least one data record in its deck.

The first data record contains the file name parameter cards used by the executive's AEXEC program. The file names on these cards are standard seven-character alphanumeric names, starting in column one of the card. The first card must contain the name assigned to the graphics COMMON file; the second card (used only during a file creation run) must contain the name of the file produced by the general purpose system loader. This source file name must agree with the name given on the program's main overlay card (see page 2-9).

SAMPLE PROGRAM DECKS

Figures 2-2 through 2-9 depict program decks for various task file creation, maintenance, and execution functions. The operation of the system utility routines called by the control cards is explained in more detail later in this section.

ZERO-LEVEL OVERLAY CONTENT

The zero-level overlays in all runs of a job must be identical. The overlays in the task file are linked to FORTRAN and application executive entry points within the zero-level overlay and are relocated with respect to the first word address of the zero-level overlay's blank COMMON. Unless the same zero-level overlay is used for all runs, task loading and COMMON linkage will not occur properly.

If the zero-level overlay, the size of blank COMMON, or the number of files used is changed, a new file creation run should be made to alter the linkages and loading addresses for each of the tasks in the task file.

If the name of a file or a blank COMMON location is changed without changing the zero-level overlay's core requirements, it is necessary only to change the task overlays affected by the name changes; this can be done with a file maintenance run.

All file requirements (such as INPUT, OUTPUT, or TAPE6) must be listed on the zero-level overlay's PROGRAM card; they may not appear on a PROGRAM card in any other overlay.

The FORTRAN compiler allocates file environment table entries and buffers for these files and sets pointers to the allocations for use during the execution run. Each subsequent allocation of a file with a given name is written over the previous one, so that a file listed in the zero-level overlay and in another overlay will have pointers only in the latter. Therefore, when the zero-level overlay is entered at execution time, the FORTRAN linkage routine will try to find a file environment table entry for file but will fail; the pointers that it searches for will be unavailable because they are in an overlay that has not yet been loaded.

When the linkage routine's search fails, the job is aborted with the diagnostic message:

```
NO OUTPUT FILE FOUND
```

This portion of a program would cause such a diagnostic:

```
OVERLAY (SOURCE, 0, 0)
PROGRAM ONE (TAPE6)
      •
      •
      •
OVERLAY (1, 0)
PROGRAM TWO (TAPE6)
      •
      •
      •
```

FILE CREATION RUNS

Figure 2-2 shows a typical card deck for an initial file creation run, using the application executive AEXEC program and the system AEFILE routine. This deck can create a file with a maximum of 63_{10} primary or secondary overlays.

A graphics COMMON file containing more than 63_{10} overlays can be created. A deck, such as the one shown in Figure 2-4, can be used to build a task file that contains as many overlays as the installation-specified limit MNOVL will permit (see AEFILE routine).

FILE MAINTENANCE RUNS

If a program library has been created for graphics jobs, and it has the same format as the sample deck shown in Figure 2-2, then a task file can be created from it. By using the system UPDATE program, the programmer can make corrections during the same run. Figure 2-3 shows a deck that will form a corrected task file from an UPDATE library tape; the routine in card deck LBTASK will be placed in the file SOURCE from tape OLDPL, and task file OBJECT will be produced.

Task overlays can be added to an existing graphics COMMON file by using AEFILE. Figure 2-4 shows a sample deck which adds a primary level overlay ADDTASK to the end of the file created by the deck in Figure 2-2.

Task overlays may also be replaced within a graphics COMMON file by using AEFILE. Figure 2-5 shows a sample deck that will substitute the revised primary overlay TASK1 for the original primary overlay TASK1 in the file created by the deck shown in Figure 2-2. The substitution is made according to the name given on the new task's PROGRAM card – the new task will replace the old task with the same name within the file.

Figure 2-6 shows a deck that will take the file OBJECT created by any of the preceding decks and store it in a purged form on magnetic tape as a file called SOURCE.

Figure 2-7 shows a sample deck that purges the file OBJECT1 (similar to OBJECT of Figures 2-2 through 2-5) and recreates it as file OBJECT for use in a subsequent execution run.

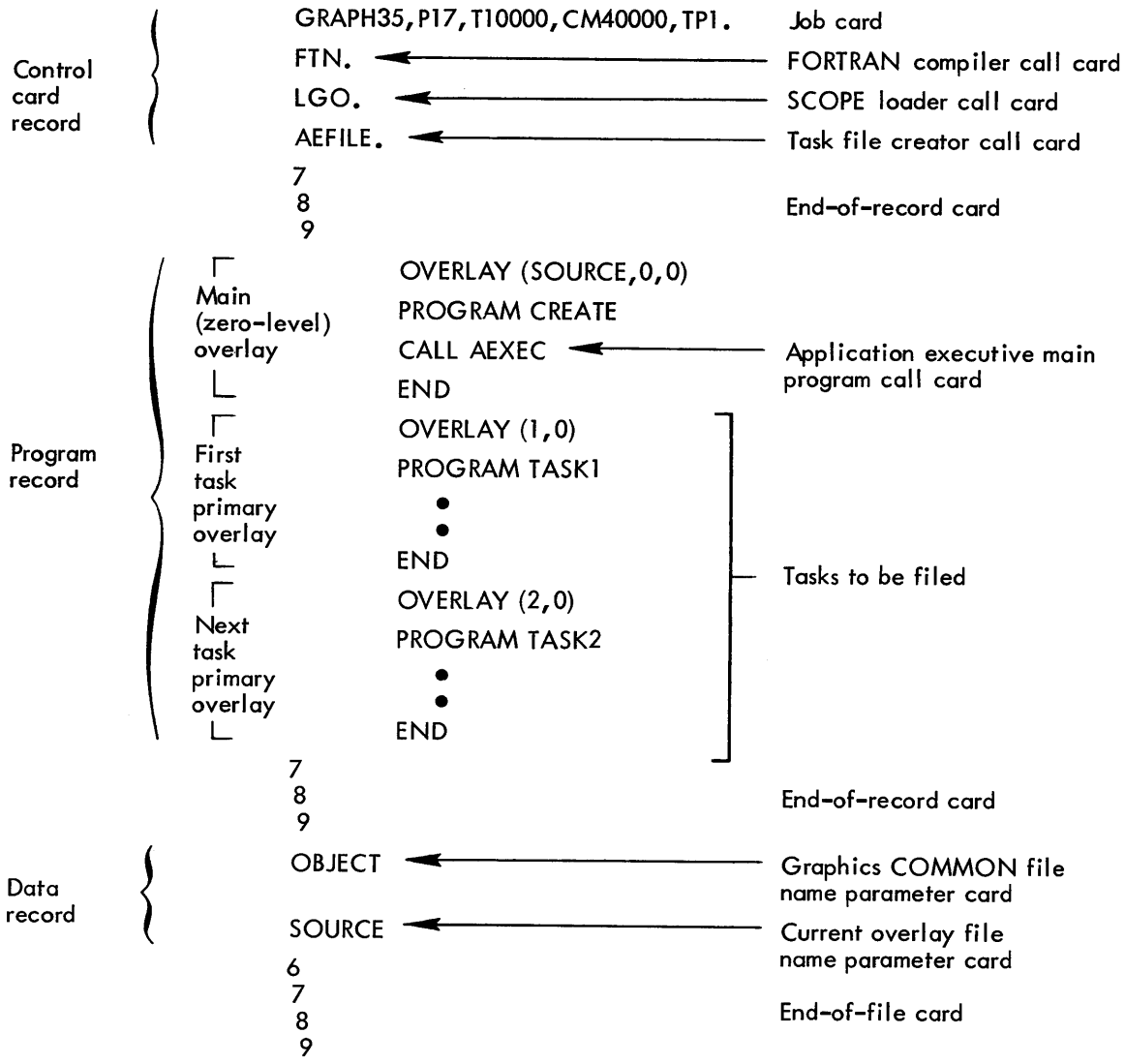


Figure 2-2. File Creation Run Deck

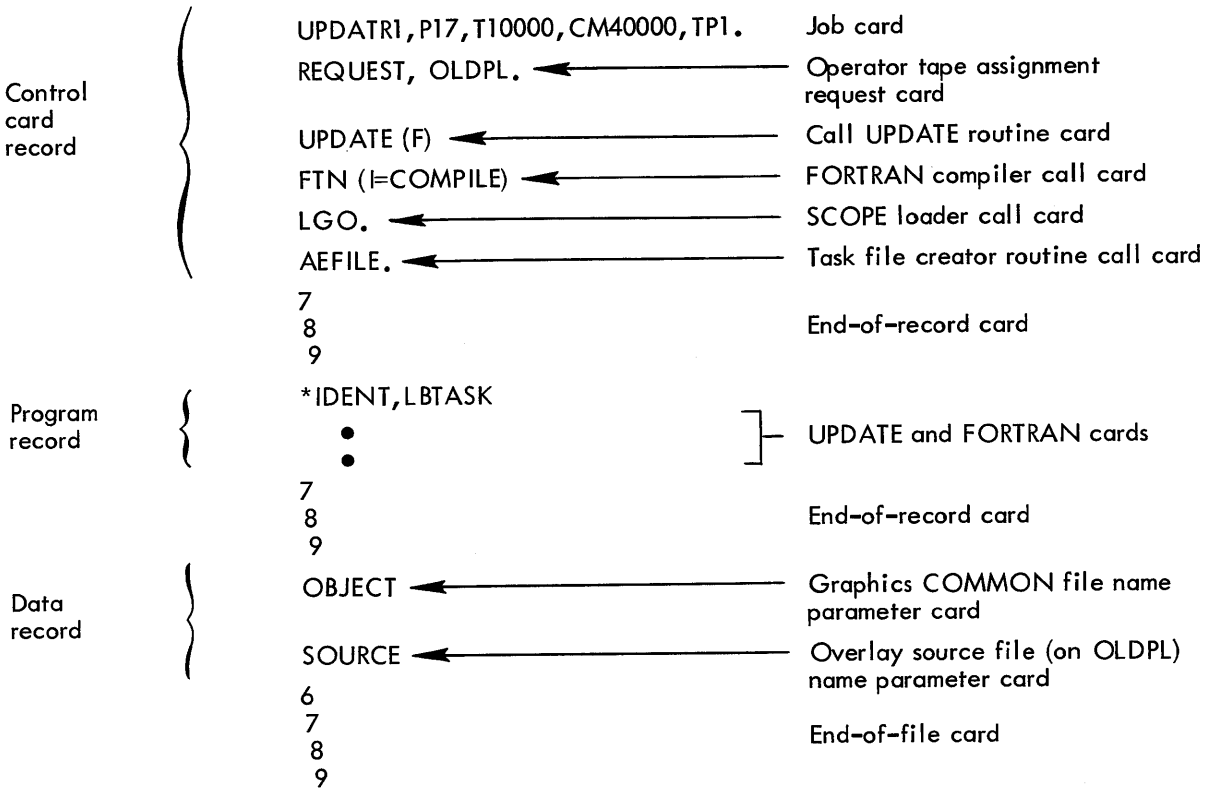


Figure 2-3. UPDATE File Correction and Creation Deck

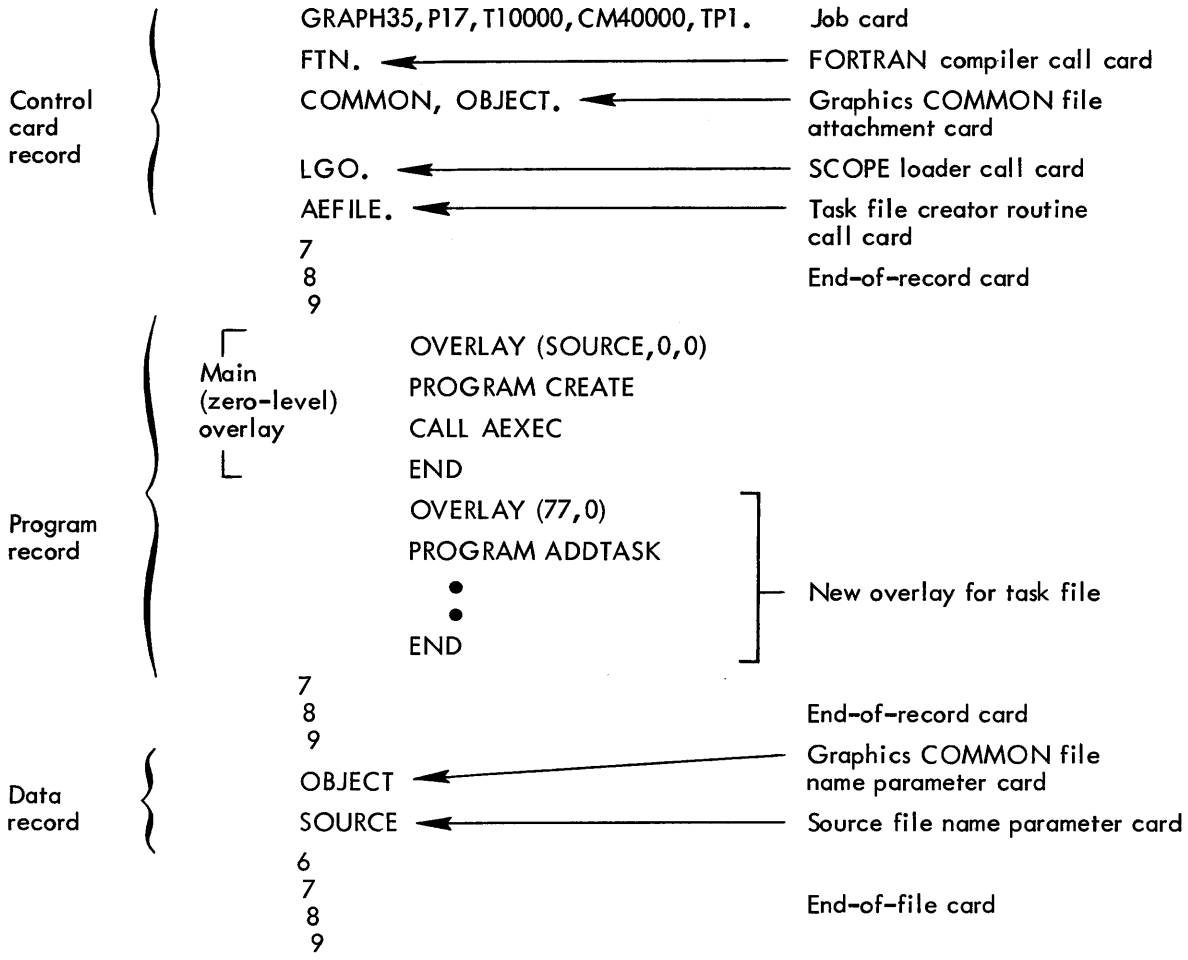


Figure 2-4. Task Addition Maintenance Run Deck

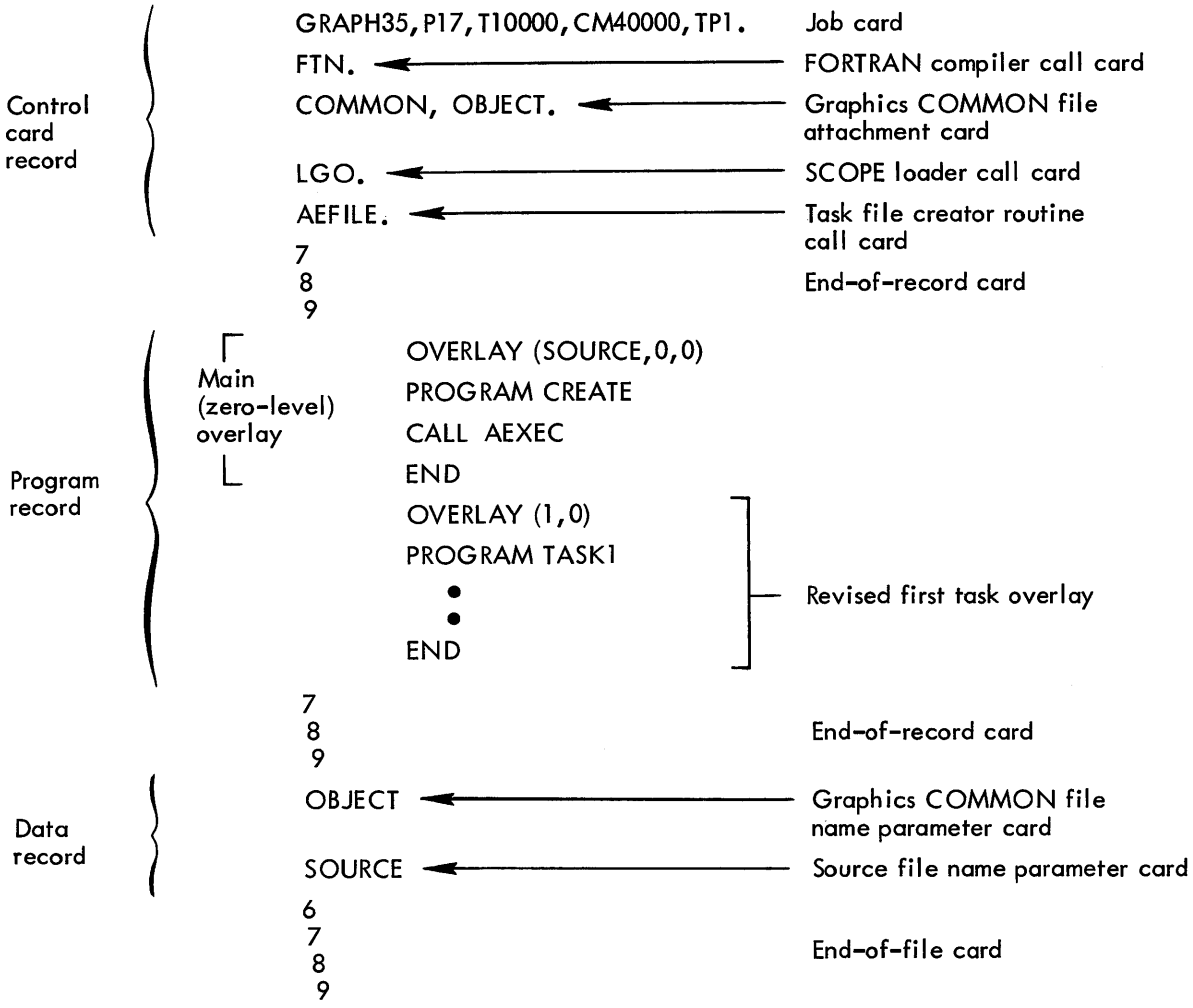


Figure 2-5. Task Replacement Maintenance Run Deck

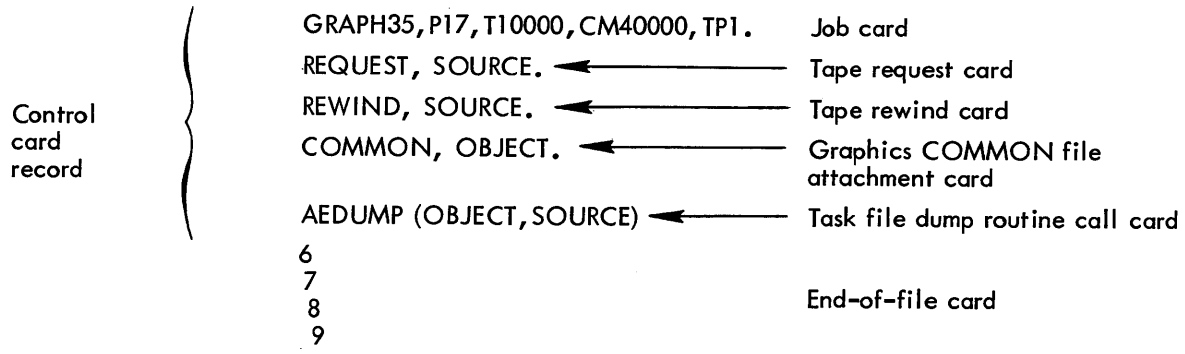


Figure 2-6. Sample Deck to Purge and Store File

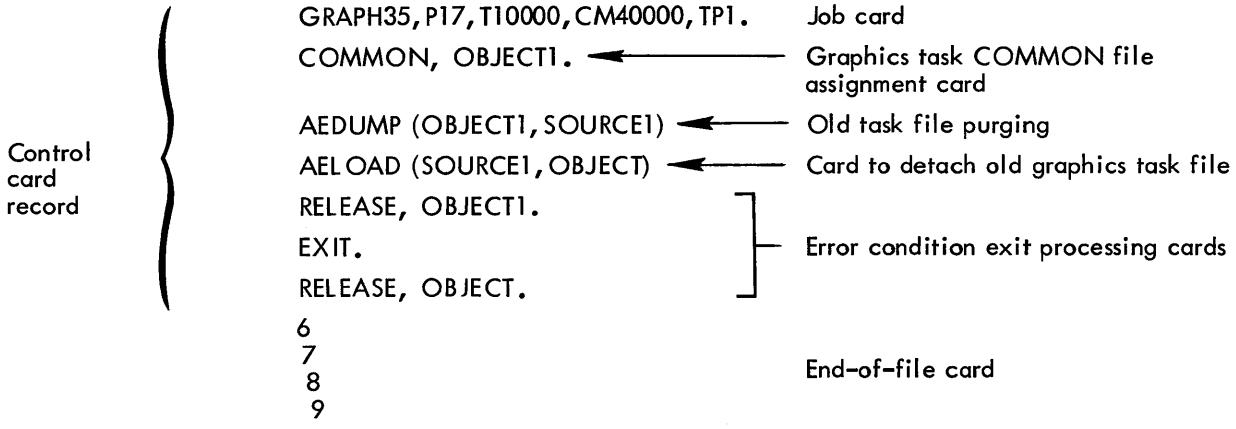


Figure 2-7. Sample Deck to Purge File Within System

PROGRAM EXECUTION RUN

Figure 2-8 shows a typical program execution run card deck; the program uses the graphics COMMON file called OBJECT, which was created by the decks in the preceding figures.

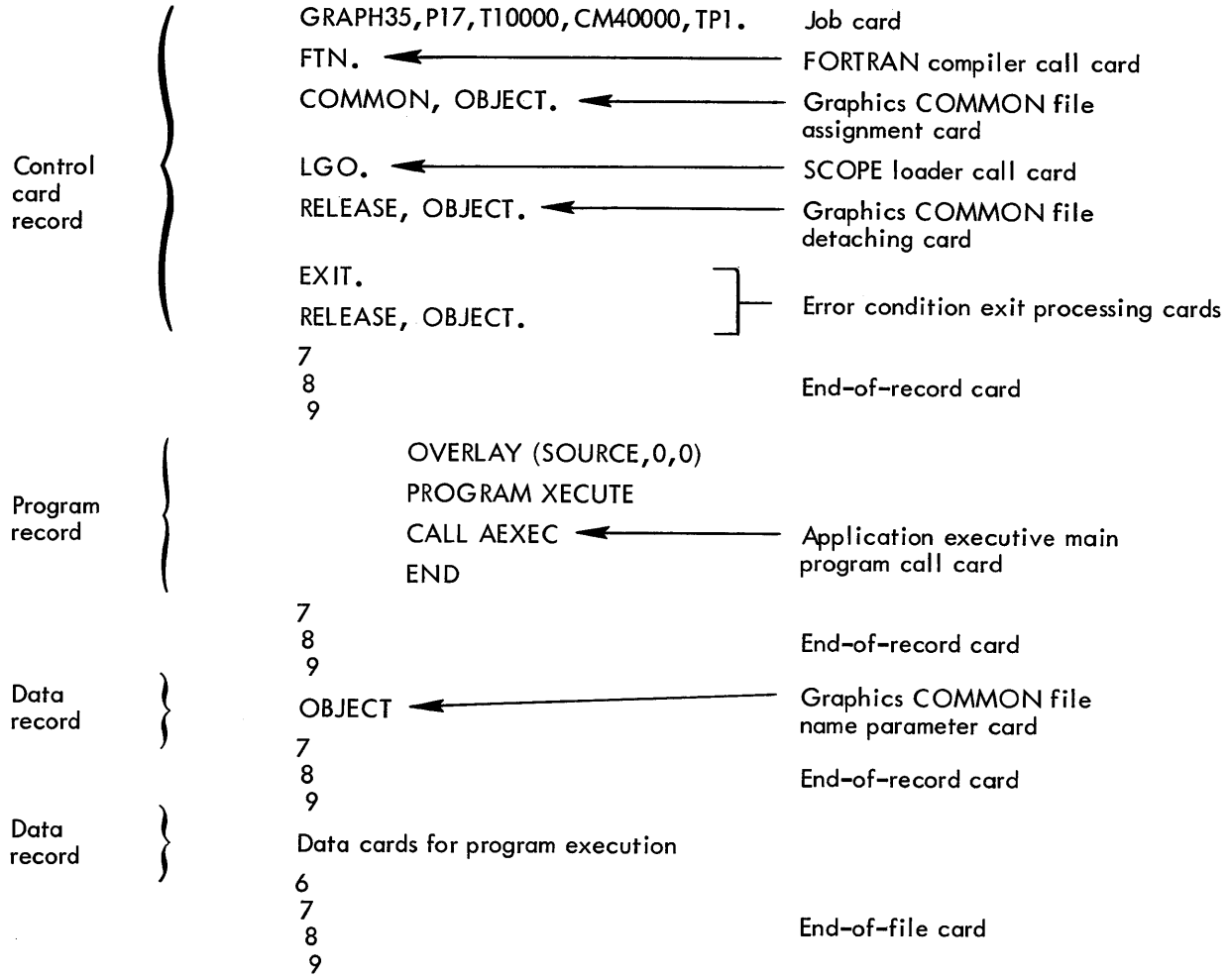


Figure 2-8. Execution Run Card Deck

COMBINED CREATION AND EXECUTION RUN

A graphics program can also create and execute its graphics COMMON file in one pass through the computer. Figure 2-9 shows a program card deck that combines the previously described creation and execution runs for the file called KCB. If the local file option is being used, insert the card COMMON, KCB. after AEFILE.

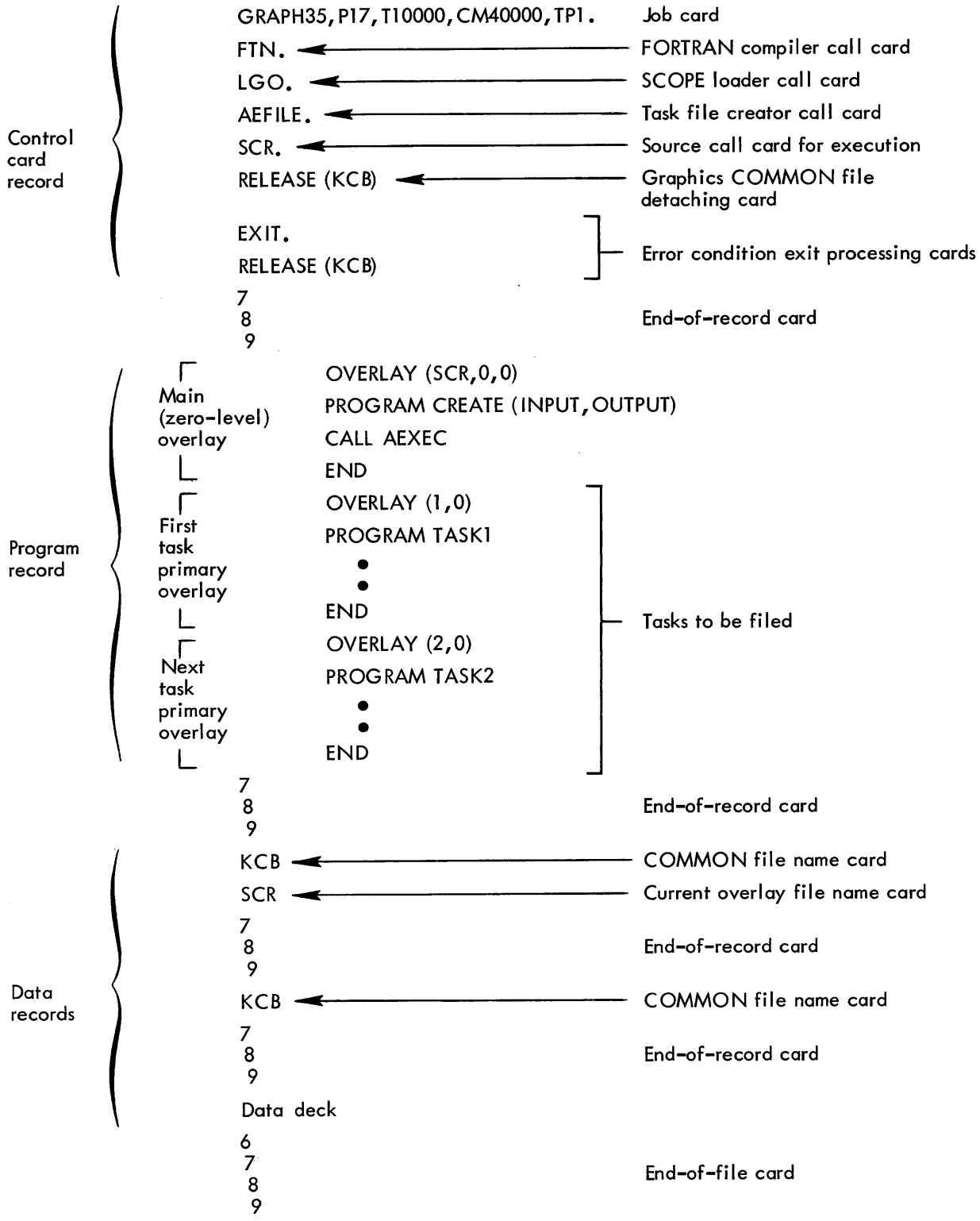


Figure 2-9. Creation and Execution Run Card Deck

SYSTEM UTILITY FUNCTIONS

In addition to the Scheduler and its subroutine, a graphics program uses SCOPE routines to create and maintain the graphics task file and to process abort conditions.

TASK FILE CREATION

Initially, application programs can enter the system either through a remote card reader at the 1700 site or at the 6000 series computer's card reader. Remote entry gives the programmer a convenient tool for program debugging.

After the program is submitted to the system, the control cards in its first logical record determine further processing.

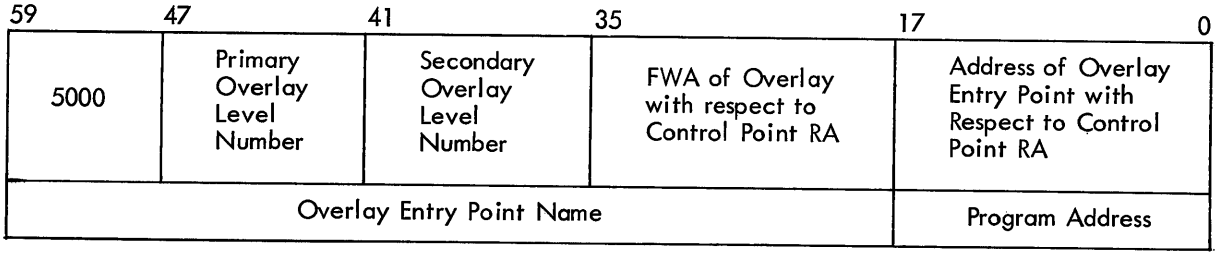
First, SCOPE queues the job in the batch input queue according to the priority on its job card and creates the proper entries in the system file environment table/ file name table (FET/FNT).

When SCOPE assigns the program to a batch job control point, the next control card is processed. This is the FTN card, which calls the FORTRAN compiler.

After compilation, the next control card is processed. For a graphics task file creation run, this would be the LGO card.

The LGO card calls SCOPE's general purpose system loader (GPSL), which takes the compiler's output, satisfies all 6000 Basic Graphics Package references from the system library and organizes this data into a serial-access scratch file of overlays. This file is given the name specified on the main (or zero-level) OVERLAY card; it is written one overlay to a record and positioned after the program's first record (the main or zero-level overlay record).

Each record of this file contains two tables. The first is the 77 or prefix table; the second is the 50 or overlay table. The 50 table contains two header words with the format:



followed by the binary text of the overlay.

SCOPE continues processing the LGO card by starting program execution; the program is initially treated as a batch job and executed at a batch job control point.

These instructions, which are supplied by either the programmer or the application executive AEXEC program (see Section 6), place file names in RA+3 and RA+4 of the program's current control point area.

The program then passes control back to SCOPE for normal termination of LGO processing. This consists of executing the next card (which should be an AEFILE card in a file creation run deck) in a control card deck.

AEFILE ROUTINE

AEFILE is the graphics task file creator; it reads the name of the loader-created overlay file from RA+3 and then writes that file (without the zero-level overlay record) on the system disk as absolute-addressed FORTRAN overlays.

This new file is the program's graphics COMMON file.† It consists of named random records, each containing a 50 table and a primary overlay (the 77 table is not written into the graphics COMMON file). The record name is taken from the overlay entry point name in the second word of the 77 table.

AEFILE catalogs the disk address of each overlay record and writes a task directory containing this information as the last logical record of the graphics COMMON file.

TASK DIRECTORY

The task directory (Figure 2-10) contains pointers for MNOVL overlays (MNOVL is an installation parameter). Since each task is accessible through its name in the task directory, the applications programmer can make additions to and deletions from an existing task file.

The task directory contains two blocks of information. The first block is a standard index for a named random file. It consists of one header word and two central memory words for each overlay record in the graphics COMMON file. The negative value of the header word indicates that the information block following it is a named random index.

The second block of information contains one entry (a single central memory word for each overlay record). This block is treated as a suffix to the index in the first block and is used by the application executive routines to load the task overlay during program execution.

---

†An installation parameter exists by which AEFILE creates local task files. The programmer, by use of the appropriate SCOPE control cards, may choose COMMON or PERMANENT files (see Figures 2-11 and 2-12). For additional information, consult the installation procedures.

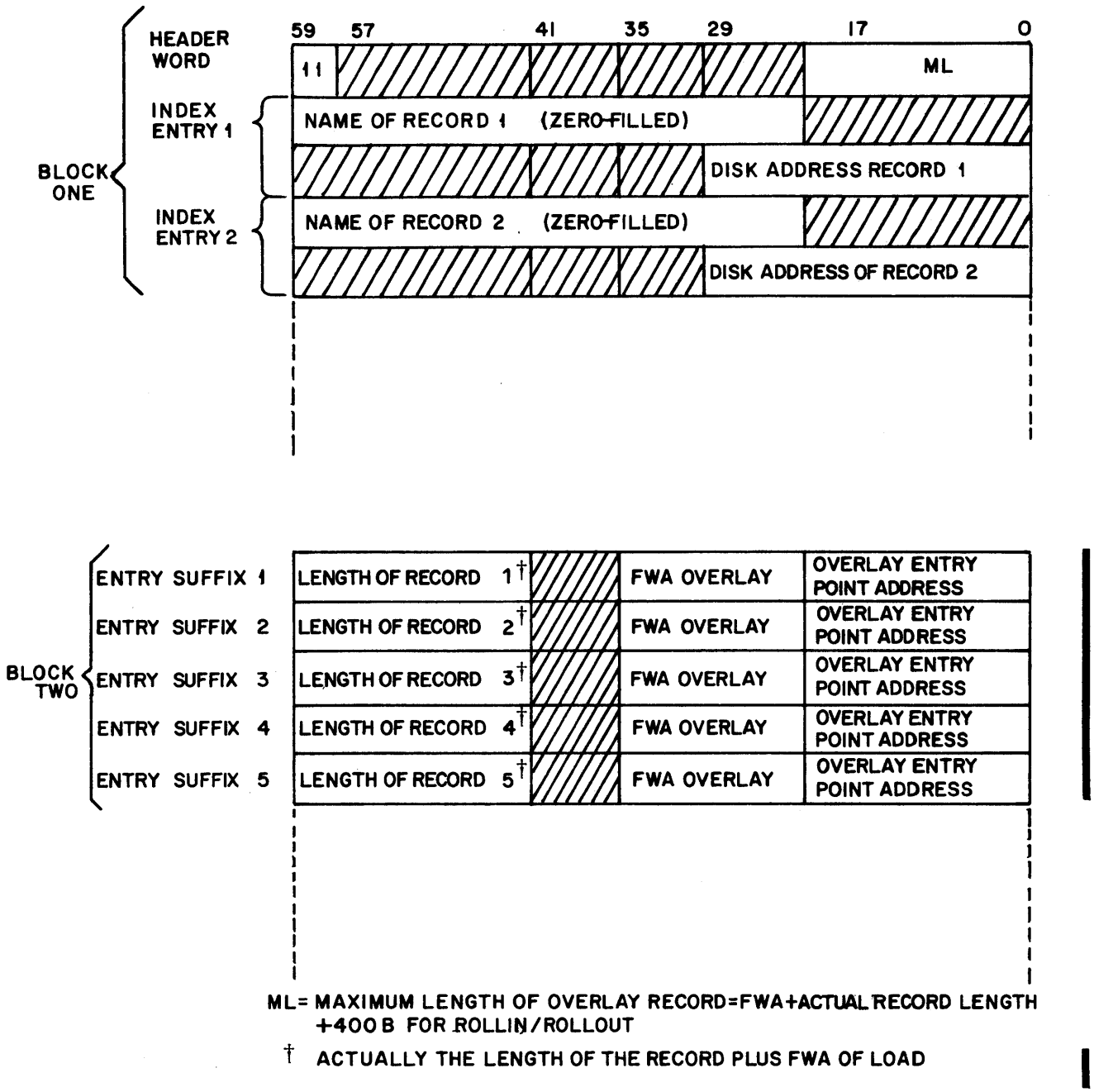


Figure 2-10. Task Directory

Only the first block of the task directory is used to read or write the graphics COMMON file, but both blocks are included in the index pointers when the file is closed or opened so that they will be retained on the disk as a catalog.

AEFILE ACTIONS

Before AEFILE can create a graphics COMMON file, it must make FET and FNT entries for both the COMMON file and the loader-created overlay source file; AEFILE uses SCOPE library macros and the contents of RA+3 and RA+4 to do this. If AEFILE detects an error in the table entries when the macros finish, it produces a dayfile message (see Appendix B) and aborts the job.

The graphics COMMON file entry defines the file as a system COMMON file and associates the programmer's graphics COMMON file name with it. The source file entry is used to save that file on the disk after the graphics COMMON file is written; the source file is treated as a local file and is destroyed when program execution ends.

After the entries are made, AEFILE uses SCOPE library macros to open the COMMON file, read the overlay source file, write the COMMON file, and close the overlay source file. These SCOPE macros write the graphics COMMON file on the most easily accessed allocable device (usually the system disk).

If AEFILE finds that FET and FNT entries already exist for the graphics COMMON file, it opens the file, saves the index, adds or inserts the contents of the overlay source file to the COMMON file, then writes a new task directory containing the latest index entries.

TASK FILE MAINTENANCE

If AEFILE is used to replace a task in an existing graphics COMMON file, it performs the action logically but not physically. This means that the old copy of the task still occupies storage space in the file, but is not listed in the new task directory index.

For example, the file OBJECT created by the decks in Figures 2-2 and 2-5 would contain:

TASK1
TASK2
•
•
•
TASK77
Old Index
New TASK1
New Index

A file like this should be purged after several debugging or updating runs to keep it from wasting mass storage and becoming unwieldy. Purging is done with the AEDUMP and AELOAD routines at a regular batch processing control point.

AEDUMP ROUTINE

AEDUMP is a system library routine that is called by a control card; it requires 15K words of memory. AEDUMP reads the indexed random file named by the first parameter on its control card and writes a new sequential file with the name specified by its second control card parameter.

The sequential file created by AEDUMP contains the index of the random file as its first record. Although the disk addresses in the index are meaningless, the record names and index suffix entries do not have to be altered to recreate a random file.

The other records of the sequential file are the binary text task overlay records; these records are written in the order that they are listed in the index. Only those records from the random file that are listed in the index are written into the new file. Unlisted records are skipped; therefore, the file created from the records in the example above would contain:

New Index
New TASK1
TASK2
•
•
•
TASK77

This sequential file could then be written on tape for storage outside of the system, or it could be used immediately to recreate a random task file – using the AELOAD routine.

AELOAD ROUTINE

AELOAD is also a system library routine and is called by a control card. AELOAD reads the sequential-access file named in the first parameter of its control card and creates a random-access COMMON file with the name specified by the second control card parameter.

The sequential-access file used by AELOAD need not be located in mass storage; AELOAD will call a tape driver to read the file if the programmer has supplied a valid REQUEST control card in his job deck.

The file created by AELOAD is structured exactly as one produced by AEFILE. The new task directory contains new disk addresses; the name of each task record is checked against the sequential file index as the record is written in the new file (if the names do not agree, a diagnostic message is produced and the job is aborted).

The AELOAD graphics task COMMON file can be used for program execution by the card deck shown in Figure 2-8.

AELOAD requires 15K words of memory.

GRAPHICS PROGRAM ABORTING

If an applications programmer wants to abort a program, he usually creates a light-button at the graphics console to call GIABRT (see Section 6). This routine displays a dayfile and console message and calls the SCOPE abort routine.

When a 6000 Basic Graphics Package routine finds a programming error, it produces a dayfile and console message; the program's application executive routine then issues the messages and calls the SCOPE abort routine.

If the 6000 series computer detects an error condition during program execution, it sets a control point flag which calls the SCOPE abort routine and produces a dayfile message.

If the 1700 series computer detects an error condtion or an illegal request, it generates MSOS IMPORT HS directive code 23 (the 1700 operator can also generate this code with a type-in command). This sends a message to the affected console and informs EXPORT HS to flag the program for abortion. The Scheduler detects EXPORT's flag during the next rollin of the program, issues a dayfile message, and calls the SCOPE abort routine.

EXPORT HS removes an aborted graphics program from the Scheduler's input queue and disconnects any graphics consoles assigned to it.

The SCOPE abort routine releases all of the job's files to the system and sends output files to the 1700 if the program originated there. It will dump a core listing with the output file if the program requests it by using a control card.

After a graphics abort, the dedicated memory assigned to the program is not released to batch jobs, as is the normal system procedure, but is retained for future graphics programs.

FILES

The programmer uses standard SCOPE control cards to attach all files used in the graphics program. A maximum of 51 files per program can be handled by the Scheduler; this number includes all local scratch files, the job's graphics COMMON file, the overlay source file named in the zero-level overlay card parameter field, and all data handler files (see Section 6). Up to 28 of these files may be local, and the remaining 23 may be any combination of COMMON and permanent files. See Figures 2-11 and 2-12 for sample creation and execution decks for a permanent file.

Once a file is attached to the graphics program, the file is not available to other programs.

GRAPHICS COMMON FILE

The file created by the AEFILE and AELOAD routines is a graphics COMMON file.

COMMON file names must be unique for each user. Using the last two digits of the file name to designate a user graphics console would eliminate possible duplications.

LOCAL FILES

All files that are local are rolled out with a program; thus, the remaining graphics control point(s) can be used (if available).

INPUT FILES

Tape and card files other than the FORTRAN input file must be put in mass storage before being used by a graphics program. These files are read in and made COMMON with names different from that of the graphics task COMMON file.

OUTPUT FILES

All tape output is through a disk file. After a graphics job is completed, a SCOPE utility program can be used to transfer the data to magnetic tape.

PERMANENT FILES

If the installation option for local files is used, the local files may be cataloged as permanent files and later attached for execution.

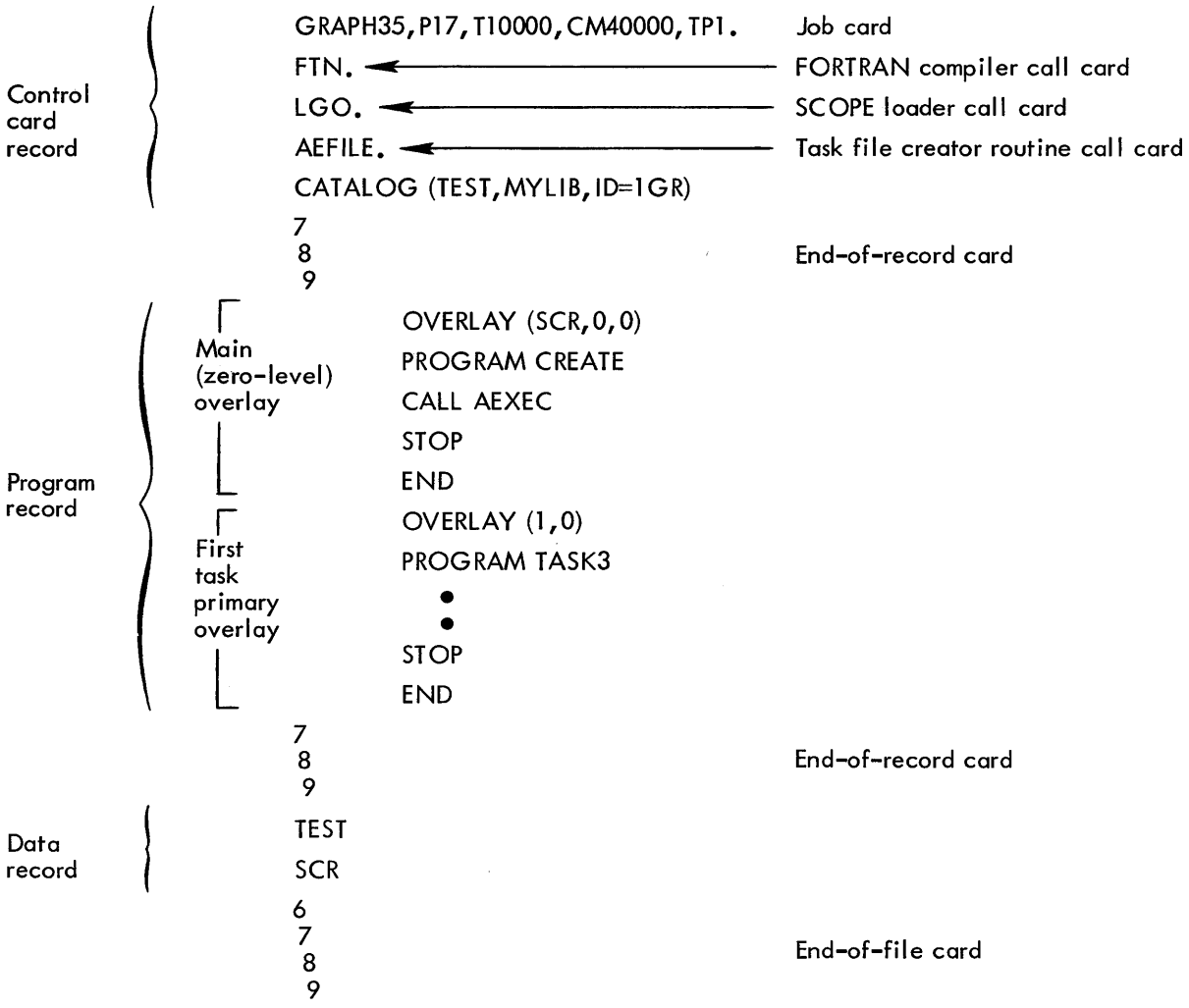


Figure 2-11. Sample Deck to Create A Permanent File

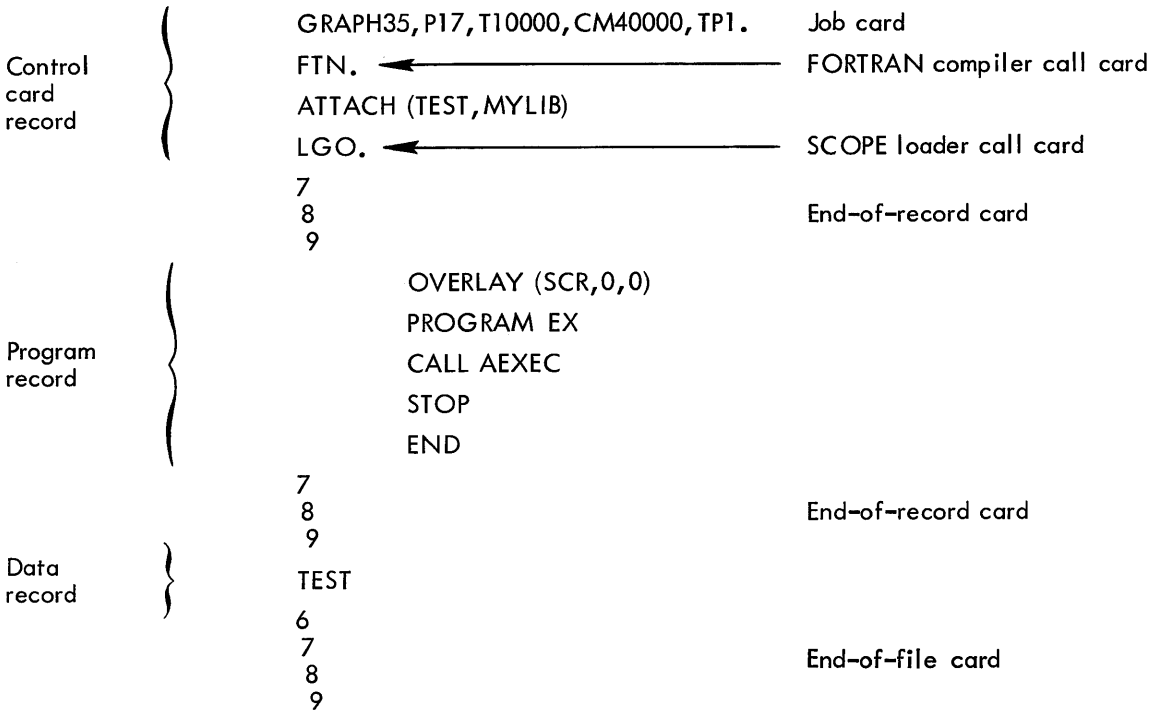


Figure 2-12. Sample Deck to Execute A Permanent File Task

GRAPHICS HARDWARE INFORMATION 3

Proper use of the 6000 Basic Graphics Package routines by the applications programmer requires a general knowledge of the graphics hardware. This section describes those system characteristics which are used by the 6000 series Interactive Graphics System applications interface routines.

GENERAL DESCRIPTION

The graphics system provides an interface for the handling of graphic or alphanumeric information; entries or modifications made at the console are placed into the 1700 series computer in digital form and become available for use by the 6000 series computer system. This graphics input becomes visible on the cathode-ray tube and can be used for information processing by an applications program under console operator control. Results of such processing can be immediately displayed on the screen. Static display of graphic and alphanumeric data at the consoles is provided by buffer memories because the consoles are essentially off-line devices. The 1700 is used to process display-change information, thus saving transfer time from the 6000 series computer.

GRAPHICS CONSOLE

The graphics console is the input/output and control center for the Interactive Graphics user. The complete range of system graphics capability can be controlled from the console without recourse to other points of control. The console is designed for maximum operator utilization and comfort and can be used efficiently at normal room light levels.

The console cabinet is a desk-size unit which mounts a rectangular housing assembly, off-centered to the left, and provides a writing surface to the right. The housing assembly contains a magnetic shield and a 20-inch diameter cathode ray tube centered on the front panel housing.

The cathode tube is a precision, 52-degree, high-resolution unit and has a nearly flat display surface to minimize parallax error. The tube is equipped with an implosion shield for the protection of the operator and is coated with a two-layer P-7 phosphor. One layer produces blue-violet light with a short persistence to facilitate light-pen tracking. The other layer produces yellow-green light and has a longer persistence to eliminate flicker. With a continuously refreshed display, the light from both phosphor components combines to appear

light blue to the human eye. The deflection yoke and driving circuitry of the console are designed to make the entire 314 square inches of cathode-ray tube surface available for display. The tube has a resolution of 1000 lines in 20 inches.

Data can be entered on the cathode-ray tube via the lightpen or one of three optional keyboards.

CONTROLS

The controls available to the console operator include the keyboards, lightpen, light-registers, and light-buttons. The light-registers and light-buttons are defined by the application program and formed for display on the screen by the 1700 Basic Graphics Package routines.

FUNCTION KEYBOARD

The 16-key function keyboard can be used to tell the application program that an operation is requested (see Figure 3-1).

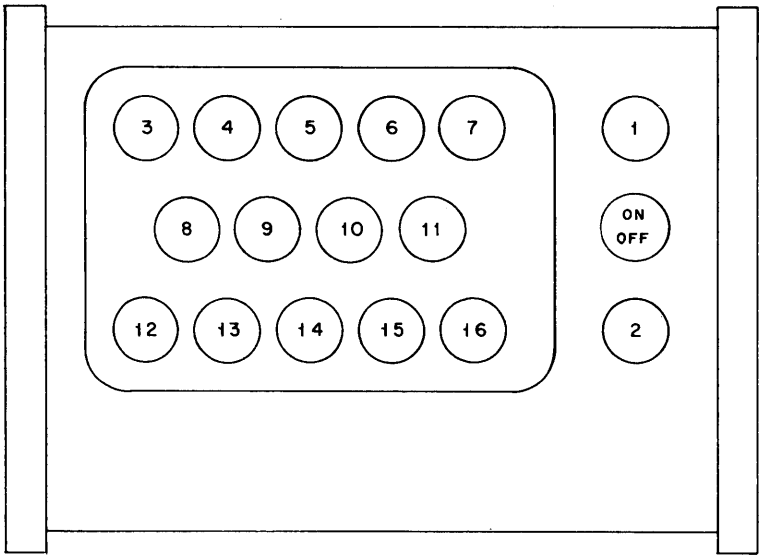


Figure 3-1. Function Keyboard

Fourteen buttons contain a snap-action switch that remains on after an initial press and off after being pressed again. The remaining buttons must be held down to give an "on" status. Each button has an internal light that shows the operator when the button is on. Removable plastic cards may be placed over the keys to label the function of each. All keys can be given new functional assignements by the application program through the 6000 Basic Graphics Package.

Any change in the status of a key produces an interrupt at the 1744 Controller. The 1700 Basic Graphics Package then fetches the on/off status of all 16 keys as bits in a status word. These status bits are placed in the IH and IV coordinate locations of a display item ID block (see Section 5) created for the keyboard by the application program through a call to the GIKYBD routine of the 6000 Package. Table 3-1 shows the relation between the coordinate bits and the keys; a 1 in a coordinate bit indicates that the button is on.

TABLE 3-1. FUNCTION KEYBOARD STATUS IH, IV

Coordinate Bit	Keyboard Button
IV 0	1
IV 1	2
IV 2	3
IV 3	4
IV 4	5
IV 5	6
IV 6	7
IV 7	8
IV 8	9
IV 9	10
IV 10	11
IV 11	12
IH 0	13
IH 1	14
IH 2	15
IH 3	16

The application program retrieves the ID block through the application executive, GIFID, GIFSID, GIBUT, or AELBUT routines of the 6000 Package; it then determines the function requested by testing the values of the coordinates.

ALPHANUMERIC KEYBOARD

The alphanumeric keyboard (see Figure 3-2) provides typewriter-like symbolic input to the application program. The keyboard layout is similar to that of a conventional teletypewriter. A key causes an interrupt at the 1744 Controller and enters an 8-bit ASCII character code in the left-hand portion of a status word that is fetched by the 1700 Package. The characters corresponding to those in the software alphanumeric font are collected into line images and displayed on the 274 Console screen in the currently defined light-register.

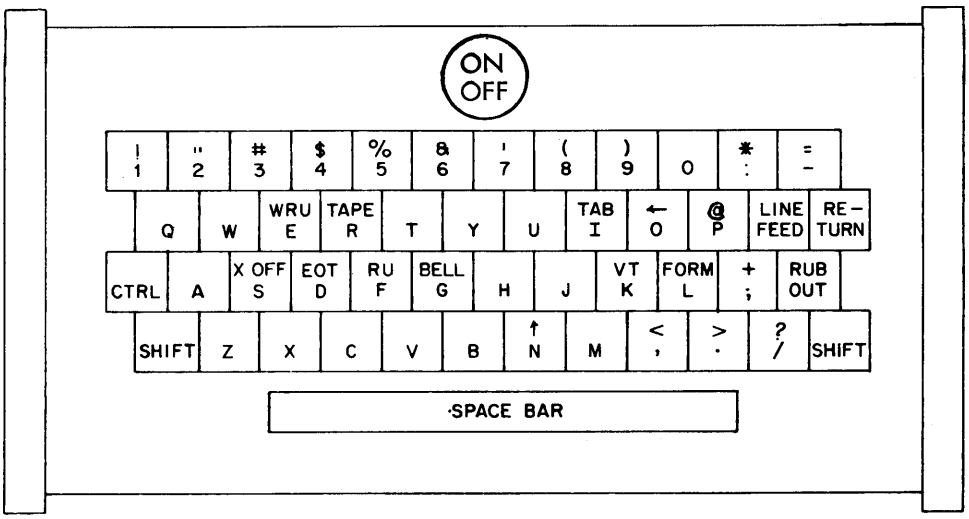


Figure 3-2. Alphanumeric Keyboard

The application program can acquire the console's input through calls to the 6000 Package GIANS and GIANE. If the package GIEOM routine has been used to assign an ID block to a particular keyboard character, that character will clear the register when it is entered. The RETURN key functions as an end-of-message character.

LIGHTPEN

The lightpen has two functions: tracking and picking. Tracking may be used to place a light source (the tracking-cross) at any desired position on the console screen so that a graphics entity may be created there or to designate that position as an area of interest to the user. Picking may be used to select an entity currently being displayed, to define points on a displayed entity, or to select a light-button or tracking-cross.

LIGHT-REGISTERS

The light-registers allow the user to input and retrieve alphanumeric information and permit the Interactive Graphics System to display error diagnostic messages. The number and locations of the registers are defined by the application program through 6000 Basic Graphics Package GUAN calls. If none have been defined, the system defines its own at (-552, 1600) on the screen (for error messages); otherwise, the last one defined by a call to a graphics utility routine in the program is used for system messages.

LIGHT-BUTTONS

The light-buttons are light spots on the console screen that are identified by a letter, digit, symbol, or instruction code specified by the application program. Any displayed entity or physical control key can also be defined as a light-button. Buttons are used to control ID block queueing (see Section 5) and to initiate tasks.

DISPLAY PRESENTATION

The entire 20-inch diameter cathode-ray tube screen can be used for display presentation. Points on the screen are addressed by a Cartesian coordinate system called the display grid.

DISPLAY GRID

The display grid (see Figure 3-3) consists of 4095 addressable points on the horizontal (H) axis and 4095 addressable points on the vertical (V) axis; coordinates can be given either octally or decimally when addressing a point. Coordinate 7777_8 equals coordinate 0000 on both axes.

The grid is larger than the screen so that all points on the screen can be addressed; points beyond the edge of the screen can be addressed by a programmer but are invisible to a user directly in front of the screen (if viewed at an angle, such points can be seen reflected off the side of the tube). There are approximately 200_{10} grid points per linear inch; however, because the cathode beam is wider than the distance between adjacent points, the console controller drops the least significant bit from each coordinate address of a point.

NOTE

The console controller address may vary over a range of plus or minus five grid units per inch, depending on the customer engineer's adjustments.

The distance between two adjacent grid points is called a display grid unit (dgu).

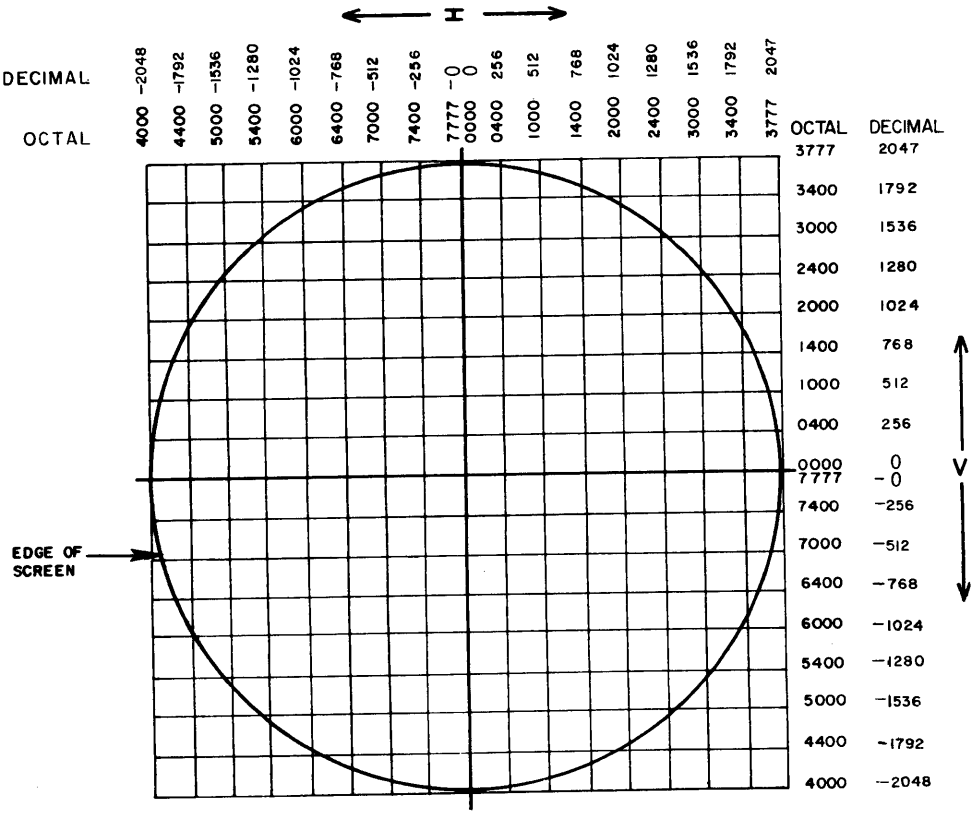


Figure 3-3. Display Grid System

SCREEN ORGANIZATION

The organization of the screen is completely up to the programmer. However, certain conventions may be used for a wide variety of applications. These conventions allow a programmer to make maximum use of the screen area, yet help him avoid addressing grid coordinates off the screen.

WORKING SURFACE

One convention is to divide the screen into a working surface and a control surface. The working surface is reserved for the display of graphics forms and is contained within the frame or frames defined by the programmer (see GULINE and GUARC, Section 6). The frame may or may not be displayed.

CONTROL SURFACE

The control surface is defined as the area outside of the frame or frames and is normally reserved for light-buttons, light-registers, and the tracking-cross (when it is not in use on the working surface). Figure 3-4 shows a sample of one type of screen organization.

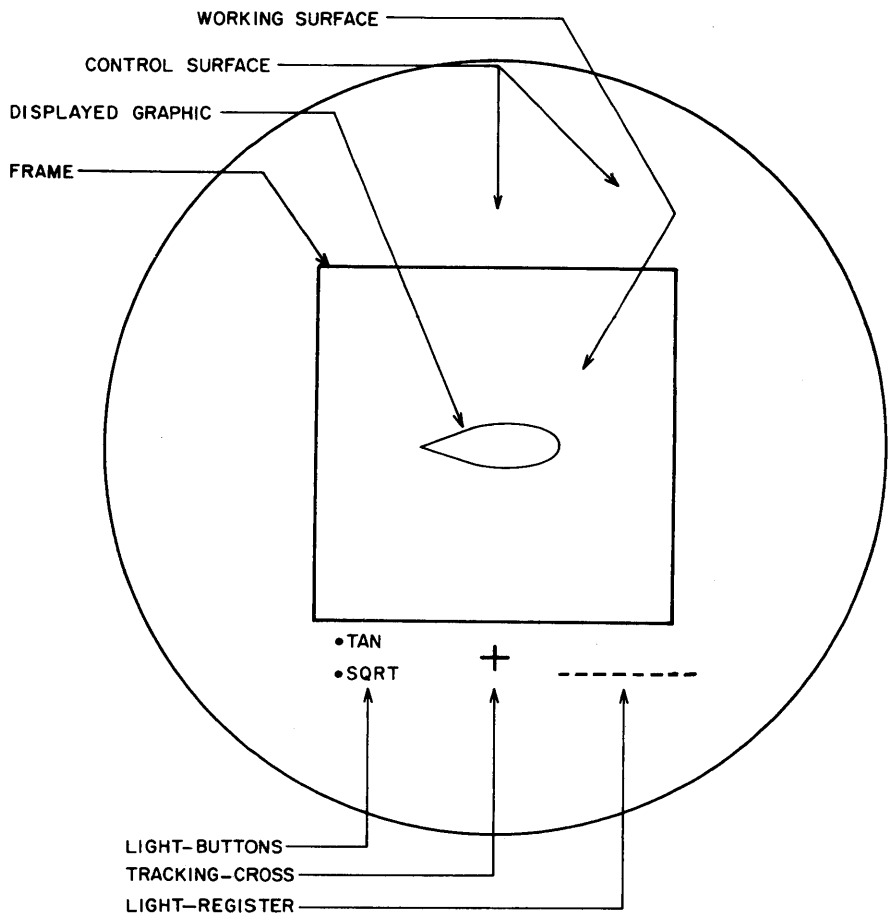


Figure 3-4. Sample Display Surface Organization

FRAMES

Table 3-2 defines possible frames within the screen area of the display grid. Several frames may exist on the screen at the same time; they may overlap, or each figure may have its own frame. The system software defines all frames as right rectangular areas, and frames may be centered anywhere on the screen.

TABLE 3-2. SAMPLE FRAMES

Frame Size	Center Coordinates	Right Corner Coordinates
Maximum square, approx. 14 by 14 inches	IHCEN = 0000B IVCEN = 0000B	IHCOR = 2570B or 1400_{10} IVCOR = 2570B or 1400_{10}
Horizontal rectangle, approx. 11 by 17 inches	IHCEN = 0000B IVCEN = 0000B	IHCOR = 3244B or 1700_{10} IVCOR = 2114B or 1100_{10}
Vertical rectangle, approx. 17 by 11 inches	IHCEN = 0000B IVCEN = 0000B	IHCOR = 2114B or 1100_{10} IVCOR = 3244B or 1700_{10}

POTENTIAL PHOSPHOR DAMAGE

It is possible for programming errors to cause endlessly repeated or excessively intense display of an item at the same location on the screen. This may cause damage to the cathode ray tube phosphor. When one of these conditions is detected by the console operator at program debugging time, the console must be turned off immediately by use of the console power switch above and to the right of the screen. Console power status does not affect the operation of the computer or of the Interactive Graphics System.

1744 DIGIGRAPHICS CONTROLLER

The controller uses a standard 1700 memory module (1708) of 4096 16-bit words for a buffer memory with an option for an additional 4096 words. A 1700 programmer may use the buffer memory as a display buffer or as an auxilliary 1700 series computer storage device.

As a display buffer, only bits 00 through 12 (with the exception of function and status codes) contain meaningful data. As an auxiliary random access storage device, all 16 bits can be used. Refer to the 1744 Digigraphics Controller Reference/Customer Engineering Manual (see Preface) for further details.

1700 GRAPHICS FUNCTIONS 4

BUFFER TRANSLATOR

The buffer translator is called by MSOS IMPORT HS when the 1700 receives a data buffer from EXPORT HS or is ordered to send a data buffer to the 6000 series computer.

The translator program will unpack the EXPORT HS buffers and put the calling parameters of the 6000 Basic Graphics Package into a format that the 1700 Basic Graphics Package will recognize. The translator also loads buffers for transfer to the 6000 series computer from the 1700 Basic Graphics Package. All alphanumeric characters are code converted by the translator into or from 1700 internal code. Floating-point conversions are done in the 6000 series computer by the 6000 Package routines.

PROGRAM ABORTING

The translator is also responsible for aborting graphics programs at the 1700. If a 1700 Package routine attempts to communicate with a console but the console's driver routine detects a communication error or failure, the Package routine sets a flag to inform the translator of the condition. The translator then displays an appropriate message on the teletypewriter and sends IMPORT directive code 23 to the 6000 (see Graphics Program Aborting, Section 2).

The translator also aborts programs if it detects an invalid IDDAD, IDDADI, or MAD programming parameter while it is processing a buffer from EXPORT HS. In this case, the translator returns a 1700 ABORT message to the 6000 series computer, displays an appropriate MSOS message (see Appendix B) on the screen of the affected consoles and at the teletypewriter, and sends the MSOS IMPORT HS directive code to EXPORT HS.

If the Digigraphic Interrupt Processor of the 1700 Package detects an error condition while attempting to process console input or output, it also sets a flag for the translator. The translator then types out one of the two reject messages given in the manual referenced above and aborts the job in the same manner as given above for a console driver error.

1700 BASIC GRAPHICS PACKAGE

The 1700 Basic Graphics Package contains a set of graphics routines and a queue handler to process lightpen/keyboard picks and save tracking-cross positions.

The functions of the 1700 Basic Graphics Package routines are similar to those of the graphics utilities and graphics hardware interface routines of the 6000 Basic Graphics Package. The calling statements for both sets of routines are identical; two Packages are used solely to prevent tying up the 6000 series computer with the detail work necessary to service a display console.

The applications programmer is concerned only with the 6000 Basic Graphics Package. He writes his programs in parametric form, and the 6000 Package then passes these parameters (via EXPORT/IMPORT HS and the buffer translator) to the 1700 Basic Graphics Package, which uses the data to actually drive the cathode-ray tube of its associated graphics console.

Specific information regarding the functions of the 1700 Package routines is beyond the scope of this manual.

SYSTEM EXPANSION

The Interactive Graphics System can be expanded by the addition of routines to the graphics utilities library of the 6000 Basic Graphics Package (Section 6). Although such additions could be made without corresponding changes in the 1700 Package, the efficiency of the system increases by the addition of a corresponding 1700 Package routine for each routine added to the 6000 Package. This approach simplifies 1700 error processing.

Additions to the two Packages can be made without changes in any of the other parts of the system software.

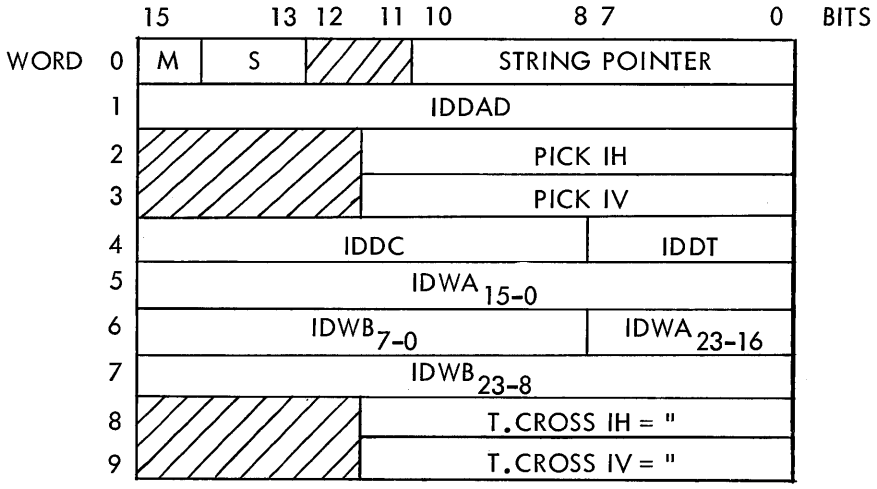
The associative address table (AAT) is currently dimensioned at 400 words. Each display item or macro requires two words of storage in this table. One word contains the IDDAD and the other contains the item's location in memory. IDDADs and locations of 200 display items or macros can now be stored in the AAT, but this table in the 1744 can be dimensioned to store more information. If the table is dimensioned larger, the maximum number of items and macros that can be erased with GIERAS and GIMACE is also larger.

DISPLAY ITEMS AND PICK PROCESSING 5

Every item that the programmer creates on the display screen has identification information associated with it in the 1700 series computer's memory, as does every console input device that he wishes to have his program service. This information includes parameters from the 6000 Basic Graphics Package calls which the programmer uses to create and manipulate the item or to define the functions of the device. These parameters (and other pick processing information) are organized into a structure called a display item ID block.

DISPLAY ITEM ID BLOCK

The 1700 Basic Graphics Package maintains a buffer of the item ID blocks created by the programmer (see Section 6) as shown in Figure 5-1.



M = 1, Item Being Marked (blinked when picked)

S = 1, Single Pick Type Item

S = 2, String Pick Type Item

S = 3, Button Pick Type Item

" = Tracking Cross Coordinates (for a Button only)

Figure 5-1. Display Item ID Block in 1700

The ID block is the basis of all graphics input processing. The four ID quantities IDDT, IDDC, IDWA, and IDWB are defined and used by the programmer. The display item type code IDDT is also used by the queue handler and the 1700 interrupt processor mask comparison routines (see GIMASK, Section 6).

The IDWA/IDWB of a light button would normally contain the name of a task to be called by the application executive AETSKR routine; the task name is left-justified beginning in IDWA. The IDWA/IDWB of a graphic figure would contain a data bead address (see Data Handler, Section 6) as its last five characters.

The contents of IDDT and IDDC cannot exceed 8 bits (377B) each; IDWA and IDWB cannot exceed 24 bits (77777777B) each. IDDT = 0 is reserved for alphanumeric input only.

ID blocks may be associated with other graphics input devices and with the items on the display. These are:

- The console function keyboard
- An alphanumeric end-of-message character
- The switch on the lightpen
- The pick of some display item of a particular type. This results in two ID blocks being queued, for example, a regular display item may also be conditioned to act as a button (see GIPBUT, Section 6).

To have an ID block from a device input to the application program, the IDDT of the device must classify it as one of the three types of pick information processed by the queue handler:

- Single pick information
- String pick information
- Button pick information

QUEUE HANDLER

Since the console operator will get ahead of the application program's execution, it is necessary to have a means of allowing the operator to use the lightpen, keyboard, and tracking-cross at his own speed while still enabling the graphics software to keep track of the picks and tracking-cross coordinates for later use by the application program. The queueing mechanism by which this is accomplished reduces the time a console operator must wait between requests.

PICK TYPES

The picks made by the console operator are queued as four types of ID blocks before being passed to the application program:

- Single pick type – only the copy of the ID block for the latest single pick display item chosen is kept in the queue, regardless of how many such items are picked.
- String pick type – one copy of a string pick display item ID block is kept in the queue for each time such an item is picked.
- Alphanumeric type – includes alphanumeric characters picked by either the lightpen or a keyboard key; queued in the same manner as a string pick type.
- Button pick type – one copy of the ID block for a light-button is kept in the queue for each time such an item is picked. The button pick ID is similar to the string pick ID except that a button pick may reactivate an idle application task.

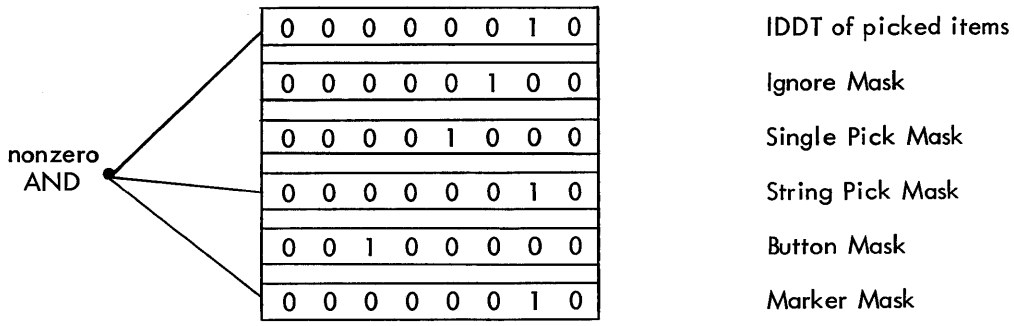
The single pick ID block and the string pick ID block are associated with the button pick ID block, and the button ID block may contain tracking-cross coordinates along with other ID information.

QUEUE HANDLER FUNCTIONS

When the 1700 Basic Graphics Package interrupt processor detects a lightpen or keyboard pick, it turns control of the 1700 series computer over to the queue handler. The queue handler then performs three actions:

1. Does an ID read of display memory to determine which item has been picked.
2. Logical ANDs the IDDT of the pick ID block with the set of ID processor masks (see GIMASK).
3. Performs the queue operation specified by the result of the ANDing.

If the logical AND of the IDDT and the set of masks is nonzero, the queue handler will place the ID block of the picked item on the end of the appropriate queue string. For example, after the AND of the following IDDT and mask values, the ID block involved is placed at the end of the set of queued string picks and the blink byte in the reset sequence is complemented; that is, a nonblinking item will blink and a blinking item will no longer blink.



In each of the following cases, the ID read processing differs from that of a normal lightpen strike. However, steps 2 and 3 above remain the same:

- If GILPKY has been called and a lightpen switch interrupt occurs, the queue handler will read the assigned ID block from a table in 1700 memory.
- If GIKYBD has been called and a keyboard interrupt occurs, a 1700 memory ID read will be performed.
- If GIEOM has been called and an EOM key press or an EOM font pick causes the interrupt, a 1700 memory ID read will be performed.
- If GIPBUT has been called and a prime button pick causes an interrupt, both a 1700 memory and a 1744 display memory ID read will be performed.

The queue handler also retrieves ID blocks from the FETCH queue (see below) when they are requested by a 6000 Basic Graphics Package GIBUT or application executive AETSKR call.

FETCH AND WAIT QUEUES

There are actually two separate queues maintained in the 1700's memory for each graphics console – the FETCH queue and the WAIT queue.

The WAIT queue serves as a temporary console input buffer in which to arrange and complete a set of picked ID blocks. The WAIT queue is not accessible to the application program; this prevents the program from receiving an incomplete set or string of pick ID blocks if it requests transfer of the blocks to the 6000 while the console user is still building a string or editing a set of queued blocks.

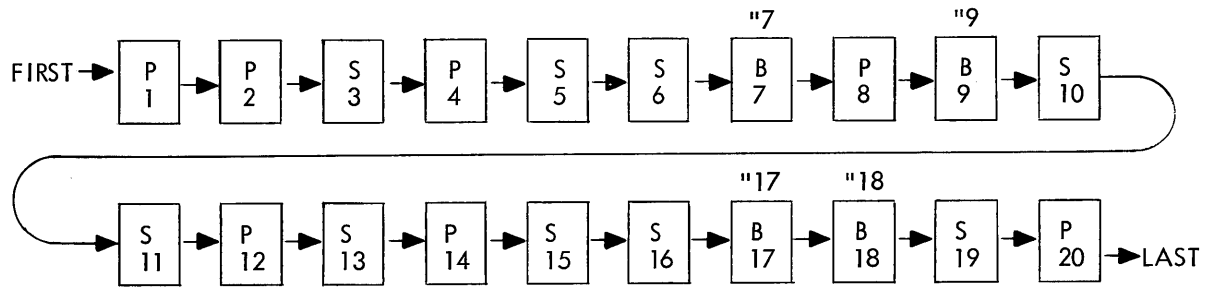
A button pick automatically transfers the ordered blocks from the WAIT queue to the FETCH queue. The blocks are then passed to the program in the 6000 series computer from the FETCH queue.

Whenever an item is erased from the display, both queues are scanned for a pick of the erased item. If the item is a single pick type or string pick type and is erased, the ID block is spliced out of the WAIT queue. If the erased item is in the FETCH queue as a button, the button ID block and its associated single pick and string pick ID blocks are all removed.

QUEUE MECHANISM OPERATION

The following set of diagrams illustrates the logical mechanism used by the queue handler to queue picks and buttons. Each square represents a core block of ID information, pointers, and coordinates. The queueing of ID blocks is controlled by the application program through the setting and clearing of type code masks (see GIMASK, Section 6).

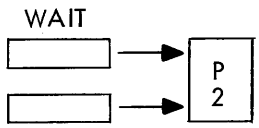
Time history is maintained as a simple queue:

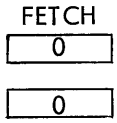


" Tracking cross coordinates
→ Flow of execution
P Single Pick ID
S String Pick ID
B Button Pick ID

Rule 1. Every time a single pick is made by the console operator, the ID block is placed at the beginning of the WAIT queue and replaces one already there.

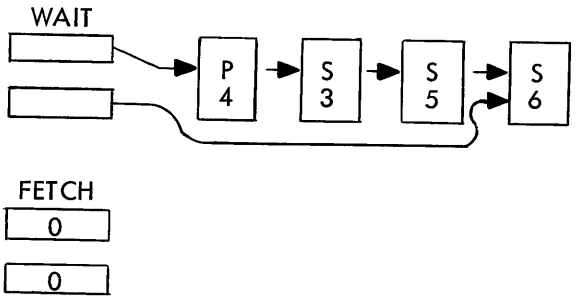
Representation after Single Picks 1 and 2:





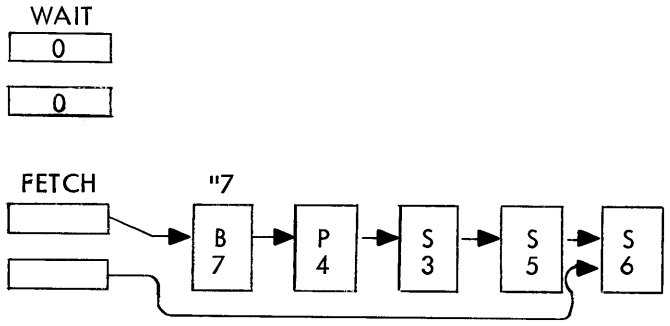
Rule 2. String pick ID blocks are always collected at the end of the WAIT queue and accumulate until the next button pick ID block is queued. There will never be more than one single pick ID block on the WAIT queue, but there may be 16 string pick ID blocks on the queue – 16 is the total number of blocks allowed in the WAIT queue.

Representation after String Picks 5 and 6:

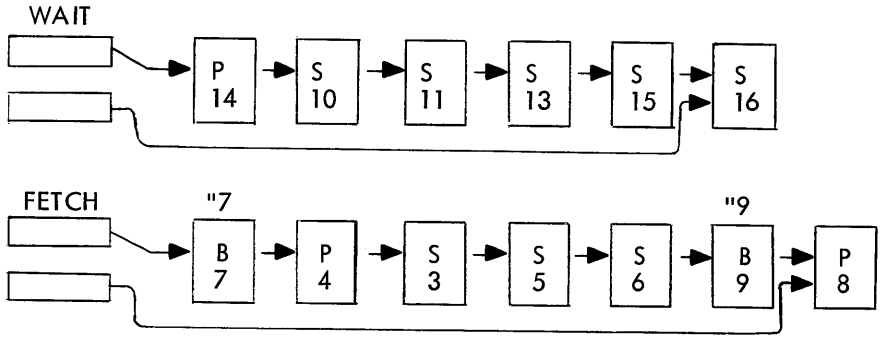


Rule 3. When the operator picks a button, the button pick ID block is always placed at the end of the FETCH queue followed by the contents of the WAIT queue (which is then cleared). Although the WAIT queue contains the ID block of only one single pick item, the FETCH queue can contain as many single pick ID blocks as button pick ID blocks.

Representation after Button Pick 7:



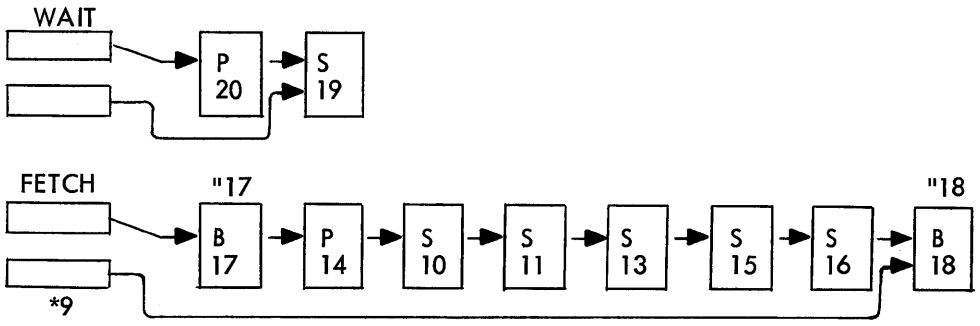
Representation after String Picks 15 and 16:



An editing feature permits the console operator to nullify a string pick, made since the last button pick, by picking the string pick item again.

If two string pick ID blocks are the same (corresponding IDDAD, IDDT, IDDC, IDWA, and IDWB identical) both blocks are removed. If string picks 11 and 16 were the same in the above example, block 11 would be spliced out of the WAIT queue and block 16 would not be saved. This feature allows limited editing of picked items without application program intervention. Single pick 12 is replaced by single pick 14 since the operator makes single pick 14 later and its block replaces any other single pick block that is at the beginning of the WAIT queue. This allows the operator to partially edit the WAIT queue by overriding a previous single pick.

Representation after Pick 20 is input and Button 9 is fetched by the application:



The tracking cross coordinates of each button ID are saved for application reference. The tracking cross coordinates fetched by GITCOF are those from the last button ID passed to the application program (*9 in the example immediately above). All string and single pick ID blocks associated with the button ID block are passed to the calling program at once.

ID blocks passed to the application program are always picked up from the start of the FETCH queue (block 17 in the last example above) and are passed only as the result of a specific request from the application. If a button fetch (CALL GIBUT) is requested at the time of the above representation, ID blocks 17, 14, 10, 11, 13, 15, and 16 are made available to the application.

Next, the button, string, and single masks are checked – in that order. The first nonzero product causes the designated queueing operation and a check of the marker mask. If the marker mask also causes a nonzero product, the queue handler performs the marking function on the item. The blink byte is reversed in order to change the appearance of the item while it is queued; its new appearance is maintained until the item is fetched.

As each ID block is removed from the queue, the marker bit is checked and the blink byte is restored to the status it had before it was queued. This notifies the console user that his pick has been sent to the 6000 series computer.

6000 SERIES COMPUTER PICK PROCESSING

The 6000 series computer receives ID blocks from the 1700 series computer only after sending it a GIBUT call or an equivalent call from the application executive. Only the first button ID block and any single or string pick blocks associated with it in the FETCH queue are sent to the EXPORT program in the 6000 series computer.

The ID blocks are all sent by EXPORT to the application executive area of the calling graphics job, where GIBUT unpacks the button ID block information and stores it for later use by the 6000 Basic Graphics Package AELBUT and GITCOF routines. The other ID blocks are stored for later use by the GIFID and GIDISP routines.

Each time a button pick is fetched from the 1700, the new ID blocks are written over the blocks stored in the 6000 by the previous fetch. The queue handler masking procedure is ordered. The ignore mask is ANDed with IDDT first. If the product is nonzero, no further action is taken.

6000 BASIC GRAPHICS PACKAGE 6

The 6000 Basic Graphics Package is a set of subroutines, written in COMPASS assembly language, designed to provide an interface between the applications programmer and the graphics hardware. The Package coexists with SCOPE; an applications programmer has full access to both.

This Basic Graphics Package has four main functions: to provide the ability to manipulate display items, to control light-buttons, to input and output alphanumeric data, and to supply the necessary tools for creating and handling a data structure.

Using the Basic Graphics Package routines, the simplest application program can send display items to the consoles. These display items are described in a language one level higher than the standard display language. For instance, a circle in display language is a stream of ΔXs and ΔYs; however, the Basic Graphics Package, using the 6000 series computer, describes a circle in parameter form. The 1700 has the ability to convert this parameterized data into display language by using the 1700 Basic Graphics Package (see Section 4).

ROUTINE TYPES

The 6000 Basic Graphics Package routines are divided into four categories:

- Graphics hardware interface
- Application executive
- Graphics utilities
- Data handler

GRAPHICS HARDWARE INTERFACE

The graphics hardware interface is a set of library subroutines that permit application program control of the display hardware. The functions performed by the graphics interface define the graphics capabilities available to a user. The interface includes routines to edit the display buffer display items, control lightpen and keyboard inputs, control lightpen tracking, and collect alphanumeric text input. All interface routine names begin with GI.

APPLICATION EXECUTIVE

The application executive controls the residence, sequencing, and execution of tasks; it includes the equivalents of SCHEDR, GIBUT, and GIABRT.

The executive is written as a single, eight-part program called AEXEC. When a programmer uses AEXEC as part of his zero-level overlay, his subsequent calls to AETSKC and AETSKR in any task overlay result in calls to the appropriate part of the AEXEC program.

FUNCTIONS OF AEXEC

AEXEC is entered as a FORTRAN subroutine from the application program's zero-level overlay, using a CALL AEXEC card (see Section 2), during both the file creation and execution runs of the job.

AEXEC first reads the file name parameter cards in the application program's next data record.

If the data record contains two cards, AEXEC

- Writes the graphics task COMMON file name (from the first card) in RA+3 of the program's current control point area.
- Writes the overlay source file name (from the second card) in RA+4
- Terminates the LGO portion of the job so that AEFILE can create the program's graphics COMMON file

If AEXEC finds only one card in the first data record, it assumes that the job is to be executed during the current run. AEXEC then

- Opens the task file named on the card
- Reads the task directory pointer to determine the amount of central memory needed to load the longest overlay in the task file
- Changes the field length
- Calls the Scheduler to assign the program to a graphics control point

The Scheduler then rolls out the job. When the Scheduler rolls the job back in, AEXEC reads the first record of the task file into central memory. Control of the central processor is then transferred to the task in that record.

AEXEC is again entered when an AETSKC call occurs; it then locates the requested task within the task file, reads it into central memory, and transfers control to it.

When an AETSKR call occurs, AEXEC requests a button fetch from the 1700 FETCH queue and waits until one has been returned. When a button pick type ID block (and its associated string and single pick blocks) is returned, control of the central processor is turned over to the task overlay named in the IDWA and IDWB parameters of the button's ID block.

AEXEC also contains an abort processor that is entered any time a 6000 Basic Graphics Package routine produces an error message. The abort processor enters all diagnostic messages supplied to it in the system dayfile; the processor aborts the application job only if a fatal error or a GIABRT call has occurred.

GRAPHICS UTILITIES

The graphics utilities are an expandable library of subroutines for general graphics applications. Included as graphics utilities are routines to frame-scissor graphic figures, generate graphic figure descriptions, and collect figure descriptions for display. The utilities routine names supplied with the 6000 Basic Graphics Package all begin with GU.

DATA HANDLER

The data handler is a set of routines that optimize access to mass storage and perform in-core list processing. The handler permits an application programmer to efficiently create and manipulate his own unique data structure. The form of data organization used is a plex data structure.

PLEX DATA STRUCTURES

Graphics interaction places stringent demands upon the application programmer in the allocation and handling of data. In general, graphics application data is completely random in the order of its manipulation and in the amounts of each data type stored. Conventional allocation and management schemes, such as FORTRAN arrays or card image files, are usually inappropriate and inefficient.

A concept of storage management has been defined† that meets all the requirements of interactive applications. The concept, called the modelling plex, involves the data, data structures, and data manipulating algorithms required to represent the physical actions required of the application. The requirements of the data and algorithms are determined by the needs of the application on one hand and by the data structure on the other.

---

† Douglas T. Ross, AED-O Programming Manual, Section 2.2 Data Structure Language, Preliminary Release No. 2, MIL-ESL, October 1964.

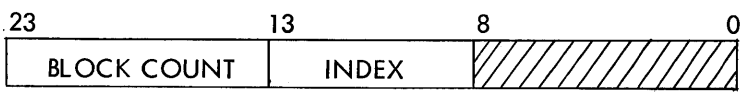
A plex data structure is the most general form in a broad class of data management techniques called list structuring. In a plex data structure, all data is contained in variable length beads of contiguous computer words. The length, format, and data content of any bead is completely under control of the application programs.

The data handler provides a pool of empty beads (free storage) from which the application may obtain new beads and to which it can return those no longer needed. Each bead has a unique addressing parameter (IBEAD) that is supplied by the system and used by the application programs as data. This bead address is used for referencing the data within the bead and may be used as data within other beads as a pointer to specify related information. In general, a plex data structure contains a greater number of pointers than do more conventional storage techniques. (See Figure 6-1 for typical bead arrangements. The arrows in the diagram represent a head address within the bead at the tail of the arrow, which points to – contains the address of – the bead at the head of the arrow.)

For a specific application, it is most efficient to include only the beads and pointers needed. A formal structure that includes all possible relationships of a rigid class introduces inefficiencies that cannot be tolerated in an interactive system.

For a simple example of a plex data structue, consider the dynamic parts of an automobile engine: the crankshaft, the connecting rods, and pistons. Each bead of the representation contains the needed information about a particular item. Each bead of a particular type has exactly the same length and format; while different data values (i.e., the mass parameter in each connecting rod bead – CR in the example) would appear in the same positions within the bead, they would reflect the actual mass of the particular connecting rod. Note that by proper design of the plex data structure, the calculating algorithms may be quite independent of the actual representational model. In the example (Figure 6-2), a 12-cylinder engine could be handled with the same structure and programs by allowing the connecting rod string of beads to be of variable length. By convention, the initial bead address in a string of beads is called the state variable of the string, and the last bead of the string holds a zero string pointer. In a plex data structure any number of strings may be passed through a bead.

Beads are floating within blocks. The bead address IBEAD contains the block count and an index to an array of pointers within the block as follows:



For use as a string pointer, the location within the bead of the next pointer (hook) may be placed in the cross-hatched area.

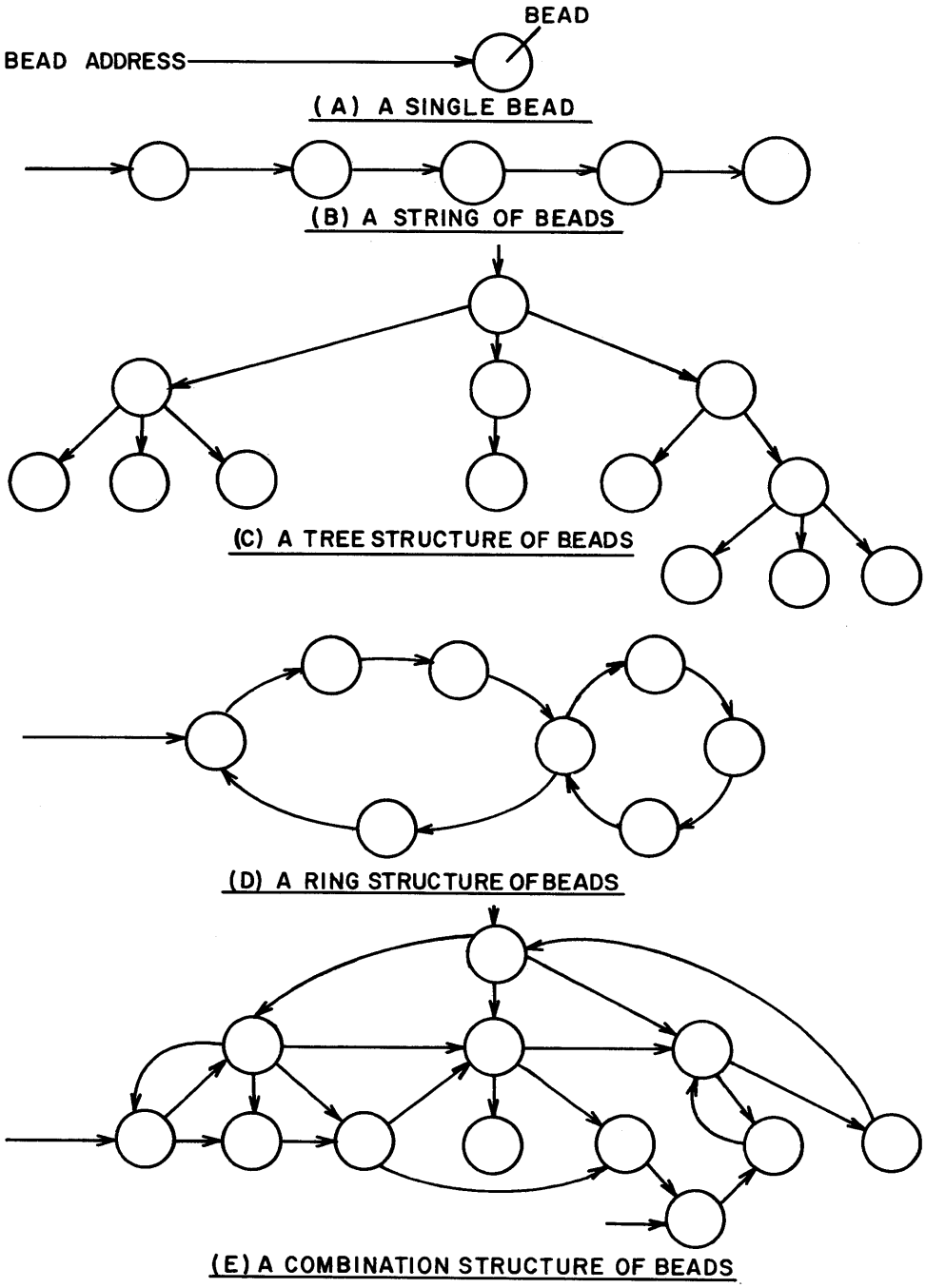


Figure 6-1. Typical Bead Arrangement

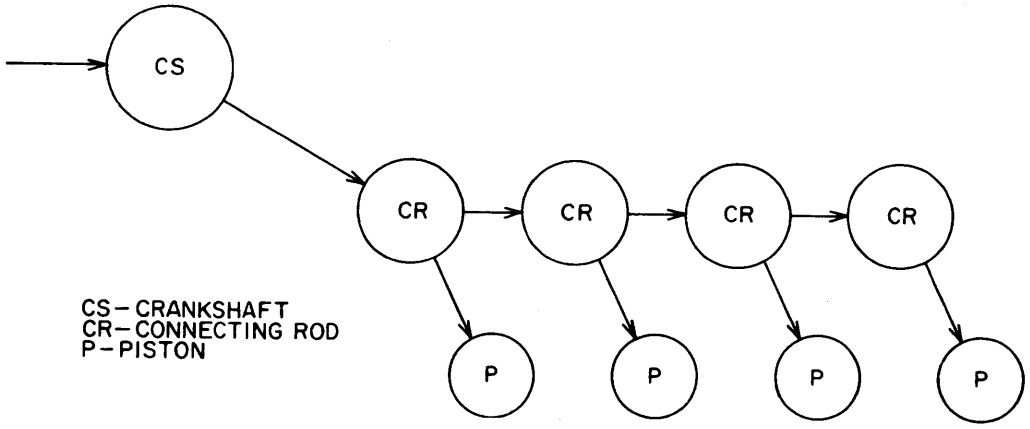


Figure 6-2. Four Cylinder Engine

The 9-bit hook limits string pointers to the first 511 locations in a bead. The data handler accepts full 24-bit addresses as a bead address and will ignore the low order 9 bits on all but string operations.

The data handler allows simple FORTRAN programming string operations. Figure 6-3 is an example of list structuring. Hooks are shown with a broken line.

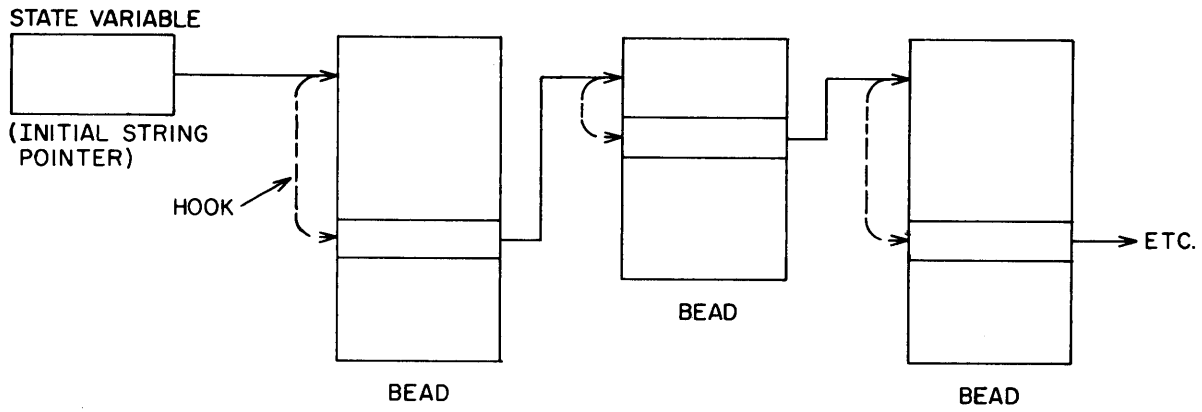


Figure 6-3. List Structure Example

BLOCK STRUCTURE AND ACCESS

Data resides in standard SCOPE random files (in logical blocks). Specification of the block length is an application programming function (see DMINIT, page 6-58).

MAXIMUM DATA UNIT SIZES

The addressing scheme used by the data handler limits the size of files, beads, and blocks. Bead and block numbers are decimal. Word numbers and empty space values are octal in a data handler dump. The limits are:

- Maximum number of blocks per file 1,023
- Maximum number of beads per block 31
- Maximum number of words per bead 1,048,575 (2^{20}-1)

The number of words in a block depends on the device the system uses for mass storage. The block size can be specified by the programmer in his DMINIT calls; if the programmer omits the block size parameter from his calls, an installation parameter is used.

Another installation parameter (MAXBLKSP $\simeq$ 40,000) defines the maximum number of words that can be allocated for the in-core blocks. Since the data handler requires at least two in-core blocks to function efficiently, this actually limits the maximum block size to MAXBLKSP/2 central memory words.

GENERAL SUMMARY

- Components
  - are bit or word spaces
  - contain values
  - reside in beads
  - are addressed by a unique code
- Beads
  - are contiguous computer words
  - contain components
  - reside in blocks
  - are addressed by a unique bit pattern
- Blocks
  - are mass storage logical blocks
  - contain beads
  - reside on mass storage and in core as IFILE
  - are addressed by count
- All data handler routine names begin with DM

ADDITIONAL INFORMATION

This section describes the implementation of the data handler and is not needed by the application programmer to begin writing data handler programs. It is included with the data handler description so some insight into the system and choice of installation parameters may be gained.

The data handler maintains in-core duplicates of those ID blocks needed to allow efficient access to the data. The number of in-core blocks is specified by the application programmer and may be changed dynamically.

The in-core blocks reside as IFILE in the application job's global data area of the graphics control point. IFILE is rolled out and in automatically as a local file with the program.

Data is confined to beads within the blocks. Data handler subroutines are provided so that the application programmer can create and destroy beads as desired.

The data content of a bead is broken into components. A component is a specific bit or word space within a bead and has a unique address code. Data handler subroutines are provided to set or fetch values of components of specified beads.

The application program does not reference mass storage blocks directly; therefore, the block accessing process and format details are not a programming function.

The data handler provides efficient automatic access to the mass storage blocks through an algorithmic optimization procedure. Three decision parameters, kept for each in-core block, are used in the algorithm:

- UC Usage count of the current in-core block from the time of the last decision
- ES Amount of empty space within the in-core block
- WE Indicator of in-core block content change

Three additional values are used to modify the decision parameters:

- NB Number of blocks in core
- TUC Total usage count of the data handler from the time of the last decision
- BS Block size

The decision process involves finding the in-core block with the minimum or maximum value of the algorithm:

$$B \times \frac{UC}{TUC} + C \times \frac{ES}{BS \times NB} + D \times WE$$

The weighting factors B, C, and D are real numbers between 0 and 100 with a combined sum of 100. Typical values are B = 15, C = 15, and D = 70. These factors are chosen by the installation. If there is little or no new bead allocation and few data modifications, B is greater than C, otherwise, C is greater than B. D should be greater than 50. The algorithm is used in each decision to optimize use of the in-core block space (IFILE) and minimize mass storage references.

Figure 6-4 shows the structure of a data handler file block as it is stored in central memory.

The first word of the duplicate block contains the amount of empty space in the block and a pointer to the empty space. When a bead is entered into a block, it is associated with a bead pointer to which the bead address is related. The bead pointer is fixed in a block but its contents can vary, since beads are floating in the block. The beads in a block move when a bead is deleted in a block and the data handler closes up any space previously occupied by a bead to maintain empty space contiguity. The bead address IBEAD, however, remains inviolate for the life of a bead.

Beads are entered into a block starting from the bottom of the empty space. If a bead is too large to fit in a single block, it is continued onto as many other blocks as necessary. (The continuation process is designed to minimize the number of blocks per bead.) A continuation bead address is added to the end of the bead segment to point to the next segment of the bead. Bit F is set in the bead pointer to indicate that its segment is the first segment of a bead. Bit G indicates the continuation of a bead.

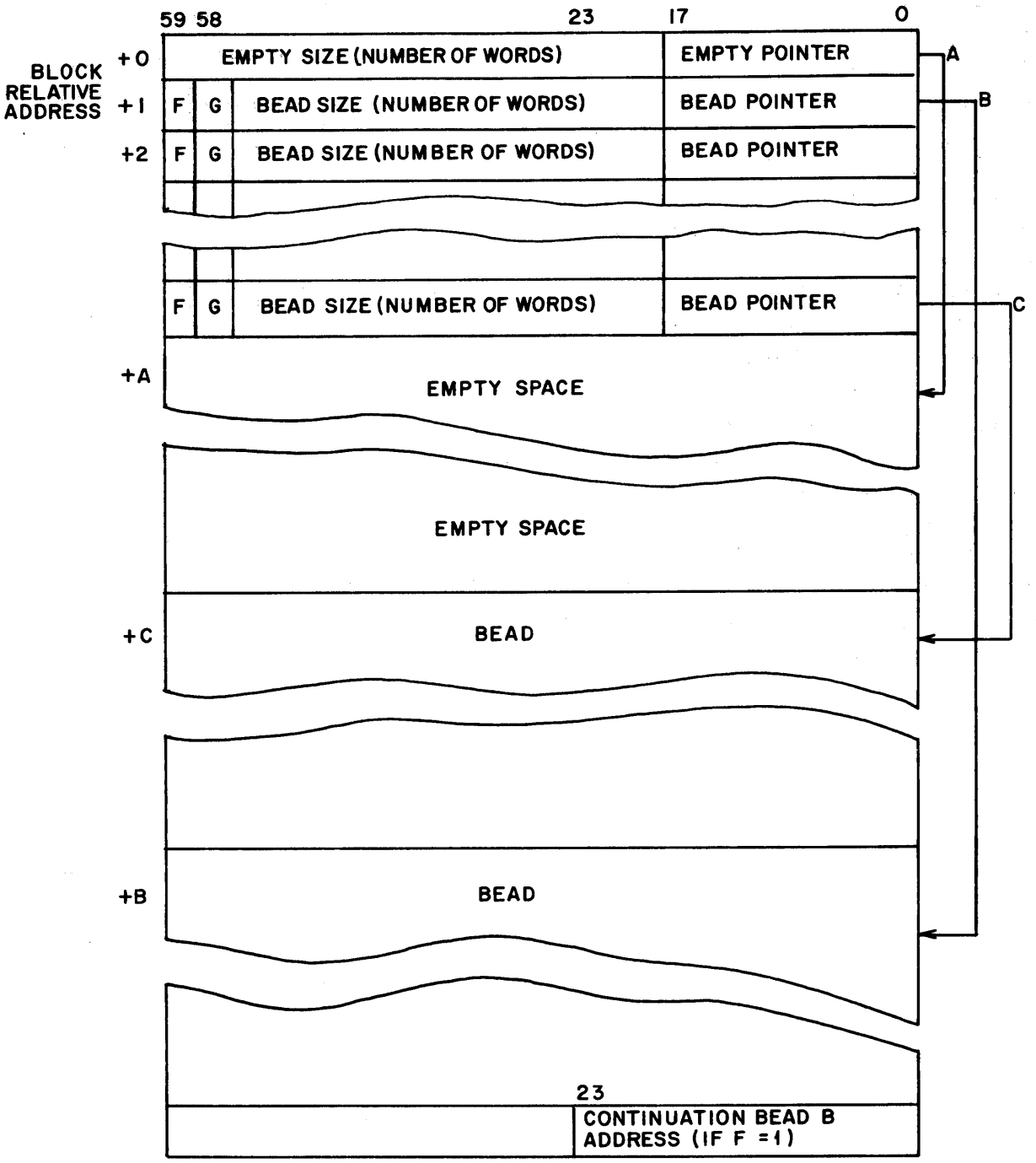


Figure 6-4. Data Handler File Block Structure

ASSOCIATIVE ADDRESSES

The Basic Graphics Package also does internal bookkeeping; the bookkeeping operation is controlled by bit patterns called associative addresses that are supplied to or by the application programmer. The major associative addresses are:

- The console address NCON that is associated with the particular console(s) assigned to an application program. More than one console address may be used by a program to control several consoles at once. NCON is a decimal or two-digit octal number; the first octal digit is the number of the 1700 to which the console is connected (0-3), and the second octal digit is the number of the console itself (1-6). Thus, NCON can vary from 01_8 to 36_8 (excluding 07_8, 10_8, 17_8, 20_8, 27_8, and 30_8) or from 1_{10} to 30_{10} (excluding 7, 8, 15, 16, 23, and 24). NCON values are defined by the installation and supplied by the programmer.
- The display item address IDDAD that is associated with a particular graphic item being displayed. IDDAD is used for editing functions and is the relative address of the item within a table containing the actual 1744 display addresses of all such items (see Section 4).
- The macro address MAD, which serves the same function for macro item information as IDDAD, serves for display items.
- The bead address IBEAD, which is associated with a particular set of contiguous computer words supplied by the data handler. The bead address is used for all references within the bead and is defined as the relative address of the first word of the bead within the IFILE.
- The application task name, NAME, is used to control program execution. A typical program may consist of over 100 individual tasks or overlays, each performing a function(s) or a computation(s). Each task resides in mass storage and is randomly accessible. NAME is used by the application executive to associate the task with its actual location in mass storage (see Task Directory, Section 2).

PROGRAMMING CONVENTIONS

To reduce application programming errors, the following calling sequence conventions are imposed on all Basic Graphics Package routines:

- All externally supplied values are passed between the routines as parameters in the calling statements. No specific COMMON configurations are imposed on the applications programmer.

- Needed values are specified as separate calling statement parameters. The code inefficiency of loading and unloading formatted arrays justifies the use of the longer calling sequences that are produced by this convention.
- Separate subroutines are provided for each function of the Package. Code parameters are not used for function selection.

The following is a general introduction to the major parameters which appear in the calling sequences of the 6000 Basic Graphics Package routines. Although this subsection is designed as a reference when questions arise about the format of frequently used parameters, it will be useful for the user to read it for a general understanding of parameter usage and formats before reading the rest of the section. Exceptions to these general conventions are mentioned in the subsection concerning a specific routine.

This introduction assumes familiarity with IGS items and queue handling (Section 5), macros (Section 4), and the general features of the hardware.

ID BLOCK PARAMETERS

IDDT

IDDT is a FORTRAN variable representing an 8-bit, unsigned, right-justified quantity. If IDDT is larger than $2^8 - 1$, only the lower eight bits are used; when IDDT is a result parameter, only the lower eight bits are returned.

For the special meaning of IDDT as the display item type, see Queue Handler (Section 5).

IDDC

IDDC is a FORTRAN variable representing an 8-bit, unsigned, right-justified quantity. If IDDC is larger than $2^8 - 1$, only the lower eight bits are used; when IDDC is a result parameter, only the lower eight bits are returned.

IDWA

IDWA is a FORTRAN variable representing a 24-bit, unsigned, right-justified quantity. If IDWA is larger than $2^{24} - 1$ (more than four right-justified alphanumeric characters), only the lower 24 bits are used; when IDWA is a result parameter, only the lower 24 bits are returned. The content of this variable is arbitrary (except when used for task calling) and may be used by the graphics programmer to store such information as bead addresses and IDDADs for use when a button is picked.

IDWB

IDWB has the same format as IDWA. If IDWA is less than four characters, IDWB may be used to store information such as bead addresses and IDDADs for use when a button is picked. For the special use of IDWA and IDWB in graphics buttons that call tasks, see AETSKC.

DISPLAY GRID COORDINATES

Display grid coordinates are treated by the 6000 Basic Graphics Package as integer parameters within the range 2047. If a coordinate is greater than 2^{11} -1, only the lower 12-bits are used. (Note the exception in the tracking-cross routines, Section 6, and Appendix B). When relative coordinates are used, it is a temptation to reference absolute coordinates greater than 2047. See the User's Guide section on Coordinates — Absolute and Relative.

When a coordinate is a result parameter, it is returned as a 60-bit, sign-extended integer within the range -2047 to +2047 decimal.

Coordinates are variously represented by IH, IV, IH1, IV1, IH2, IV2. Note that IHC and IVC (the coordinates of the center of an arc in GUARCG) are not necessarily on the 274 console surface. GUARCG allows center coordinates in the range -32, 767 to +32, 767 (-2^{15} -1 to $+2^{15}$ -1). If IHC or IVC is outside that range, only the lower 16 bits are used.

The coordinates in the scissoring routines are not output to the console and are not truncated (see GULINE and GUARC).

DISPLAY ITEM, MACRO AND BEAD ADDRESSES

IDDAD and MAD are FORTRAN variables representing 21-bit, right-justified, system-generated associative addresses, which are used by the application programmer to address an item or a macro (respectively). The associative address is a result parameter returned to the user when an item or macro is created.

IBEAD is a system-generated bead address used by the programmer to address a bead within an application's data handler file. IBEAD is a result parameter generated by the data handler at the programmer's request before data is set into a bead.

NCON ADDRESS

NCON is the associative address for the graphics console. It consists of two octal digits. The upper digit runs from 0 to 3 and represents the 1700 being addressed. The lower digit ranges between 1 and 6 representing the console on that 1700. For 1700 zero, the decimal console addresses are 1 to 6; for 1700 one, 9 to 14; for 1700 two, 17 to 22; and for 1700 three, 25 to 30.

IBEAM ADDRESS

IBEAM, when it appears in a parameter list, governs whether the beam is on or off for a particular line segment. If IBEAM is equal to ±0 (beam off), the entire line segment will be generated with beam off. If IBEAM is equal to ±1 (beam on), the beam will be on in accordance with the bit pattern in ISTYLE.

ISTYLE ADDRESS

ISTYLE is a FORTRAN variable representing a right-justified 12-bit byte. Each one in ISTYLE corresponds to a portion of the line segment generated with the beam on. Each zero corresponds to a portion generated with the beam off. ISTYLE is most conveniently represented as an octal number (for example, 525 B corresponds to bit pattern 101010101010). If IBEAM for the line segment (see above) is zero, the line segment will be generated – independent of the ISTYLE value – with the beam off.

ICODE ADDRESS

ICODE is a FORTRAN variable representing a right-justified 7-bit byte which governs light-pen sensitivity, blink, and brightness of a graphics display item. ICODE appears in calling sequences where a reset sequence is generated (GURSET or GICOPY) or altered (GIMOVE). See these routines for the format of ICODE. Because it is a bit pattern, ICODE is most conveniently represented as an octal number.

The other parameters appearing in 6000 Basic Graphics Package routines are described in other portions of the manual.

PARAMETERS

In the following list of routines, the required parameters are underlined. Parameters following the last one underlined are optional. The parameter sequences may be terminated by a minus zero parameter or a right parenthesis. For the meaning of the truncated parameter list, see the individual routine description.

AELBUT (IDDT, IDDC, IDWA, IDWB, IH, IV)

DMINIT (IFILE, NBLK, NBSIZE)

DMRLBD (IBEAD,, $IBEAD_2$, ... , $IBEAD_n$)

GIBUT (IR, NCON, IDDT, IDDC, IDWA, IDWB, IH, IV)

GICOPY (IDDADI, NCON, IH, IV, ICODE, IDDAD, IDDT, IDDC, IDWA, IDWB)

GIDISP (NCON, IBUF, NBYTE, IDDAD, IDDT, IDDC, IDWA, IDWB)

GIEOM (NCON, IBCD, IDDT, IDDC, IDWA, IDWB)

GIERAS ($IDDAD_1$, $IDDAD_2$, ... , $IDDAD_{36}$)

GIFID (NCON, IDDT, IDDC, IDWA, IDWB, IH, IV)

GIFSID (NCON, N, IDDT, IDDC, IDWA, IDWB, IH, IV)

GIKYBD (NCON, IDDT, IDDC, IDWA, IDWB)

GILPKY (NCON, IDDT, IDDC, IDWA, IDWB)

GIMACE (MAD_1, MAD_2, ... , MAD_{36})

GIMOVE (IH, IV, ICODE, IDDAD, IDDT, IDDC, IDWA, IDWB)

GIPBUT (NCON, IIDDT, IDDT, IDDC, IDWA, IDWB)

GITCON (NCON, IH, IV)

All parameters of the following routines are required in every case where they are employed.

AERTRN

AETSKC (NAME)

AETSKR

DMDMP

DMFLSH

DMGET (ICOMP, IBEAD, VAL)

DMGTBD (N, IBEAD)

DMSET (ICOMP, IBEAD, VAL)

GFONTA (NCON, IH, IV, IDADA, IDADN)

GFONTN (NCON, IH, IV, IDDAD)

GIABRT (NCON, IALF, NC)

GIANE (NCON, NC, IBUF)

GIANS (NCON, NC, IH, IV)

GICLR (NCON)

GICNJB (NCON)

GICNRL (NCON)

GIMAC (NCON, IBUF, NBYTE, MAD)

GIMASK (NCON, IDDTC, IDDTS, IMASK)

GIPLOT (NCON, IBUF, NBYTE, IDENT, ITYPE)

GITCOF (NCON, IH, IV)

GITIMV (NCON, IDDAD)

GITMMV (NCON, MAD)

GUAN (IBCD, NC, IBUF, NBYTE, MBYTE)

GUARC (IHCEN, IVCEN, IHCOR, IVCOR, HC, VC, H1, V1, H2, V2, KSHOW, IHC, IVC, IH1, IV1, IH2, IV2)

GUARCG (KSHOW, IHC, IVC, IH1, IV1, IH2, IV2, ISTYLE, IBUF, NBYTE, MBYTE)

GUBYTE (IBYTE, L, IBUF, NBYTE, MBYTE)

GULINE (IHCEN, IVCEN, IHCOR, IVCOR, H1, V1, H2, V2, KSHOW, IH1, IV1, IH2, IV2)

GUMACG (MAD, L, IBUF, NBYTE, MBYTE)

GURSET (IH, IV, ICODE, IBUF, NBYTE, MBYTE)

GUSEG (IH, IV, IBEAM)

GUSEGA (IH, IV, IBEAM, N, ISTYLE, IBUF, NBYTE, MBYTE)

GUSEGI (IH1, IV1, ISTYLE, IBUF, NBYTE, MBYTE)

GUSEGS (IH1, IV1, IH2, IV2, IBEAM, ISTYLE, IBUF, NBYTE, MBYTE)

NOTE

Placing an extra parameter in the calling sequence is just as fatal as omitting a required parameter.

For example:

Correct

```
CALL   GIDISP (NCON, IBUF, NBYTE, IDDAD, IDDT, IDDC, -0)
CALL   GIDISP (NCON, IBUF, NBYTE, IDDAD, IDDT, IDDC)
```

Incorrect

```
CALL   GIDISP (NCON, IBUF, NBYTE, -0)
CALL   GIDISP (NCON, IBUF, NBYTE, IDDAD, IDDT, IDDC, IDWA, IDWB, -0)
```

Parameter lists which are too short or too long will not be specifically diagnosed; however, they will cause mode errors or store meaningless data in a data area or the entry address of an IGS system subroutine. Since these problems may be hard to diagnose, care should be exercised in conforming to required conventions.

Care should also be exercised in calling subroutines whose calling sequence contain return parameters. For example, IDDAD is the fourth parameter in the calling sequence to GIDISP; the display associative address is returned to this variable location.

Example:

```
CALL   GIDISP (NCON, IBUF, NBYTE, -0)
```

This example will cause the destruction of the contents of the location represented by the literal -0.

SUMMARY OF USER FORTRAN-CALLABLE ROUTINES

These routines are all part of the 6000 Basic Graphics Package; all perform parameter checking functions and may cause the system software to abort an application program if its parameters are illegal. If a display item buffer exceeds the maximum length of the EXPORT/IMPORT HS input or output buffers (320 12-bit bytes each), it is considered a fatal error. Diagnostic messages for these and other errors are given in Appendix B.

All of the Package routines can be accessed through standard FORTRAN CALL statements. Unless otherwise specified, all parameters in the statements are passed to the routines as programmer-supplied arguments; integers may be either decimal or Boolean octal in form. The programmer may choose his own parameter names, although use of the names supplied in this manual would eliminate confusion when interpreting the diagnostic messages listed in Appendix B. Because many of these diagnostics contain the parameter names used in this manual, all parameter names throughout the book have been capitalized – a convention normally used to indicate only those words or letters whose presence is required by the system.

PROGRAM INITIATION

SCHEDR rolls the program out to mass storage so that it can undergo real-time scheduling and be rolled into a graphics control point for execution. A call to this routine must precede all Graphics Interface calls if the application executive AEXEC program is not used (see Appendix H). When the SCHEDR call is made, the system Scheduler program rolls out the entire control point and all associated files.

When AEXEC is used, a call to SCHEDR serves no useful function.

Call Statement Format:

```
CALL SCHEDR
```

PROGRAM CONSOLE CONTROL

The subroutines GICNJB and GICNRL flag the 1700 Basic Graphics Package interrupt processor to establish or break the correspondence between a console and the calling job. The two routines also perform such housekeeping duties as clearing the 1744 display buffer and resetting interrupt tables.

Good programming practice dictates that a call to the console release subroutine GICNRL be made before terminating the program. However, it is not mandatory to do so since a call to GICNJB from a later job (in the time sequence of job runs) will perform the same function.

The functions performed by GICNJB and GICNRL are console-oriented. Any task of any job may request initialization of the console-to-job correspondence for a particular console number. More than one console may be initialized for a job.

Once console-to-job correspondence is made, any task of that job may address that console. If a task addresses a console that has not been initialized through GICNJB for the job of that task (or if a task addresses a console that has been initialized for some other job), the task and its job will be aborted.

A console may be in one of three states with respect to a particular job:

1. Not attached to any job
2. Attached to some other job
3. Attached to a particular job

The purpose of GICNJB is to go from state 1 to state 3. The purpose of GICNRL is to go from state 3 to state 1.

GICNJB

This subroutine assigns a programmer-specified graphics console to the calling program and performs such initial clean-up duties as clearing the display buffer. GICNJB must be called before any other GI routines are called, or an NCON error will result.

GICNJB aborts the calling job if the console number, NCON, is invalid, or if the console is not available (i.e., has been declared out of service by the 1700 series computer operator or is assigned to another job). GICNJB clears any tables and ID processor masks that have been set. A call to GICNJB following a call to GICNRL can be used by the programmer to erase the console at the end of a job.

Call Statement Format:

```
CALL GICNJB (NCON)
```

NCON — Number of the graphics console that should be assigned to this job; only one console can be assigned through each call

NCON can easily be changed by loading data cards with the application program through either the remote or local card reader.

GICNRL

This routine releases the specified graphics console from the control of the calling job. GICNRL terminates internal display for console NCON and clears console-oriented tables kept by the 1700 Basic Graphics Package interrupt processor. The programmer should usually preceed a GICNRL call card with a FORTRAN STOP card.

Call Statement Format:

```
CALL GICNRL (NCON)
```

NCON — Console number; the same constraints apply here as to NCON in the GICNJB statement

PROGRAM TASK CONTROL

AETSKC and AETSKR establish the linkage between the application executive AEXEC program and/or individual tasks of the application job.

AETSKC

This routine can be called from the zero-level overlay, any task overlay, or from any subroutine within an overlay. A call to AETSKC causes the named task overlay to be loaded

into core memory from the graphics task COMMON file. AETSKC then turns control of the 6000 series computer over to the new task; there is no return from a call to AETSKC.

Call Statement Format:

```
CALL AETSKC (NAME)
```

NAME — Name of the task to be called; this is the 1 to 7 character identifier on the PROGRAM card at the beginning of each task overlay. The name in this call must be written in 6000 internal display code, left-justified within NAME, and blank or zero-filled.

AETSKR

An AETSKR call terminates execution of the current task, then performs the functions of AETSKC for the task overlay named in the IDWA and IDWB parameters of the next button pick type ID block in the 1700 FETCH queue.

AETSKR determines which task to load by requesting that a button pick type ID block be fetched from the 1700. If no button ID block is queued there, AETSKR waits until one is entered, then loads and executes the task indicated by the button picked. There is no return from a call to AETSKR.

If a STOP or END card is encountered within a task before a call to AETSKR or AETSKC occurs, the card will cause normal termination of the entire application job.

There is no console argument in the AETSKR calling sequence. AETSKR asks for a button from the graphics console number used as the argument of the last call to a GIBUT or GICNJB routine.

Call Statement Format:

```
CALL AETSKR
```

SPECIAL ID BLOCK ASSIGNMENT

ID blocks similar to those described in Section 6 can be assigned to various input devices at each console. These special ID blocks give the devices queuing and input significance that they would not otherwise possess.

One such block may be assigned to a console for each of the following:

- All of the buttons on the function keyboard
- The switch on the lightpen
- A specific alphanumeric character, which will be used to terminate the console's current alphanumeric input
- One display item that is not defined as a light-button but is to be treated as one

GIKYBD

GIKYBD associates an ID block similar to that of Figure 5-1 with the function keyboard of a particular graphics console. This block provides a means to examine the status of the keyboard's keys or to call a task overlay when a key is pressed.

Once GIKYBD has been called, a copy of the keyboard ID block is queued every time a keyboard key is pressed. Queueing is done according to the IDDT of the block.

Key status is contained in the IH and IV parameters of the block (see Table 3-1).

A GIKYBD call can also be used to change the ID parameters of an existing keyboard ID block.

GIKYBD cannot be used for a graphics console that is not equipped with a function keyboard.

Call Statement Format:

```
CALL GIKYBD (NCON, IDDT, IDDC, IDWA, IDWB)
```

NCON	Number of the console to which the block should be assigned; only one console can be referenced by each call
IDDT	ID type code; used to specify how the queue handler will treat the ID block
IDDC	ID code word; the contents assigned by the programmer can be $0 \leq IDDC \leq 2^8-1$
IDWA	ID information word A; contents are arbitrary unless block is referenced by an application executive routine (see GIDISP)
IDWB	ID information word B; contents are arbitrary unless block is referenced by an application executive routine (see GIDISP)

The ID block assigned to console NCON by GIKYBD contains the representation of the input parameters IDDT through IDWB.

Only one keyboard ID block can be associated with a particular console; if several calls are made to GIKYBD with the same NCON value, the parameters of the latest call will replace all of the parameters previously entered in the block.

GILPKY

GILPKY assigns an ID block to the switch on the lightpen of a particular graphics console. If GILPKY has been called, the effect of releasing the key on the pen is identical to the act of pointing to an item on the display; the ID block assigned to the key is processed by the queue handler as if it were the ID block of a display item. This allows the programmer to detect the use of the switch.

Call Statement Format:

```
CALL GILPKY (NCON, IDDT, IDDC, IDWA, IDWB)
```

NCON	Number of the console to which the block should be assigned; only one console can be referenced with each call
IDDT	ID type code; used to specify how the queue handler will treat the ID block
IDDC	ID code word; the contents assigned by the programmer can be $0 \leq \text{IDDC} \leq 2^8-1$
IDWA	ID information word A; contents are arbitrary unless the block is referenced by an application executive routine
IDWB	ID information word B; contents are arbitrary unless the block is referenced by an application executive routine

The ID block generated by a GILPKY call contains the representation of the input parameters IDDT through IDWB. If NCON is the only nonzero input parameter (or the only parameter given in the call), the existing ID block for the lightpen switch of console NCON will be removed from the 1700 computer's memory.

When the lightpen key is released, the copy of the ID block queued in the 1700 will contain the current H and V coordinates of the tracking cross in the IH and IV words – which are used for the coordinates of a lightpen pick in the ID block of a display item.

Only one lightpen key ID block can be associated with a particular console; if several calls are made to GILPKY with the same NCON value, the parameters of the latest call will replace all of the parameters previously entered in that block.

GIEOM

This routine assigns an ID block to a single alphanumeric character at a specified graphics console. The character may be part of the display font (excluding BKSP, SPC, and CLEAR) or a corresponding character on the alphanumeric keyboard. RETURN or alphanumeric characters which correspond to those in the alphanumeric font may be end-of-message characters. When the character associated with a GIEOM call is pressed (or picked, in the case of the display font) during an alphanumeric input operation, the ID block assigned to it is queued as if it were the ID block of a display item. This gives the programmer a means to detect an end-of-message condition.

An end-of-message character is displayed on the screen and returned through a GIANE call, like any other character.

Call Statement Format:

```
CALL GIEOM (NCON, IBCD, IDDT, IDDC, IDWA, IDWB)
```

NCON	Number of the console to which the ID block should be assigned; only one console can be referenced with each call
IBCD	A right justified display code character which is to act as an end-of-message indicator and to which the ID block should be assigned
IDDT	ID type code; used to specify how the queue handler will treat the ID block
IDDC	ID code word; the contents assigned by the programmer can be $0 \leq IDDC \leq 2^{8}-1$
IDWA	ID information word A; contents are arbitrary unless the ID block is referenced by an application executive routine
IDWB	ID information word B; contents are arbitrary unless the ID block is referenced by an application executive routine

The ID block created by a GIEOM call contains the representation of the input parameters IDDT through IDWB.

Only one end-of-message character ID block can be associated with a particular console; if several calls are made to GIEOM with the same NCON value, the parameters of the latest call will replace all of the parameters previously entered in the block.

GIPBUT

GIPBUT will create an ID block and a queue handler mask for a prime button at a particular console. Anything for which an ID block exists may be defined as a prime button, but the prime button ID information and its associated mask are usually used to allow a display item — not defined as a button pick type — to activate a task when picked. A display item which is a button pick type cannot be used as a prime button. In other words, IIDDT may not contain the button mask value. If it does, the item will be treated as a normal button pick item.

A prime button can have an IDDT which identifies it as a single pick item, a button pick item, a string pick item, or two of the three. A button, by contrast, can have only the IDDT that identifies it as a button pick item (marked or not marked).

There are two ID blocks for a prime button in memory, which makes it possible for the programmer to simultaneously queue the item as two types. Both ID blocks are queued according to their type code values when the item is picked.

Call Statement Format:

```
CALL GIPBUT (NCON, IIDDT, IDDT, IDDC, IDWA, IDWB)
```

NCON	Number of console to which the block should be assigned; only one console can be referenced with each call
IIDDT	Value to be used as a mask to determine if an item is a prime button type
IDDT	ID type code; used to specify how the queue handler will treat the prime button ID block
IDDC	ID code word; the contents assigned by the programmer can be $0 \leq IDDC \leq 2^8-1$
IDWA	ID information word A; if the IDDT of this ID block classifies it as a button pick type, this parameter should contain a portion of the name of the task overlay to be called by AEXEC
IDWB	ID information word B; contents are arbitrary unless the ID block is referenced by an application executive routine

Only one prime button ID block can be created for a given console; if several calls to GIPBUT occur with the same NCON value, the parameters from the latest call will replace all of the parameters previously entered in the block. If NCON is the only nonzero parameter used, the existing prime button ID block for console NCON will be removed from the 1700 series computer's memory.

The prime button mask is used in the following manner (the other queue handler processing masks are explained in the paragraphs on GIMASK).

When a string pick or single pick entry is made at console NCON, the queue handler processing mask comparisons are made. If the pick is not a button type and is not ignored, then the following comparison is performed by the 1700 interrupt processor:

IIDDT —⩛— IDDT

IIDDT	Prime button mask value
IDDT	ID type code of picked item
—⩛—	Logical exclusive OR

If this algorithm equals zero, the picked item is considered to be a prime button. The IDDT value in the prime button ID block is then compared with the programmer-defined masks to see how the prime button ID block should be processed; the item's regular ID block is processed separately, according to its own IDDT value.

The prime button ID block IDDT value can be any one (or none) of the valid mask values; it does not have to equal the mask value established for buttons.

CONTROL OF QUEUE HANDLER AND PICK PROCESSING

When an entry is made at a console, the 1700 interrupt processor ANDs the IDDT value in the entry's ID block with the value that the programmer has previously placed in the ignore mask. If the result of the operation is not zero, the entry is ignored. If the result is zero, the IDDT value is compared with values in the string pick, single pick, button, and marker masks. If the IDDT values correspond to any of these mask values, the queue handler performs the appropriate function; the ID block of the entry is either placed on one of the appropriate queue strings (string pick, single pick, or button pick) or the item on the screen is blinked (marker function). If the IDDT corresponds to more than one pick mask value, the ID block is queued according to the hierarchy: string, single, button. When one of the masking expressions is satisfied, no futher comparisons are made.

The algorithm used for the mask comparisons is given here to further explain the mask concept. In the following paragraphs:

IDDT	ID type code parameter from the ID block of the entry
∧	Logical AND
∨	Logical inclusive OR
IGM	Value set in ignore mask
SPM	Value set in single pick mask

STPM	Value set in string pick mask
BM	Value set in button mask
MM	Value set in marker mask

MASK COMPARISONS

The comparisons are listed below in the order in which they are made by the software.

IGNORE MASK

If IGM $\wedge$ IDDT $\neq 0$, the entry will be ignored, regardless of the contents of any other mask. For example, if IDDT also equals the value in the marker mask (indicating that the item should be blinked when picked), the item will not be blinked.

BUTTON MASK

If BM $\wedge$ IDDT $\neq 0$, the ID block for this entry is a button pick type; the ID block for this entry and any associated tracking cross coordinates, single pick ID blocks, and string pick ID blocks are queued after the information queued for the last button entry.

STRING PICK MASK

If STPM $\wedge$ IDDT $\neq 0$, the ID block for this entry is a string pick type; the ID block for this entry is queued after the ID block queued for the last string pick type entry.

SINGLE PICK MASK

If SPM $\wedge$ IDDT $\neq 0$, the ID block of this entry is a single pick type; the ID block queued for the last single pick type entry is replaced by the ID block of this entry.

MARKER MASK

If $\left(\text{IDDT} \wedge (\text{SPM} \vee \text{STPM} \vee \text{BM}) \neq \right) \wedge (\text{IDDT} \wedge \text{MM} \neq 0)$, the picked display item reverses blink status until its queued ID block is fetched by the application program. If IDDT is to be queued (i.e., when ANDed with any of the pick masks, the result is non-zero), it is compared with the marker mask. If the result of that comparison is non-zero, then the blink status is reversed.

GIMASK

This routine sets and clears the bits in the pick processing masks defined above. Each graphics console has its own set of masks, and the programmer establishes the value of each according to the IDDT parameter values that he wishes to use in his current application program. (If IMASK is not set, its default parameter is 1, for an ignore item. IMASK and IDDT may be replaced in the parameter string by such values as 16+8, 24, or 30B.)

Call Statement Format:

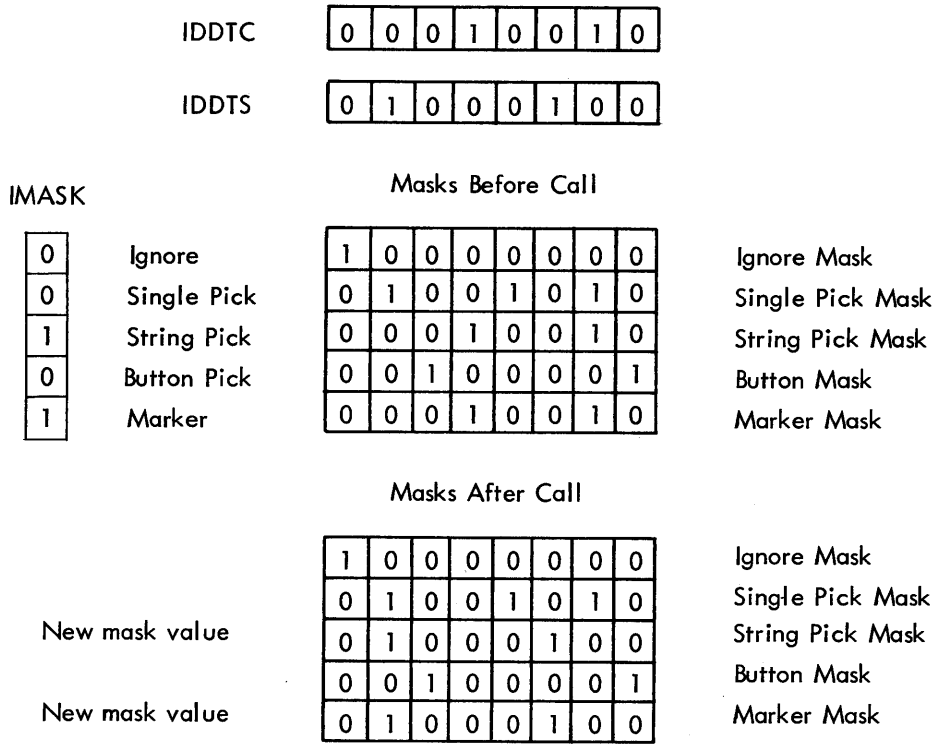
```
CALL GIMASK (NCON, IDDTC, IDDTS, IMASK)
```

NCON	Number of the graphics console for which the mask values will be used; only one console can be referenced through each call
IDDTC	Value of the bit pattern to be cleared from the specified pick processing masks
IDDTS	Value of the bit pattern to be set in the specified pick processing masks
IMASK	Mask indicator code; may be any one or any combination of the following:

= 1, set or clear the indicated bits in the ignore mask
= 2, set or clear the indicated bits in the single pick mask
= 4, set or clear the indicated bits in the string pick mask
= 8, set or clear the indicated bits in the button mask
= 16, set or clear the indicated bits in the marker mask

Several masks can be cleared or set simultaneously by placing the appropriate values in IDDTC and IDDTS, as in the following illustration. The IMASK value used is 24_8 ($=20_{10}$), IDDTC is 22_8 ($=18_{10}$), and IDDTS is 104_8 ($=68_{10}$).

The f bit in ICODE of GURSET, GICOPY, and GIMOVE controls the original blinking status of the item. If f (ICODE = s00tfbb) is set to 1, the item will blink; if f is not set, the item will not blink. However, the original blinking status will be reversed (i.e., a blinking item will stop blinking, a nonblinking item will blink) if the item is queued, provided that the marker mask is set for the item. As soon as the item is fetched, it will resume its original blinking status.



As an example of mask operation, assume that a programmer has defined grid lines as pick type 2. Each grid line has an ID block associated with it that contains an IDDT value of 2. Every time a grid line is picked by the console operator, the programmer wants to place the ID block for that grid line in the queue of string pick blocks and blink the grid line. To do this, he would make a GIMASK call with IDDTC = 0, IDDTS = 2, and IMASK = 16+4. This call would set both the string pick mask and the marker mask equal to two. See the IGS User's Guide and the IGS IMS Volume I for additional examples of mask manipulation by GIMASK.

GICLR

The GICLR routine clears <u>all</u> ID blocks associated with a particular graphics console from the FETCH and WAIT queues in the 1700 series computer's memory. This prevents the application program from acting upon the queued information after the programmer or console operator has decided that it is no longer needed to solve his problem.

Call Statement Format:

```
CALL GICLR (NCON)
```

NCON — Number of the console that should have its queued pick information destroyed; only one console can be referenced with each call

FETCHING ID BLOCKS FROM CONSOLE ENTRIES

ID information that has been queued as a result of console operator action can be retrieved from two areas within the Interactive Graphics System. GIBUT (and AETSKR) fetches ID blocks and ID information from the FETCH queues in the 1700 series computer. AELBUT, GIFSID and GIFID fetch ID information from the ID blocks stored in the 6000 series machine by the last GIBUT or AETSKR action.

Because the ID information is queued in two separate areas, the programmer must be careful when he fetches or uses it after a call to GICLR; the GICLR call erases information from the 1700 queues only. This means that calls to GIBUT will always fetch ID information queued after that last GICLR call occurred, but calls to AELBUT, GIFID, and GIFSID may reference information queued before the last GICLR call occurred. To avoid referencing the wrong ID information, a call to GICLR should be followed by a GIBUT call with IR = 0; after this call, the other four routines can be used without causing confusion.

AELBUT

This routine returns the ID information stored in the last button pick type ID block fetched from the 1700 series computer by a GIBUT or AETSKR call. AELBUT enables the programmer to investigate the parameters of the button which caused the calling of the current task overlay.

Call Statement Format:

```
CALL AELBUT (IDDT, IDDC, IDWA, IDWB, IH, IV)
```

IDDT — ID type code; returned as a result of the call

IDDC — ID code word; returned as a result of the call

IDWA — ID information word A; returned as a result of the call

IDWB — ID information word B; returned as a result of the call

IH	H axis (horizontal) coordinate of the lightpen pick which caused the button to be queued; returned as a result of the call
IV	V axis (vertical) coordinate of the lightpen pick which caused the button to be queued; returned as a result of the call

Parameters IDDC through IV are optional.

If a keyboard key, rather than a light-button, caused the calling of the current task, IH and IV will contain the keyboard status bits (see Table 3-1).

The IDDC parameter of any button referenced by AELBUT can be used to store the NCON of the console to which the button is assigned. This would give the programmer a means of determining which NCON value he should use in subsequent GIFID or GIFSID calls; if an NCON value other than that of the last GIBUT or AETSKR call is given in a GIFID or GIFSID call, a fatal error occurs (see Appendix B).

GIBUT

This routine fetches the first sequential button pick type ID block, and all related string pick type and single pick type ID blocks, from the FETCH queue of a particular graphics console. GIBUT also returns the parameters in the button ID information to the calling task. Once a call to GIBUT has been made, the information in the button ID block can be accessed again only through an AELBUT call, because another call to GIBUT will cause the next set of queued ID blocks to be fetched from the 1700 and will write over the information stored in the 6000 series machine. If the ID block was created by GILPKY, i.e., queued by a light-pen key interrupt, IH and IV will contain the coordinates of the tracking cross at the time of the interrupt.

Suppose the user displays three buttons with IDDCs of 1, 2, and 3; the names are RERUN, ERASE, and STOP; and the IDWAs and IDWBs are properly right-justified. He wishes to branch to the corresponding routines when a button is picked so he uses the following code:

```
      •
      •
      CALL GIBUT (0, NCON, IDDT, IDDC, IDWA, IDWB)
      GO TO (1, 2, 3) IDDC
    1 CALL AETSKC (5LRERUN)
      GO TO 10
    2 CALL AETSKC (5LERASE)
      GO TO 10
    3 CALL AETSKC (4LSTOP)
   10 CONTINUE
      •
      •
```

Call Statement Format:

```
CALL GIBUT (IR, NCON, IDDT, IDDC, IDWA, IDWB, IH, IV)
```

IR	Code to control return; if IR: = 0, wait for a button pick type ID block to be queued = 1, return to the calling task immediately
NCON	Number of the console from which the information should be retrieved
IDDT	ID type code; returned as a result of the call
IDDC	ID code word; returned as a result of the call
IDWA	ID information word A; returned as a result of the call
IDWB	ID information word B; returned as a result of the call
IH	H axis (horizontal) coordinate of the lightpen pick which caused the block to be queued; returned as a result of the call
IV	V axis (vertical) coordinate of the lightpen pick which caused the block to be queued; returned as a result of the call

If there is no button pick type ID block queued for console NCON and the call parameter IR equals zero, the application job will be rolled out until such a block is queued. If no such block is queued but IR equals 1, IDDT is returned as a positive zero.

GIFID

GIFID fetches the ID parameters from the last single pick type ID block stored in the 6000 series input buffer area by an AETSKR or GIBUT call. This is the ID block of the last single pick display item associated with the last button returned to the 6000 series computer. The NCON in a GIFID call must agree with the NCON of the last AETSKR or GIBUT call (see AELBUT, above).

Call Statement Format:

```
CALL GIFID (NCON, IDDT, IDDC, IDWA, IDWB, IH, IV)
```

NCON	Number of the console from which the ID block should be retrieved
IDDT	ID type code; returned as a result of the call
IDDC	ID code word; returned as a result of the call

IDWA	ID information word A; returned as a result of the call
IDWB	ID information word B; returned as a result of the call
IH	H axis (horizontal) coordinate of the lightpen pick which caused the block to be queued; returned as a result of the call
IV	V axis (vertical) coordinate of the lightpen pick which caused the block to be queued; returned as a result of the call

IH and IV contain the keyboard status bits if a keyboard key, rather than a display item pick, caused the block to be queued. The IH and IV parameters returned are the coordinates of the position where the beam was when the interrupt occurred and are in the vicinity of the display item; because the beam position varies slightly from pick to pick of the same item, IH and IV may also vary.

If no single pick type ID block is stored in the 6000 series computer, IDDT is returned as a positive zero; the values returned for the other parameters cannot be predicted.

A call to GIFID destroys the queued ID block. Thus, a second call to GIFID will return IDDT = 0.

GIFSID

GIFSID fetches the ID parameters from the last string pick type ID block stored in the program's application executive area by an AETSKR or GIBUT call. This is the ID block of the last string pick display item associated with the last button returned to the 6000 series machine.

A single GIFSID call can be used to fetch the parameters from several associated ID blocks, but the programmer must dimension the ID parameter and coordinate parameter names that he uses in his calling statement.

The NCON parameter of a GIFSID call must agree with the NCON of the last AETSKR or GIBUT call (see AELBUT).

Call Statement Format:

```
CALL GIFSID (NCON, N, IDDT, IDDC, IDWA, IDWB, IH, IV)
```

NCON	Number of the graphics console from which the information should be retrieved.
N	The number of string pick type ID blocks from which the programmer wishes to fetch parameters; if fewer than N blocks are queued in the 6000, N is returned equal to the number of blocks from which parameters could be returned. If N > 1, the following calling parameters must be dimensioned. N will always be the smaller of a) the number requested and b) the number of string pick type ID blocks available.

IDDT	ID type code; returned as a result of the call
IDDC	ID code word; returned as a result of the call
IDWA	ID information word A; returned as a result of the call
IDWB	ID information word B; returned as a result of the call
IH	H axis (horizontal) coordinate of the lightpen pick which caused the block to be queued; returned as a result of the call.
IV	V axis (vertical) coordinate of the lightpen pick which caused the block to be queued; returned as a result of the call

Only 16 (20_8) string pick blocks at a time are queued in the 6000 series computer.

If a keyboard key, rather than a display item pick, caused the block to be queued, then IH and IV contain the keyboard status bits. The IH and IV parameters returned after a display item pick indicate the position where the beam was when the interrupt occurred and is in the vicinity of the display item.

If no string pick type ID block is associated with the last button pick type ID block stored in the 6000 series computer, IDDT is returned as a positive zero and N is returned as 0; the values returned for the other parameters cannot be predicted.

Once retrieved, ID block parameters are lost; thus, a second call to GIFSID will return the ID parameters from the next string pick type ID block in the 6000 series queue (the next block in the time sequence of string pick type queue entries).

A GICNJB call cannot be made between two GIFSID calls that are intended to return values from the same string of ID blocks; such a call would cause a conflict in NCON and result in a fatal error.

CONTROL OF CONSOLE ALPHANUMERIC INPUT

No alphanumeric information can be entered into the system unless the application program first provides a place on the screen to enter it and then makes a call to the 1700 series computer requesting it.

GIANS

This routine creates a light-register on the screen so that the console operator can enter alphanumeric information. The register can contain up to 80_{10} characters at a time, and can appear anywhere on the screen.

GIANS displays a series of underline segments beginning at the screen coordinates supplied by the programmer and extending to the right across the screen up to the equivalent of 80 characters. The area immediately above this underline constitutes the light-register. When the console operator presses an alphanumeric keyboard key or picks a font character with the lightpen, the individual letter, symbol, or number is displayed in the register, starting at the left, and the corresponding portion of the underline disappears.

If GIANS is called again while the console operator is entering alphanumeric information, the current contents of the register are destroyed; each call to GIANS defines a new register.

Call Statement Format:

```
CALL GIANS (NCON, NC, IH, IV)
```

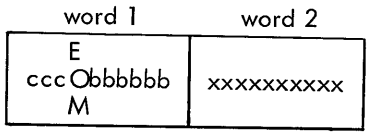
NCON	Number of the graphics console on which the light-register should be created; only one console can be referenced through each call
NC	Maximum number of characters that will be permitted in the register (defines the number of underline segments)
IH	H axis (horizontal) coordinate of the left end of the underline
IV	V axis (vertical) coordinate of the left end of the underline

GIANE

This routine performs three functions, in the following order:

1. It stops the entry of alphanumeric information into the currently defined light-register.

2. It then transfers the characters currently in the register to the calling program as an array buffer; this buffer contains 10 characters (in 6000 series display code) per word. The characters are left-justified within a word, and blank-fill is provided for any word not completely filled. When $11 \leq NC \leq 20$, word 2 is blank-filled if not completely filled. When $0 \leq NC \leq 10$, word 1 is blank-filled, but word 2 is not; i.e.,



3. It clears all characters from the register and removes any remaining portion of the underline.

If the number of characters entered in the register is less than the maximum number specified by the NC parameter of this call, the number entered in the register will be returned as a result parameter.

Call Statement Format:

```
CALL GIANE (NCON, NC, IBUF)
```

NCON	Number of the console from which the characters should be fetched; only one console can be referenced through each call.
NC	Maximum number of characters in the character buffer. If more than NC characters are entered, only NC characters are returned; if fewer than NC characters are entered, NC is returned equal to the number of characters in the character buffer.
IBUF	Array buffer of picked characters; returned as a result of the call.

After GIANE has been called, the programmer must call GIANS before any further alphanumeric information can be entered.

FRAME-SCISSORING DISPLAYS

Before displaying a line or arc on the console screen, the programmer may want to assure that it lies entirely within a specific area (see Display Presentation, Section 3). He can do this by calling either the GULINE or GUARC frame-scissoring routine and then using the results of his call in subsequent calls to display item generation routines. GULINE and GUARC do not display anything on the console screen or create an item description that can be displayed; this must be done by other routines.

GULINE

This routine determines the points at which a given line intersects a given frame. If the given line lies completely within the frame, the display grid coordinates of the end points of the line are returned to the application program. If the line is partially within the frame, the grid coordinates of the end points of that part of the line are returned.

GULINE also scissors out lines that are too small for the graphics console operator to discern. This microscissoring is performed on any line less than six display grid units long. The end point coordinates returned after such an operation are meaningless.

If the given line lies completely outside of the given frame, the end points returned by GULINE are meaningless.

Call Statement Format:

```
CALL GULINE (IHCEN, IVCEN, IHCOR, IVCOR, H1, V1, H2, V2,
             1 KSHOW, IH1, IV1, IH2, IV2)
```

IHCEN, IVCEN	Horizontal and vertical display grid coordinates of the center of the frame
IHCOR, IVCOR	Horizontal and vertical display grid coordinates of the upper right-hand corner of the frame
H1, V1, H2, V2	Horizontal and vertical display grid coordinates of the left and right ends (respectively) of the line that the programmer wants scissored; these should be floating-point values, rather than integers
KSHOW	Scissor flag, returned as a result of the call; if KSHOW: = 0, the given line is either completely outside the frame or has been microscissored = 1, the given line is completely within the frame = 2, the given line is partially within the frame and has been scissored
IH1, IV1, IH2, IV2	Horizontal and vertical display grid coordinates of the left and right end points (respectively) of that portion of the line within the frame; returned as a result of the call, but meaningless if KSHOW equals zero

GUARC

This subroutine determines the points at which a given arc intersects a given frame. If the given arc lies completely within the frame, the display grid coordinates of the arc's center and end points are returned to the application program. If the arc is partially within the frame, the grid coordinates of the arc's center and of the end points of those parts of the arc within the frame are returned.

GUARC also scissors out arcs that are too small for the graphics console operator to discern. This microscissoring is performed on any arc with end points less than six grid units apart. The end point values returned after such an operation are meaningless.

If the given arc lies completely outside of the given frame, the end point coordinates returned by GUARC are meaningless.

GUARC is used for both arcs and circles, since the Interactive Graphics System defines only circular arcs. If the programmer wants to frame-scissor an arc that is almost a complete circle, the end point values returned to him may represent up to five separate arc segments, as in Figure 6-5.

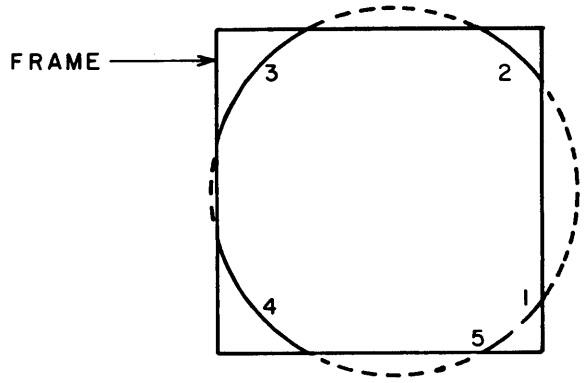


Figure 6-5. Example of a Frame-scissored Arc

Call Statement Format:

```
CALL GUARC (IHCEN, IVCEN, IHCOR, IVCOR, HC, VC, H1, V1
           1 H2, V2, KSHOW, IHC, IVC, IH1, IV1, IH2, IV2)
```

IHCEN, IVCEN	Horizontal and vertical display grid coordinates of the center of the frame
IHCOR, IVCOR	Horizontal and vertical display grid coordinates of the upper right-hand corner of the frame
HC, VC	Horizontal and vertical display grid coordinates of the center of the circular arc that the programmer wants scissored
H1, V1, H2, V2	Horizontal and vertical display grid coordinates of the right and left ends (respectively) of the arc that the programmer wants scissored; arcs are defined counterclockwise
KSHOW	Scissor flag, returned as a result of the call; if KSHOW: = 0, the given arc is either completely outside of the frame or has been microscissored = 1 through 5, it indicates the number of arc segments within the frame
IHC, IVC	Horizontal and vertical display grid coordinates of the center of the arc; returned as a result of the call, but meaningless if KSHOW equals zero
IH1, IV1, IH2, IV2	Horizontal and vertical display grid coordinates of the end points of those portions of the arc within the frame; returned as a result of the call, but meaningless if KSHOW equals zero

Each of the last four parameter names is the first word of an array KSHOW words in length. The coordinate value in each word corresponds to the segment of the arc that follows sequentially, counterclockwise, after the coordinate of the segment corresponding to the word before it. The first word in each array contains the coordinate of the first such segment that occurs after the initial end point specified for the programmer's original arc.

DISPLAY ITEM GENERATION

The nine routines that generate display item descriptions can be used to create a figure composed of lines or arcs (or any arbitrary figure combining lines and arcs), to display alphanumeric information, to define an item as display macro, and to change the 1744 Controller's current control byte values.

These routines do not display anything on the graphics console screen; that can be done only by a separate GIDISP call.

All but one of the nine routines have the following three programmer-defined parameters in common:

IBUF	An array buffer used to contain description bytes produced by the display item generation routines. The contents of each IBUF define one display item or display macro. IBUF must be dimensioned by the programmer; the recommended size is 64_{10} 60-bit words.
MBYTE	Maximum number of 12-bit bytes which the programmer will allow to be packed in the IBUF words. MBYTE should be $\leq 310_{10}$.
NBYTE	Number of bytes currently in the IBUF words. NBYTE is set equal to zero by the programmer every time he starts a new IBUF, and its value is automatically updated after each call to a generation routine.

Each call to a generation routine produces bytes of information in addition to that supplied by the programmer. These bytes are calls to the 1700 Package equivalent of the 6000 Package routine and are the first bytes packed into IBUF by the call. Because of similar extra bytes, the IBUF used in a call to GIDISP cannot be filled such that NBYTE is greater than 310 before the call; the limit on an IBUF used in a call to GIMAC is 318_{10} bytes before the call.

Under certain conditions, a non-fatal error may occur and cause a generation call to be ignored (see Appendix B); in this case, the extra bytes are not placed in IBUF.

If a generation routine is called and its actions cause NBYTE to exceed MBYTE, IBUF will only include the last MBYTE description bytes placed in it. This condition produces a non-fatal error diagnostic and the overflow bytes are lost. A statement such as

```
IF (NBYTE.LT.MBYTE)  10,  1000
```

could be used to check for the buffer overflow condition and avoid later problems caused by a truncated IBUF.

GURSET

This routine establishes the initial conditions for a display item which is described in subsequent generation routine calls. A call to GURSET turns the light beam off, initiates a reset sequence, and drives the beam to the absolute screen coordinates given in the call. GURSET places a reset sequence specified by the programmer in an IBUF description buffer — which is then filled with description bytes by calling other display item generation routines.

The reset sequence consists of bytes that set the cathode beam intensity, item lightpen sensitivity, item blinking capability, and the display grid coordinates to which the beam should be moved (with the beam off). However, if many GURSETs are used, the reset sequence may cause flicker or blinking of the display as well as fill up the refresh buffer in the controller.

The reset sequence is the equivalent of several 1744 Controller command and control bytes.

A GURSET call should precede all other generation routine calls when a new display item description buffer is started. GURSET can also be used to place reset information in a partially filled IBUF if the programmer wishes to move the beam or change intensity in the middle of a display item; an example of these uses is given in the paragraphs concerning GUAN.

A display macro IBUF does not require a reset sequence.

Call Statement Format:

```
CALL GURSET (IH, IV, ICODE, IBUF, NBYTE, MBYTE)
```

IH, IV	Absolute horizontal and vertical display grid coordinates of the point at which the cathode beam should be repositioned on the screen (less than $\|\pm 2047\|$ dgus)

ICODE — Reset control code bit pattern of the form: s00tfbb B
i.e.: 1 0 3 B

- s = 0, disable item's sensitivity to lightpen strike
  = 1, enable item's sensitivity to lightpen strike
- t = 0, normal display processing
  = 1, inhibit display processing until next reset sequence
- f = 0, don't blink item when it is displayed
  = 1, blink item when it is displayed
- bb = 01, display item with beam at low intensity
  = 10, display item with beam at medium intensity
  = 11, display item with beam at high intensity

IBUF — Description buffer for this display item; contains reset information as a result of this call

NBYTE — Number of bytes currently in IBUF

MBYTE — Maximum number of bytes the programmer will allow in IBUF

GUAN

This routine packs a description of alphanumeric information into a display item description buffer. A subsequent GIDISP call will display the information as lines of characters on the console screen. The first character is positioned with the lower left-hand corner of the character's 24 by 24 (or optional 32 by 32) dgu square on the point defined by the most recent GURSET call. The characters are left-justified within the square, with spacing at the top and right side of the square.

Although a single GUAN call will pack up to 255_{10} characters into an IBUF, there is a smaller practical limit for a single call. Each character occupies an area on the screen that is 24_{10} grid units square unless the installation has chosen the 32_{10} dgu character set option. This limits the maximum length of a line defined by a GUAN call to 170_{10} characters; if more than 170_{10} characters are placed in IBUF for a single line, wraparound will occur on the display.

Because a GUAN call generates a return jump to a 1700 macro, this routine cannot be used to place alphanumeric information into a display macro IBUF.

Call Statement Format:

```
CALL GUAN (IBCD, NC, IBUF, NBYTE, MBYTE)
```

IBCD — First word of the array of characters which are to be displayed; 10 per array word in left-justified 6000 series internal display code

NC — Number of characters from IBCD that should be packed by this call; if NC > 255_{10}, the extra characters will not be packed in IBUF

IBUF	Description buffer for this display item; the packed alphanumeric data is returned as a result of the call
NBYTE	Number of bytes currently in IBUF; updated as a result of the call
MBYTE	Maximum number of bytes the programmer will permit in IBUF

Each line of alphanumeric information should be defined by a separate GUAN call, but at least seven full lines can be placed in one IBUF as a single display item. The following example illustrates this.

```
      •
      •
   COMMON IBUF (63), IBCD (49)
   NBYTE = 0
   MBYTE = 310
   ICODE = 102B
   READ 10, (IBCD (N), N=1, 49)
10 FORMAT (8A10/)
   CALL GURSET (4200B, 400B, ICODE, IBUF, NBYTE, MBYTE)
   CALL GUAN (IBCD (1), 70, IBUF, NBYTE, MBYTE)
   CALL GURSET (4200B, 350B, ICODE, IBUF, NBYTE, MBYTE)
   CALL GUAN (IBCD (8), 70, IBUF, NBYTE, MBYTE)
      •
      •
```

Note that the lines of alphanumeric information in the above example are not 170 characters long. Because the console screen is circular, the maximum line length depends on the point of origin or the line on the screen; a 170-character line would have to originate at (or very near) IH = -2047, IV = 0000. An 88-character line will fit almost anywhere on the screen.

If the programmer wishes to display a character other than those defined for the 274 console screen (see Appendix C), he cannot use GUAN unless he changes the macro address table in the 1700 Basic Graphics Package equivalent of GUAN; each character is defined as a display macro by the latter routine.

GUSEGS

This routine generates the description of a line segment and packs it in an IBUF description buffer. GUSEGS can be used to generate the description of a single line or the description of the first line segment in a figure; in the latter case, the parameters in the GUSEGS call

can be used to give this first line segment an appearance different from that of the reset of the figure. GUSEGS may be used to draw a line of minimum length (6 dgu) which will appear on the console screen as a point. The width of the light beam is approximately 5 dgus (at 50% intensity).

Although GUSEGS can be used to initialize a figure (which is generated by later calls to other routines), it does not place a reset sequence in IBUF. If IBUF does not already contain a reset sequence, a GUSEGS call must be preceded by a GURSET call; a macro buffer does not need a reset sequence.

Call Statement Format:

```
CALL GUSEGS (IH1, IV1, IH2, IV2, IBEAM, ISTYLE, IBUF, NBYTE, MBYTE)
```

IH1, IV1 — Horizontal and vertical display grid coordinates for starting point of the line segment. The coordinates are relative to the most recent call to GURSET.

IH2, IV2 — Horizontal and vertical display grid coordinates for the end point of the line segment which must be at least 6 dgus from the starting point of the segment if it is to appear on the console.

IBEAM — Beam control parameter that determines the appearance of this line segment only; if IBEAM

- = 0, this segment is not displayed
- = 1, this segment is displayed according to ISTYLE

The following values can be used when figure generation is finished; if IBEAM

- = -0, turn beam off and stop the beam at the end point
- = -1, turn beam on and stop the beam at the end point

ISTYLE — Style control parameter that determines the appearance of this segment and any figure generated by subsequent GUSEG calls; the degree of solidity of the line depends on the number of set bits in this parameter, as in the following sample values:

- = 7777B, 0, or -0, segment is solid (7777B is conventional)
- = 5252B, segment is dashed
- = 6666B, segment is broken
- = 7272B, segment has appearance called center line by engineers

IBUF — Description buffer for this display item; contents returned depend on the call

NBYTE — Number of bytes currently in IBUF; an updated value is returned as a result of this call

MBYTE — Maximum number of bytes the programmer will permit in IBUF

If the programmer wants to frame-scissor his figure, the IH1, IV1, IH2, IV2 parameters passed to this call should contain the values returned by a GULINE call.

GUSEGI

This routine is used to initialize a figure that is generated by later calls to GUSEG, GUSEGI does not generate the description of a line segment, as GUSEGS does, but merely determines the starting point of a figure and controls its appearance.

GUSEGI does not place a reset sequence in IBUF. If IBUF does not already contain such a sequence, a GURSET call must precede the call to GUSEGI; a macro IBUF need not contain a reset sequence.

Call Statement Format:

```
CALL GUSEGI (IH1, IV1, ISTYLE, IBUF, NBYTE, MBYTE)
```

IH1, IV1	Horizontal and vertical display grid coordinates for the starting point of the figure. The coordinates are relative to those specified in the last call to GURSET.
ISTYLE	Style control parameter that determines the appearance of the entire figure; the solidity of the lines in the figure depends on the number of set bits in this parameter, as in the sample values given for GUSEGS.
IBUF	Description buffer for this display item; the contents returned depend on the results of this call.
NBYTE	Number of bytes currently in IBUF; an updated value is returned as a result of this call.
MBYTE	Maximum number of bytes that the programmer will permit in IBUF.

GUSEG

Each call to GUSEG generates the description of a single line segment and packs it in an IBUF description buffer. GUSEG does not initialize a figure and must be preceded by either a GUSEGA, GUSEGS, or GUSEGI call; otherwise, a fatal error occurs.

The appearance of a figure generated by calls to GUSEG depends on the ISTYLE value used in the initial GUSEGA, GUSEGS, or GUSEGI call and on the beam control code of each GUSEG call. The last point specified in a preceding call to GUSEGS, GUSEGA, GUSEGI, or GUSEG is used as the starting point for the line segment generated by the current GUSEG call.

Call Statement Format:

```
CALL GUSEG (IH, IV, IBEAM)
```

IH, IV — Horizontal and vertical display grid coordinates for end point of this segment

IBEAM — Beam control parameter that determines the appearance of this line segment only; if IBEAM:

- = 0, this segment is not displayed
- = 1, this segment is displayed according to ISTYLE

The following values can be used when figure generation is finished; if IBEAM:

- = -0, turn beam off and stop the beam at the end point
- = -1, turn beam on and stop the beam at the end point

The IBUF array and MBYTE parameter used by a GUSEG call are the ones specified in the last GUSEGS or GUSEGI call; each GUSEG call also automatically updates the last NBYTE value.

GUSEGA

In contrast to the GUSEG routine, which must be used in conjunction with GUSEGS or GUSEGI, the GUSEGA routine performs its own initialization and then generates the description of an entire figure. One GUSEGA call can thus be used to replace many GUSEG calls if none of the parameters defining the figure depend on a console operator's actions.

GUSEGA does not place a reset sequence in the IBUF description buffer. If IBUF does not already contain such a sequence, a GURSET call must precede the call to GUSEGA; a macro IBUF need not contain a reset sequence. The coordinates in the call to GUSEGA are relative to those in the last call to GURSET.

Call Statement Format:

```
CALL GUSEGA (IH, IV, IBEAM, N, ISTYLE, IBUF, NBYTE, MBYTE)
```

IH, IV — First words of arrays containing the horizontal and vertical (respectively) display grid coordinates for the end points of each figure segment; this routine uses the end point of the last segment as the starting point of the next, so each segment after the first requires only one pair of coordinates

IBEAM — First word of an array containing the beam control code for each figure segment; if an array word:

- = 0, the segment is not displayed
- = 1, this segment is displayed according to ISTYLE

The following values can be used when figure generation is finished; if IBEAM:

- = -0, turn beam off and stop the beam at the end point
- = -1, turn beam on and stop the beam at the end point

N — Number of figure segments to be generated by the current call

ISTYLE — Style control parameter that determines the appearance of the entire figure; the solidity of the lines in the figure depends on the number of set bits in this parameter, as in the sample values given for GUSEGS

IBUF — Description buffer for this display item; the contents returned depend on the results of the call

NBYTE — Number of bytes currently in IBUF; an updated value is returned as a result of the call

MBYTE — Maximum number of bytes that the programmer will permit in IBUF

N should always be one less than the number of values in the IH and IV arrays because the first two words in IH and IV define only one line segment; that is, the first word identifies the starting point and the second word identifies the end point of the first segment in the figure. However, IBEAM(i) identifies a segment only by its end point IH(i+1) IV(i+1); thus IBEAM (i+1) identifies, but does not describe, a segment. IBEAM(i) can be set equal to turn the cathode beam off when the figure is completed.

GUARCG

This routine generates a description of several arcs or a circle and packs the information in an IBUF description buffer. GUARCG can define up to five separate or connected circular arcs, deployed counterclockwise around a common center.

If the programmer wishes to frame-scissor a circular figure, the array of end points used in the GUARCG call should be the same as the array produced by a previous call to GUARC.

GUARCG does not place a reset sequence in IBUF. If the description buffer does not already contain such a sequence, a call to GURSET should precede the GUARCG call; a macro IBUF need not contain a reset sequence.

Call Statement Format:

```
CALL GUARCG (KSHOW, IHC, IVC, IH1, IV1, IH2, IV2, ISTYLE, IBUF,
   1 NBYTE, MBYTE)
```

KSHOW	Number of arc segments to be generated by this call; must be less than six
IHC, IVC	Horizontal and vertical display grid coordinates for the common center of the arcs
IH1, IV1	First words of arrays containing the horizontal and vertical display grid coordinates for the starting point of each arc segment
IH2, IV2	First words of arrays containing the horizontal and vertical display grid coordinates for the end point of each arc segment
ISTYLE	Style control parameter that determines the appearance of all the arc segments; the solidity of the lines depends on the number of bits set in the parameter, as in the sample values given for GUSEGS
IBUF	Description buffer for this display item; the contents returned depend on the call
NBYTE	Number of bytes currently in IBUF; an updated value is returned as a result of the call
MBYTE	Maximum number of bytes that the programmer will allow in IBUF

GUBYTE

GUBYTE is a general purpose routine. It is used to place information into an IBUF description buffer when the information is a type other than that processed by the regular display item generation routines.

The information packed by GUBYTE is placed into the 1744 Controller. (See Appendix I, page I-3, for example of GUBYTE calls. The octal equivalent of the hexadecimal column in Appendix C gives equivalents of alphanumeric characters. These equivalents are used in GUBYTE calls.)

GUBYTE transfers the lowest 12 bits from each word in an input array to the specified IBUF. Each 12-bit byte is left-justified next to the last byte entered in the buffer. IBUF, as well as any other buffer produced by a display item generation routine, is packed with five bytes in each of its words.

Call Statement Format:

```
CALL GUBYTE (IBYTE, L, IBUF, NBYTE, MBYTE)
```

IBYTE	First word of the array containing one description byte at the lower end of each word
L	Number of consecutive words in IBYTE from which bytes are to be transferred
IBUF	Description buffer for this display item; contents returned depend on the call
NBYTE	Number of bytes currently in IBUF; an updated value is returned as a result of the call
MBYTE	Maximum number of bytes which the programmer will allow in IBUF

GUMACG

This routine places a macro call description into an IBUF description buffer. This allows a display item to use display macros that were previously defined by calls to GIMAC. Each call to GIMAC sends an IBUF to the 1700, where its contents are translated, converted into a display byte stream, and stored in the memory of the 1744 Controller. GIMAC then returns an associative address for that macro to the calling program. The macro is not displayed until a GUMACG call and a subsequent GIDISP call place a calling sequence for it into the display byte stream of a regular display item. This is done by inserting the sequence into the IBUF which describes the regular display item of which the macro is to be a part.

Call Statement Format:

```
CALL GUMACG (MAD1, L, IBUF, NBYTE, MBYTE)
```

MAD1	First word of an array containing macro address MAD parameters returned by previous calls to GIMAC
L	Number of consecutive MAD parameters from MAD1 that are to be placed in IBUF by this call
IBUF	Description buffer for this display item; the contents returned depend on the call
NBYTE	Number of bytes currently in IBUF; an updated value is returned as a result of this call
MBYTE	Maximum number of bytes that the programmer will allow to be placed in IBUF

STORING AND DISPLAYING ITEMS

Once the description of a display item is finished, the filled IBUF is placed in the display buffer of the 1744 Controller through a GIDISP or GIMAC call; GIDISP defines the contents of IBUF as a regular display item, which is then shown on the console screen; GIMAC defines the contents as a display macro, which does not appear on the screen unless a call to it occurs in a regular display item. After the item is placed in the display buffer, it can be

- Duplicated on another part of the screen, with a new reset sequence and a new ID block
- Moved to another part of the screen, with a new reset sequence and a new ID block
- Erased from the screen and the display buffer
- Turned off (i. e., not displayed) but not removed from the buffer so the user may turn it on again at a later time with a reset

GIMAC

This routine sends the contents of an IBUF description buffer to the 1700 series computer, where its contents are translated and then converted into a display byte stream by the 1700 Package routines. (There is a check for a valid call code and parameters before the contents of IBUF are sent to the 1700 translator.) The 1700 version of GIMAC stores this display byte stream as a display macro in the display buffer of the specified console's controller.

GIMAC does not display the macro on the console screen, but returns the associative address of the macro to the programmer. This address parameter is then used by GUMACG to generate a macro call in the IBUF of a regular display item. A subsequent call to GIDISP for the regular display item also displays the macro.

Note that the ID block entered into the 1700 queue, when a macro is picked, is the ID block of the regular display item which called the macro.

There is only one level of macros within the Interactive Graphics System. If an IBUF is being used for the description of a macro, it cannot contain a call to another macro; GIMAC can never be called to process an IBUF that has been used for previous calls to GUAN or GUMACG.

Call Statement Format:

```
CALL GIMAC (NCON, IBUF, NBYTE, MAD)
```

NCON	Number of the console to which the macro should be sent; only one console can be referenced through each call

IBUF	Description buffer for this macro; contents returned depend on the call
NBYTE	At the time of the call the number of bytes currently in IBUF; NBYTE is returned = 0
MAD	Display buffer associative address of the new macro; returned as a result of the call

GIMACE

GIMACE removes one or more macros from a console controller's display buffer and frees that area of the buffer for later use.

If GIMACE is called to erase a macro that is used by one of the regular display items, the GIMACE call will have unpredictable – and probably chaotic – results on the screen. The programmer can avoid this problem by preceding a GIMACE call with a call to GIERAS; the GIERAS call erases all regular display items which use the macro that the programmer wants to erase.

Call Statement Format:

```
CALL GIMACE (MAD1, MAD2, ..., MADn)
```

MAD_i	Display buffer address of the macro to be erased; a right parenthesis or a MAD_i equal to minus zero may be used to end the parameter list. The maximum value of n is 36.

If a MAD_i equal to positive zero occurs in the middle of the call's parameter list, the addresses following it will be ignored and their associated macros will not be erased. A zero is returned in the MAD_i parameter of each macro that has been erased.

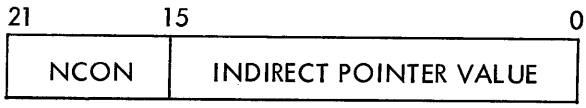
GIDISP

This routine sends the contents of an IBUF description buffer to the 1700 series computer where its contents are translated and then converted into a display byte stream by the 1700 Package routines. The validity of the call code and parameters is checked before the contents of IBUF are sent to the 1700 translator. The 1700 version of GIDISP stores this display byte stream in the display buffer of the specified console's controller and associates an ID block with it. GIDISP then returns an associative address to the calling program.

This address is the relative address of the regular display item within a table of actual display buffer addresses maintained by the 1700 Package equivalent of GIDISP; the associative address is used by the programmer for all subsequent references to the display item.

GIDISP is the only routine in the 6000 Basic Graphics Package which can display a new item on the console screen.

The IDDAD associative address parameter has the following structure:



NCON Octal number of the console to which the item is assigned

Call Statement Format:

```
CALL GIDISP (NCON, IBUF, NBYTE, IDDAD, IDDT, IDDC, IDWA, IDWB)
```

NCON	Number of the graphics console on which the item should be displayed; only one console can be referenced through each call
IBUF	Description buffer for this display item; contents returned depend on the call parameters
NBYTE	At the time of the call the number of bytes currently in IBUF/ NBYTE is returned = 0
IDDAD	System-defined associative address of the display item; returned as a result of the call
IDDT	ID type code; used to specify how the queue handler will treat the item's ID block (see GIMASK)
IDDC	ID code word; the contents assigned by the programmer are $0 \leq IDDC \leq 2^8 - 1$
IDWA	ID information word A; contents are arbitrary unless the item ID block is used by AETSKR
IDWB	ID information word B; contents are arbitrary unless the item ID block is used by AETSKR

A standard display item identification byte stream is formed from parameters IDDT through IDWB and is added to the end of the display byte stream for the item in the 1744's display buffer; if any of the last four parameters is set equal to -0, that parameter and any subsequent ones are omitted from the identification byte stream. (A parameter list may also be terminated by a right parenthesis.) An item defined as a button and processed by AETSKR must have an IDWA; if none exists, AETSKR produces a diagnostic (see Appendix B) and no task is loaded. If IDDT, IDDC, IDWA, or IDWB are not used, it is good programming practice to terminate the argument list after the last variable used; this conserves display buffer memory.

A task name used by AETSKR must be right-justified within both the IDWA and IDWB words, but it must be left-justified as a whole.

For example, the name TSK could be placed in bits 23 through 6 of word IDWA, with zero fill in bits 5 through 0. This could be done by the statement

```
IDWA = 4RTSK
```

Note that the statement

```
IDWA = 3RTSK
```

would produce an invalid task name by placing 0TSK in IDWA. AETSKR does not handle a task name that begins with a zero, so this condition would abort the job.

A longer name such as TSKNAM would have to be stored so that TSKN filled IDWA and bits 23 through 12 of IDWB contained the characters AM. The statements

```
IDWA = 4RTSKN
IDWB = 4RAM
```

illustrate how TSKNAM can be stored. Note that the statement

```
IDWB = 2RAM
```

will produce an invalid task name by placing 00AM in IDWB; AETSKR will not recognize the resulting TSKN00AM as the task called TSKNAM. Once GIDISP specifies a routine name with IDWA and IDWB, a call to AERTRN can branch to that routine in another overlay level. Then AELBUT can investigate the parameters of the item displayed in the original overlay before the call to AERTRN.

The programmer cannot allow NBYTE to exceed 312_{10} bytes. The EXPORT HS graphics output buffer contains space for 320_{10} bytes of information, and the identification bytes fill eight of them (the equivalent GIMAC bytes fill two). Since the first two bytes of every IBUF are reserved for the function code and the NBYTE value, no more than 310_{10} description bytes can be placed in the IBUF of a regular display item and no more than 316_{10} in a macro IBUF.

GIERAS

The GIERAS routine removes one or more display byte streams from the display buffers of the consoles. This erases the display item associated with each byte stream and also removes any of the items' ID blocks currently in the FETCH or WAIT queues – regardless of the blocks' pick types.

The programmer uses the associative addresses (produced by previous calls to GIDISP) to indicate the display items that he wants GIERAS to erase from the console screen. If he plans to call GIERAS (IDDAD) before GIDISP has defined IDDAD (as in a loop), it is good programming practice to initialize IDDAD to 0.

Call Statement Format:

```
CALL GIERAS (IDDAD1, IDDAD2, . . . , IDDADn)
```

$IDDAD_i$	Associative address of the display item to be erased; an $IDDAD_i$ equal to minus zero, zero, or a right parenthesis may be used to end the parameter list. The maximum value of n is 36.

If an $IDDAD_i$ equal to minus zero occurs in the middle of the call's parameter list, the addresses following it are ignored and their associated display items are not erased; if an $IDDAD_i$ equal to positive zero occurs, it is not processed but subsequent addresses are.

A zero is returned in the $IDDAD_i$ parameter of each display item that has been erased.

GICOPY

The GICOPY routine duplicates an existing display item, assigns a new ID block and a new reset sequence to the copy, and displays the copy at a new location on the console screen or on the screen of a different console. The duplication process does not erase or change the original display item.

Note that the reset sequence changed by a GICOPY call is the first such sequence placed in the IBUF of the original item; if the description of the original display item contains more than one reset sequence, the values assigned to the reset sequence of the copy should not be changed.

Display of the duplicate item begins at the same point within the item as it begins in the original item; i. e., if the original item was described beginning in its lower left-hand corner, then the duplicate will also begin there.

Call Statement Format:

```
CALL GICOPY (IDDADI, NCON, IH, IV, ICODE, IDDAD, IDDT, IDDC, IDWA, IDWB)
```

IDDADI	Associative address of the item which is to be duplicated
NCON	Number of the graphics console on which the duplicate display item should appear; only one console can be referenced through each call

IH, IV	Horizontal and vertical display grid coordinates for the reset sequence of the duplicate item; these are the absolute coordinates of the copy's point of origin
ICODE	Reset control code to be assigned to the copy; the s00tfbb bit pattern has the same meanings as those defined for GURSET
IDDAD	Associative address assigned by the system to the duplicate display item; returned as a result of this call
IDDT	ID type code to be assigned to the ID block of the duplicate item; used to specify how the queue handler will treat the duplicate item's ID block
IDDC	ID code word for the ID block of the duplicate item; the contents assigned by the programmer are $0 \leq \text{IDDC} \leq 2^8 - 1$
IDWA	ID information word A for the ID block of the duplicate item; contents are arbitrary unless the item ID block is processed by AETSKR (see GIDISP)
IDWB	ID information word B for the ID block of the duplicate item, contents are arbitrary unless the item ID block is processed by AETSKR (see GIDISP)

When one of the call statement parameters IH through IDWB (not including IDDAD) is set equal to -0, that parameter for the copy will be left as it is in the original display item. The parameters IDDT through IDWB may be omitted; this has the same effect as setting them equal to -0.

If controller memory for the display byte stream of the copy is unavailable, a buffer overflow message will be produced at the 1700 series operator's console and the 274 console.

GIMOVE

The GIMOVE routine can change the location, reset sequence, and/or ID block information of an existing display item. This allows the programmer to change such features of a display item as its pick type, intensity, sensitivity to lightpen pick, and whether or not it can be blinked. GIMOVE does not create a new item; it alters the location of the existing display item.

Note that the reset sequence changed by a call to GIMOVE is the first such sequence placed in the item's IBUF. When the description of the item contains more than one reset sequence, the item cannot be moved; therefore, IH, IV, and ICODE in the following call should be set equal to -0.

Call Statement Format:

```
CALL GIMOVE (IH, IV, ICODE, IDDAD, IDDT, IDDC, IDWA, IDWB)
```

IH, IV	New horizontal and vertical display grid coordinates for the reset sequence of the item; these are the absolute coordinates of the item's point of origin
ICODE	New reset control code for the item; the s00tfbb bit pattern has the same meanings as those defined for GURSET
IDDAD	Associative address of the display item; not changed by the call
IDDT	New ID type code for the item's ID block; used to specify how the queue handler will treat the block
IDDC	New ID code word for the item's ID block; contents assigned by the programmer are $0 \leq$ IDDC $\leq 2^8 - 1$
IDWA	New ID information word A for the item's ID block; contents are arbitrary unless the block is processed by AETSKR (see GIDISP)
IDWB	New ID information word B for the item's ID block; contents are arbitrary unless the block is processed by AETSKR (see GIDISP)

When one of the call statement parameters IH through IDWB (not including IDDAD) is set equal to -0, that parameter for the copy will be left as it is in the original display item. The parameters IDDT through IDWB may be omitted; this has the same effect as setting them equal to -0.

CONTROL AND USE OF THE TRACKING CROSS

Each graphics console in the Interactive Graphics System is equipped with a lightpen tracking feature called the tracking-cross. This cross always exists somewhere on the console screen grid, but the programmer may move it off the visible screen area if he wishes. The cross is a system-defined display item described by a byte stream that is automatically placed in the display buffer of each console's controller whenever the console is initialized.

The display grid coordinates of the cross are kept in a fixed location in the display buffer (see Section 3). The 6000 Basic Graphics Package contains routines that set and fetch these coordinates; by using these routines, the programmer can determine or change the cross location.

The cross and lightpen are used together in the following manner. The pen is used to pick the cross at some location on the screen. The cross is then automatically attached to the pen so that it moves with, or tracks, the lightpen as the pen is moved across the screen. When the pen is stopped and the cross comes to rest, the location of the cross defines the point of a lightpen pick. If the pen and cross are moved across a display item, no lightpen

pick is recorded; the cross must be motionless before a pick can be detected. This feature allows the cross to be moved across the screen without causing unwanted lightpen picks.

Also provided in the 6000 Package are two routines which attach a display item or display macro to the tracking-cross. Such an item or macro moves with the cross across the screen until detached by another call.

GITCON

GITCON turns the tracking-cross on (makes it visible) and initially locates it at any programmer specified point on the screen. The console operator can then use the cross for the tracking procedure described above.

A call to GITCON will reposition the tracking-cross at the location specified in the call even if the console operator is using the cross when the call is made. No repositioning will occur; however, if a button ID block is queued for the specified console; the assumption is made that a queued button will initiate some action which requires the tracking-cross to be at its present coordinates.

Call Statement Format:

```
CALL GITCON (NCON, IH, IV)
```

NCON	Number of the graphics console on which the tracking-cross should appear or be relocated; only one console can be referenced through each call
IH, IV	Horizontal and vertical display grid coordinates of the point at which the cross should be placed; the cross is centered around this point

The IH and IV parameters may be omitted from any call to GITCON. If IH and IV are not supplied in a call, GITCON will display the cross at the current coordinates.

GITCOF

GITCOF returns the display grid coordinates of the cross associated with the last button pick ID block retrieved from the 1700 FETCH queue. These coordinates represent the location of the cross when that button was picked; they are not necessarily the coordinates of the cross at the time of the call to GITCOF or the coordinates of the last button picked.

NOTE

Although the tracking-cross cannot be turned off, the user can move it to an area off the screen but within the 2047 to -2048 range, i.e., coordinates (1792, 1792).

Call Statement Format:

```
CALL GITCOF (NCON, IH, IV)
```

NCON	Number of the graphics console to which the call is addressed: only one console can be referenced through each call
IH, IV	Horizontal and vertical display grid coordinates of the tracking-cross from the last button pick ID block fetched; returned as a result of the call

GITIMV

GITIMV attaches a previously defined display item to the tracking-cross so that the item moves with the cross across the screen. The initial point of the item coincides with the writing point (center) of the tracking cross.

Call Statement Format:

```
CALL GITIMV (NCON, IDDAD)
```

NCON	Octal number of the graphics console to which this call is addressed only one console can be referenced through each call
IDDAD	Associative address of the display item which should be attached to the tracking-cross

The programmer should assure that the IDDAD value he supplies in his call is defined for console NCON; if the same display item has been created at several different consoles, it will have as many different associative addresses. Use of the wrong IDDAD value aborts the job (see Appendix B).

A call to GITIMV can also be used to detach a display item from the tracking-cross. If IDDAD is set equal to zero, GITIMV will detach any item currently attached to the cross, and the item will remain at the place on the screen that it occupied when the call occurred.

GITMMV

This routine attaches a previously defined display macro to the tracking-cross so that the macro moves with the cross across the screen. A call to GITMMV displays the macro with the initial point of the macro coinciding with the writing point (center) of the tracking-cross.

If the macro contains a reset sequence, it will not be moved when the tracking-cross is moved.

Call Statement Format:

```
CALL GITMMV (NCON, MAD)
```

NCON	Octal number of the graphics console to which this call is addressed; only one console can be referenced through each call
MAD	Associative address of the macro which should be attached to the tracking-cross

A call to GITMMV can also be used to detach a display macro from the tracking-cross. If MAD is set equal to zero, GITMMV will detach any macro currently attached to the cross, and the newly displayed macro will remain at the place on the screen that it occupied when the call occurred while the previous macro is erased.

The same restrictions on MAD/NCON agreement apply to this call as apply on IDDAD/NCON agreement in a call to GITIMV.

USE OF THE DATA HANDLER

Seven of the routines in the 6000 Basic Graphics Package manipulate, store, and retrieve data from files organized in a plex data structure. One or more such local files can be defined for each graphics application job. The installation parameter, MAXNFILE, specifies the maximum number of files that can be used by a single job.

The programmer uses one file at a time. To reduce the required number of disk accesses, the data handler keeps in-core duplicates of the most used record blocks within the mass storage file. (A block is a fixed-length record; the programmer can specify an approximate length for each block in his DMINIT call.) He specifies the number of in-core duplicates to be kept, and the data handler selects that number of the most frequently used blocks from the file and duplicates them in central memory. The programmer does not need to know which blocks have duplicates in central memory at any given time; for the purpose of data storage and retrieval, he can consider that the entire file always resides in central memory.

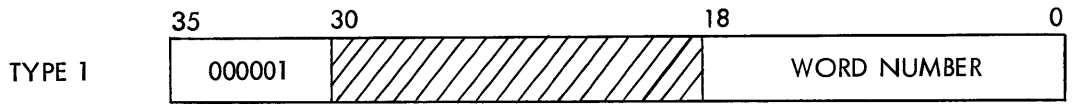
COMPONENT CODES

Data is stored within the mass storage file in word or bit spaces of variable length; these variable memory areas are called components and make up the beads of the plex data structure. Each component within a bead is accessed according to the value of bit patterns called component codes.

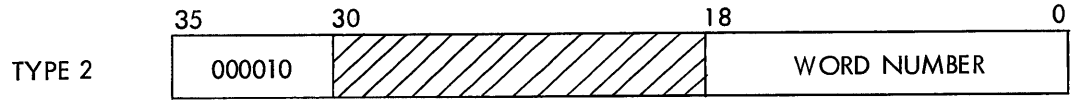
All component codes begin with a component type number, of which there are nine, followed by the particular data needed for that component type.

A given value can be inserted in a bead or retrieved from it in a number of ways; the method and component code used depend on the personal preference of the applications programmer and the requirements of his program. For instance, code 6 can be used to perform the functions of all the other component codes, but not necessarily in the most efficient manner. An operation such as the retrieval of the connecting bead addresses is best done with component code 10 and a call to DMGET.

The component codes and their formats are:



Type 1 code represents a 60-bit word as a bead component. The code can be written as 010000wordxxB in Boolean octal or as the arithmetic expression 2**30+wordxx.

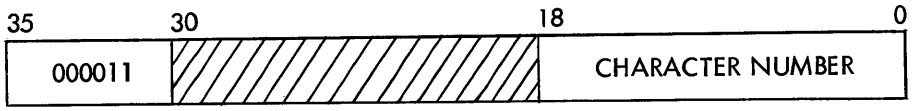


Type 2 code represents a 120-bit double-precision floating-point value as a bead component; this value is not checked for validity as a floating-point number when it is stored or retrieved. The code can be written as 020000wordxxB in Boolean octal or as the arithmetic expression 2*2**30+wordxx.

02 Component type

wordxx Position of the first 60-bit word of the value within the bead, expressed as a word number; the first word in a bead is word number 1

TYPE 3

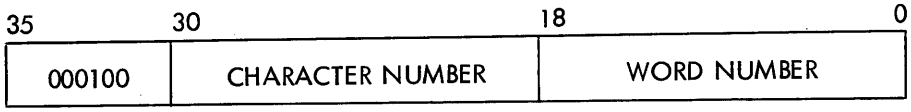


Type 3 code represents a 6-bit alphanumeric or special character as a bead component. The code can be written as 030000charxxB in Boolean octal or as the arithmetic expression 3*2**30+charxx.

03 Component type

charxx Position of the character within the bead, expressed as a character number; the first character in a bead is character number 0

TYPE 4



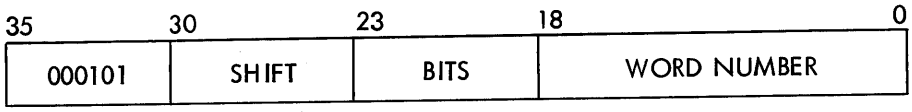
Type 4 code represents a 6-bit alphanumeric or special character within a word or word array as a bead component. The code can be written as 04charwordxxB in Boolean octal or as the arithmetic expression 4*2**30+char*2**18+wordxx.

04 Component type

char Position of the character within the word or word array, expressed as a character number; the first character in the first array word is character number 0

wordxx Position of the first word of the array within the bead, expressed as a word number; the first word in any bead is word number 1

TYPE 5



Type 5 code represents a pattern of bits within a word as a bead component. The code can be written as 05shbtwordxxB in Boolean octal or as the arithmetic expression 5*2**30+sh*2**24+bt*2**18+wordxx.

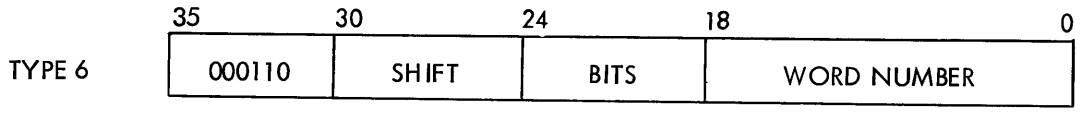
05 Component type

sh Number of bits to shift right in order to right-justify the bit pattern within the word

bt — Number of bits in the bit pattern

wordxx — Position of the word containing the pattern within the bead, expressed as a word number; the first word in any bead is word number 1

The bit pattern stored or retrieved by the type 5 code is not sign extended; the pattern stored or retrieved by type 6 code is sign extended. In a bit pattern that is not sign extended, the left-most bit in the pattern is part of the octal value of the pattern, while in a sign extended pattern the left-most bit indicates the sign of the value represented by the rest of the pattern's bits. For example if the bits [1|0|0|0] or the bits [0|1|0|0|0] are retrieved as patterns that are not sign extended, both groups of bits represent the value 10_8; however, if the same bits are retrieved as sign extended patterns, [1|0|0|0] represents the value -7_8 and [0|1|0|0|0] represents $+10_8$. This means that a negative octal value can be stored in its 1's complement form by using type 6 code.



Type 6 code represents a sign extended pattern of bits within a word as a bead component. The code can be written as 06shbtwordxxB in Boolean octal or as the arithmetic expression 6*2**30+sh*2**24+bt*2**18+wordxx.

06 — Component type

sh — Number of bits to shift right in order to right-justify the bit pattern within the word

bt — Number of bits in the bit pattern

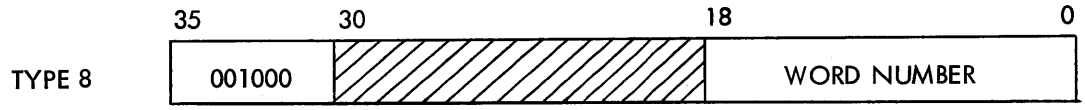
wordxx — Position of the first word containing the pattern within the bead, expressed as a word number; the first word in any bead is word number 1

Type 7 code represents an array of 60-bit words as a bead component. The code can be written as 07arylwordxxB in Boolean octal or as the arithmetic expression 7*2**30+aryl*2**18+wordxx.

07 — Component type

aryl — Length of the array in words

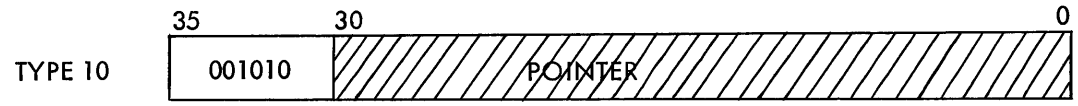
wordxx — Position of the first word of the array within the bead, expressed as a word number; the first word in any bead is word number 1



Type 8 code represents the 18-bit address portion of a word as a bead component. The code can be written as 100000wordxxB in Boolean octal or as the arithmetic expression 8*2**30+ wordxx.

10 Component type

wordxx Position of the word within the bead, expressed as a word number; the first word of any bead is word number 1



Type 10 code represents the hook (pointer) address of the next bead in a string. The code can be written as 120000000000B or as the arithmetic expression 10*2**30. When bits 0-29 are non-zero, they act as a pointer to a certain word in the next bead and/or block. Where they are zero, they simply point to the zero word in the next reference block. See Figure 6-6 for an example of pointer use.

A fragment of a sample program, showing the use of these component codes, is given after the following routine descriptions.

DMINIT

This data handler initializing routine establishes new or changes previously defined mass storage file and core storage parameters. DMINIT is used to control the number of duplicate blocks the data handler maintains in central memory, to specify which file the programmer is currently using, and to establish an approximate length for each block in the file.

Call Statement Format:

```
CALL DMINIT (IFILE, NBLK, NBSIZE)
```

IFILE Alphanumeric name of the file which the data handler should use; this identifier is one to seven characters long, left-justified within the IFILE word, and in a form and format that SCOPE will recognize as a valid file name

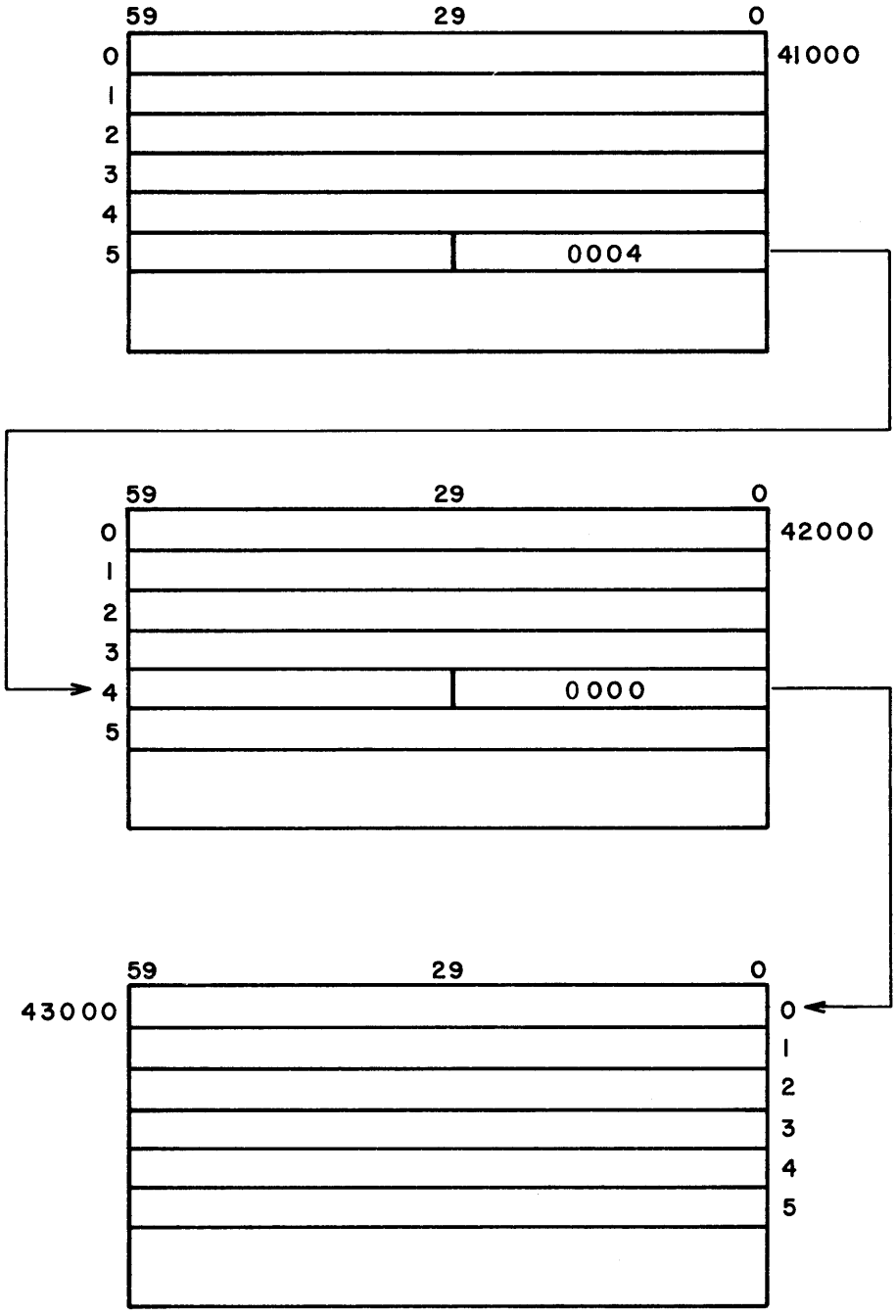


Figure 6-6. Example of Pointer Use

NBLK — Number of in-core duplicate blocks to be maintained; must be ≥ 2 for the data handler to operate efficiently

NBSIZE — Approximate size of the data blocks, expressed as an octal number of 60-bit words

The NBSIZE value specified by the application programmer is rounded up to the next highest multiple of 100_8-1; this rounding up provides for the most efficient use of the 6000 series system disk space and does not affect program execution. For example, if the programmer specifies NBSIZE equal to 90 (132_8), the blocks are assigned a size of 127_{10} (177_8).

If the programmer omits NBSIZE from his DMINIT calling sequence, the data handler uses an installation parameter to determine block size.

NBLK should be chosen carefully. If too many duplicates are maintained, the program ties up an excessive amount of central memory with its data file; if NBLK is too small, the program's response time deteriorates because the data handler must access mass storage so often. The sole purpose of maintaining duplicate blocks in central memory is to avoid these problems.

The programmer can use DMINIT to switch files during program execution. If DMINIT is called with an IFILE different from the one used in a previous call, the data handler replaces each mass storage block in the old file with its in-core duplicate if their contents differ.† The data handler then uses the new file when processing all subsequent calls from the programmer.

DMINIT can also be used to change the number of duplicate blocks maintained in central memory for the current IFILE. Each call to DMINIT will change the field length of the job as necessary to accommodate any additional blocks.

Any call to DMINIT may change the field length of the job, since the in-core portion of IFILE is appended to the field length of the rest of the job (see Figure 2-1). However, if an IFILE is already open at the time of a DMINIT call, the old file's central memory area is released before space for the new one is allocated.

If the programmer dynamically changes the field length of the job after his first call to DMINIT, the change is nullified by any subsequent calls; the data handler always begins allocating space for its file at the same location and requests a field length change just large enough to accommodate it.

---

†Information entered in the file is written in the duplicate blocks in central memory; consequently, the contents of the blocks in mass storage are not up-to-date until the data handler writes the duplicates back into the file.

DMFLSH

This routine updates the mass storage file by writing the duplicate blocks from central memory into it.† DMFLSH closes the file. This routine must be used if permanent files are to be created.

Call Statement Format:

```
CALL DMFLSH
```

No data handler routine can be used after a DMFLSH call unless another call is first made to DMINIT to re-establish the file-processing parameters.

DMDMP

The DMDMP routine prints an octal dump of the entire IFILE data file. This dump, which is formatted for easy reference to beads and string addresses, enables the applications programmer to examine the data contained in the blocks and beads of the file; empty spaces within the file are indicated but not shown.

The dump is always placed in the standard OUTPUT file.

A call to DMDMP has no effect on the contents of the data file.

Call Statement Format:

```
CALL DMDMP
```

DMGTBD

DMGTBD allocates a specified number of contiguous words from free space in the IFILE data file and defines those words as a bead. This provides the programmer with dynamic working storage. DMGTBD zeros out each word of the bead (i.e., each word is full of zeros before DMSET is called).

---

†Information entered in the file is written in the duplicate blocks in central memory; consequently, the contents of the blocks in mass storage are not up-to-date until the data handler writes the duplicates back into the file.

Call Statement Format:

```
CALL DMGTBD (N, IBEAD)
```

N	Number of 60-bit words to be allocated as a bead; N must be less than 2**18
IBEAD	Relative address of this bead within the block; returned as a result of this call

If there is no space available in IFILE for a bead of N words, IBEAD is returned equal to zero.

DMRLBD

The DMRLBD routine releases the space in IFILE occupied by beads that the programmer no longer needs. This space then becomes available for the allocation of new beads.

Call Statement Format:

```
CALL DMRLBD (IBEAD1, IBEAD2, . . . , IBEADn)
```

$IBEAD_i$	Relative bead addresses from one or more blocks, indicating the beads that should be released; an $IBEAD_i$ equal to minus zero can be used to terminate the parameter string, in addition to a right parenthesis. Up to 20 beads may be released with one call to DMRLBD.

$IBEAD_i$ is returned equal to zero when a bead is released.

DMSET

This routine places a given value in a specified position within a bead. If the value used in the call does not occupy a full 60-bit word, the value must be right-justified within the call parameter word.

Call Statement Format:

```
CALL DMSET (ICOMP, IBEAD, VAL)
```

ICOMP	Component code specifying the position within the bead that the value should occupy; ICOMP must contain one of the nine valid type codes described above

IBEAD	Relative address of the first word of the bead in which the information is to be placed
VAL	Component value to be placed in the bead; the contents of VAL must be right-justified

DMGET

This routine retrieves a previously defined value from a specific position within a bead. If the value returned by the call does not occupy a full 60-bit word, it will be right-justified within the returned call parameter word.

Call Statement Format:

```
CALL DMGET (ICOMP, IBEAD, VAL)
```

ICOMP	Component code specifying the position within the bead that the value occupies; ICOMP must contain one of the nine valid type codes defined above
IBEAD	Relative address of the first word of the bead which contains the information
VAL	Component value returned by this call; the value returned is right-justified within VAL

A call to DMGET does not destroy the information within the bead.

EXAMPLE OF BEAD USE

Figure 6-6 illustrates a bead designed to use the nine different component codes; in several cases, more than one code is used to pack a single bead word, as in words six and nineteen.

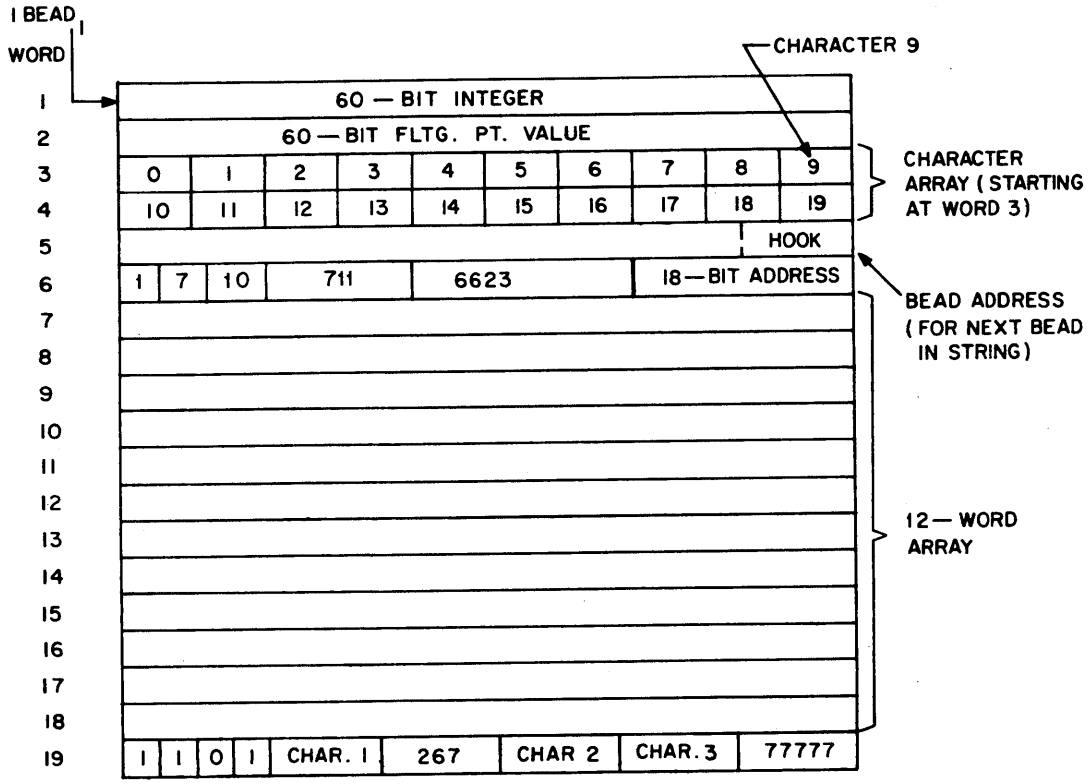


Figure 6-7. Example of Components in a Bead

The bead shown in Figure 6-6 is created and filled by the calls below:

Call	Explanation
•	
•	
•	
CALL DMINIT (7HDMFILE1, 2)	Initializes file DMFILE1, with two duplicate blocks in-core
CALL DMGTBD (19, IBEAD(1))	Establishes a bead 19 words long with a bead address returned in IBEAD(1)
•	
•	
•	
CALL DMSET (040011000003B, IBEAD(1), 1RS)	Sets character S in character position 9 in the bead array starting at word three
•	
•	
•	
CALL DMGTBD (50, IBEAD(2))	Establishes a second bead, 50 words long, for a bead string
•	
•	
•	
IHOOK = IBEAD(2) + 7B	Creates string pointer to word seven in the second bead
•	
•	
•	
CALL DMSET (010000000005B, IBEAD(1), IHOOK)	Sets the string pointer in word five of the first bead
•	
•	
•	

The other calls used are not shown because of space limitations.

Word six shows six components packed into one bead word by calls using six different component codes. The components include a 1-bit value, a 3-bit value, a 5-bit value, a 9-bit value (eight bits sign extended), a 12-bit value, and an 18-bit address.

Word 19 includes four individual bit states, three alphanumeric characters, an 8-bit value, and a 15-bit value. These nine components were placed by nine calls to DMSET.

These two words (six and 19) demonstrate the flexibility and utility of component code usage. As a further example, the character S placed by the call shown above could also have been stored by:

```
CALL DMSET (050006000003B, IBEAD(1), 23B)
```

VOLUNTARY ABORTION OF A JOB

The GIABRT routine allows the application programmer to terminate his job at any point during execution. GIABRT can be used to abort the job if a non-fatal error or another type of programming problem occurs; it can also be used to abort the job if the console user is not obtaining the desired results during an application run.

GIABRT displays an abort message, supplied by the programmer, on the screen of the graphics console at (-552, 1600); it then performs all of GICNRL's functions, enters the abort message in the SCOPE dayfile, and calls the standard SCOPE abort processor. Any exit control cards, such as DMP, are then processed.

There is no return from a call to GIABRT.

Call Statement Format:

```
CALL GIABRT (NCON, IBCD, NC)
```

NCON	Number of the graphics console that should receive the abort message; only one console can be addressed
IBCD	First word of an array buffer containing the abort message
NC	Number of characters in IBCD; must be less than 47_{10}

If the application job is servicing more than one console, one should be considered a master console to which all GIABRT messages are addressed.

HARDCOPY FILE CREATION

A console user may require a permanent record of data contained in a display. The GIPLOT routine provides a means by which such a hardcopy record can be made. Because the type of hardcopy required varies according to the job and the equipment available, GIPLOT does not actually create the hardcopy record. It creates a system file (called PLOT) of display information in a device-independent format. This file can then be used by a special driver to duplicate the display. The driver used depends upon the device at the installation; thus, specific information may be found in the handbook pertaining to the particular device installed.

The following background information is needed to understand the use of GIPLOT.

An Interactive Graphics program intersperses a sequence of calls to the 6000 Basic Graphics Package routines with manipulations of data residing in the mass storage IFILE. The displays produced by the program depend upon the console user's choice of call sequences and call parameters; he chooses these variables by making task selections and data entries from the console. The display created by one set of choices is usually modified by a subsequent set until the user obtains the desired graphics forms and information.

When the user obtains a display for which he wants a hardcopy, he makes a console entry requesting it.

The entry should then cause the program to repeat the sequence of operations that produced the display, without repeating the intermediate steps. By repeating the sequence, the parameter string which resulted in the display is reproduced. This duplicate parameter string is then used in calls to GIPLOT, rather than GIDISP or GIMAC.

An alternative to duplicating the parameter string would be the insertion of coding, similar to the following, at the end of each task which creates a display:

```
      CALL GIBUT (0,NCON, IDDT, IDDC, IDWA, IDWB, IH, IV)
      IF (IDWA.EQ.4RPLOT) GO TO 500
      CALL AETSKR
500   DO 501 I = 1, IDDC, IDWB
      CALL GIPLOT (NCON, IBUF(I),NBYTE(I),IDENT, ITYPE)
501   CONTINUE
      CALL AETSKR
      END
```

This coding checks for an entry made by a light button called PLOT and returns control to AEXEC if the button has not been picked. If the button has been picked, the contents of several display item buffers are sent to GIPLOT and then control is returned to AEXEC and the next task. The display item buffers might be stored in labeled COMMON before each GIDISP call that creates a display item in its final form.

Call Statement Format:

```
CALL GIPLOT (NCON, IBUF, NBYTE, IDENT, ITYPE)
```

NCON	Number of graphics console containing the display which this file should reproduce; only one console can be specified by each call
IBUF	Description buffer of the item to be entered in the file
NBYTE	Number of bytes contained in IBUF
IDENT, ITYPE	Information used by the programmer to identify himself and his file when it is later processed by the hardcopy driver

ADDITIONAL ROUTINES FOR DISPLAY FONT CREATION

Two routines have been added to the 6000 Basic Graphics Package library to facilitate the creation and use of display fonts. These routines are written in FORTRAN, using the other 6000 Package routines. One routine creates an alphanumeric font display resembling a teletypewriter keyboard, and the other creates a numeric font display resembling a clock face. The two routines actually display the fonts and return address parameters so that the programmer can manipulate the fonts as he would a regular display item.

GFONTA

This routine creates the keyboard-like figure shown in Figure 6-8. The alphanumeric display items IDDA and IDDN are the associative addresses of the two parts of the font. The figure is created in two parts because the parameter string describing it exceeds the length of a single EXPORT HS buffer.

BKSP is a special character for backspace (137B is the octal equivalent of the hexadecimal), SPC is a special character for space (40B is the octal equivalent of the hexadecimal), and CLEAR is a special character for clear (177B is the octal equivalent of the hexadecimal).

Call Statement Format:

```
CALL GFONTA (NCON, IH, IV, IDDA, IDDN)
```

NCON	Number of the graphics console that the font should appear on; only one console can be addressed through each call
IH, IV	Horizontal and vertical display grid coordinates of the approximate center of the display font; the font is displayed from IH -332 to IH +332 and from IV +172 to IV -172.
IDDA, IDDN	First and second associative addresses of the display font created by this call; returned as a result of the call

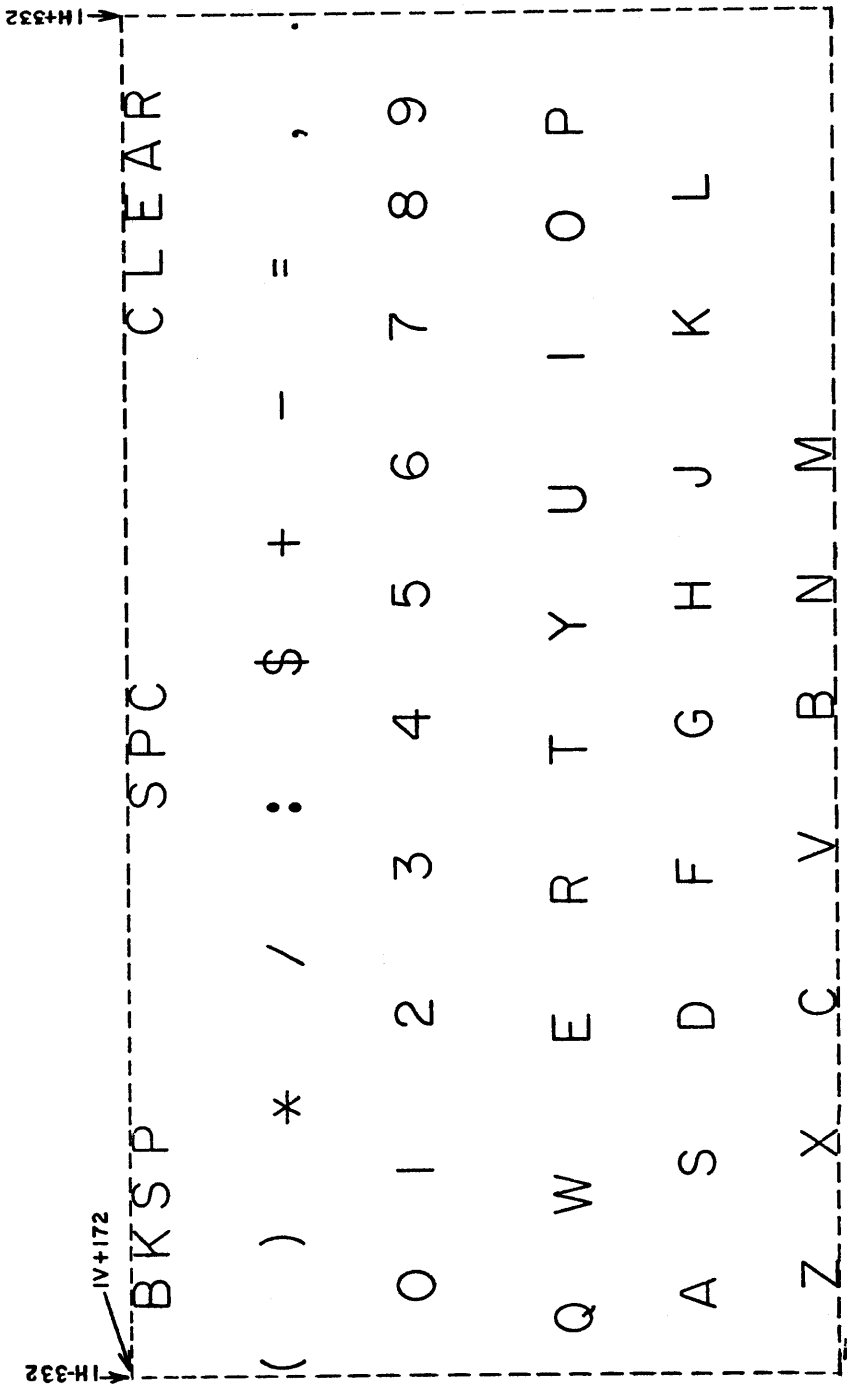


Figure 6-8. Alphanumeric Display Font

GFONTN

This routine creates a numeric font display item like the one shown in Figure 6-9.

Call Statement Format:

```
CALL GFONTN (NCON, IH, IV, IDDAD)
```

NCON	Number of the graphics console that the font should appear on; only one console can be addressed through each call
IH, IV	Horizontal and vertical display grid coordinates of the decimal point in the center of the circle; the figure is located between IH +268 and IH -244 and between IV +334 and IV -244
IDDAD	Associative address of the font display item; returned as a result of the call

The characters BKSP, SPC, and CLEAR have the same octal equivalents as they have for the alphanumeric display font described previously.

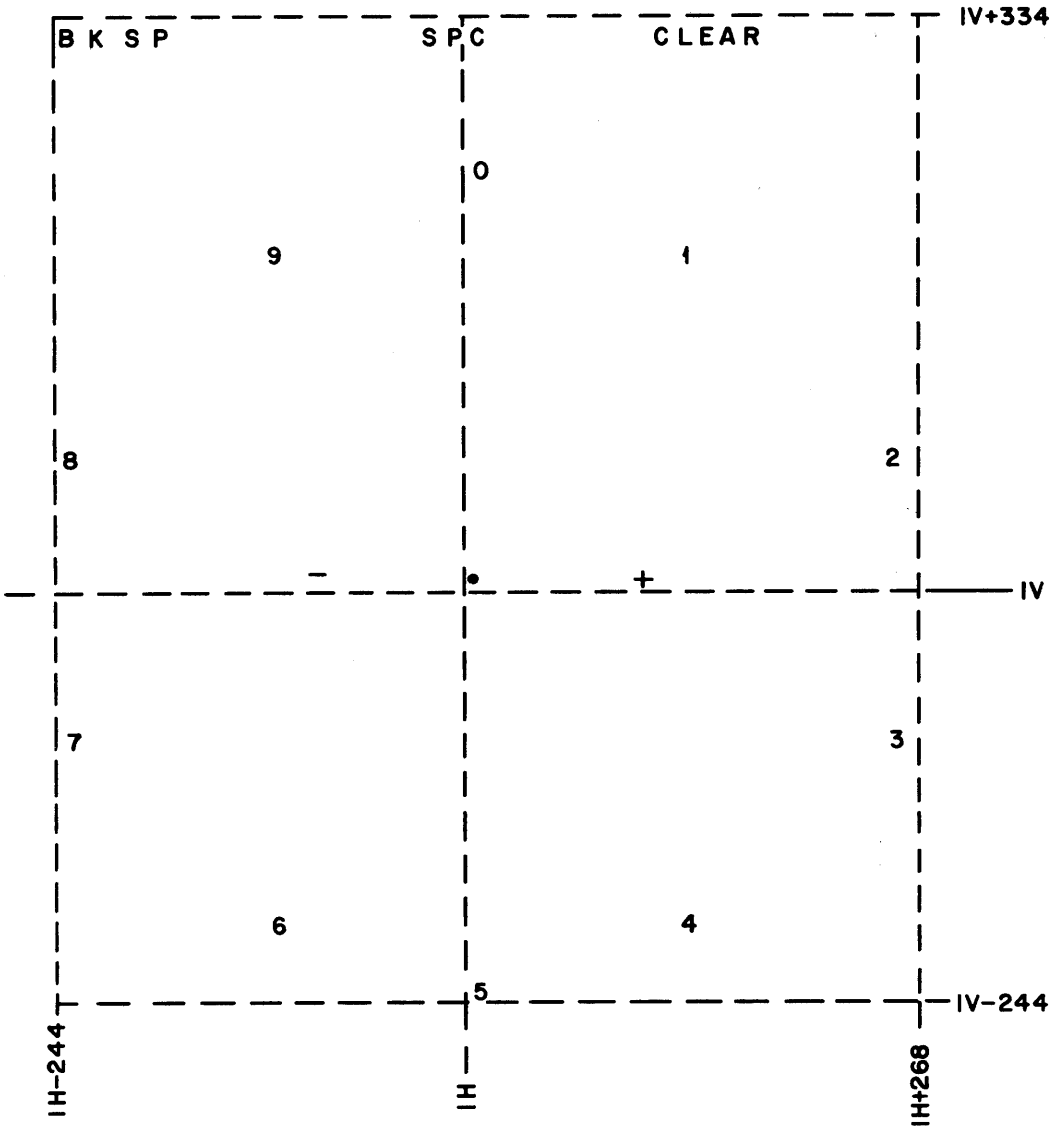


Figure 6-9. Numeric Display Font

PROGRAMMING CONSIDERATIONS 7

This section contains hints and warnings for the application programmer.

TIME ACCOUNTING

The standard SCOPE accounting procedure is used for all jobs, including graphics jobs. Sufficient time must be requested on the job card for each job to ensure its completion.

The hardware interrupt handlers of the 1700 Basic Graphics Package operate on a time-stealing basis (i.e., the 6000 series computer CPU time record is not incremented during graphics hardware interrupt handling). When graphics consoles are heavily used, the time indications in the SCOPE accounting records can be expected to lag behind clock time. Data channel use time for graphics I/O is not considered CPU time.

MEMORY ALLOTMENT AND LIST PROCESSING EFFICIENCY

The data handler is designed to make efficient use of the core space allotted to it by DMINT. The design of the algorithm used to minimize mass storage references (see Section 6) presumes that the data structure for the application will be built and referenced as a local file.

Application programs that use a widely scattered and cross-linked data structure should allot larger amounts of core storage for data handling functions. Improper assignment of core space causes slow response at the console and excessive referencing of mass storage.

DATA HANDLER COMPONENT CODES

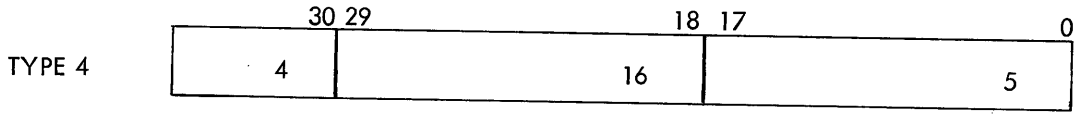
The data handler offers powerful tools for the general handling of all types of application data. The component codes required in the calls to DMSET and DMGET specify the exact location of particular pieces of data within beads. However, the bit pattern form of the component codes makes them awkward to use directly in FORTRAN programs and causes programming errors.

One solution to the problem is the convention of naming the codes through FORTRAN labeled COMMON. The application programmer can lay out his bead formats and specify a name for each component code. The name, can be typed as an INTEGER, tape and assigned a particular value by using a DATA statement. The component code can be transmitted to each

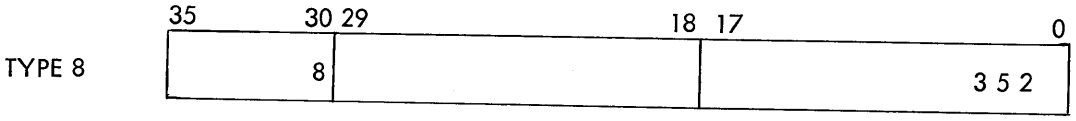
subroutine handling data through use of a COMMON/DATA/statement. All DMSET and DMGET calls may then refer to component codes by name.

The use of COMMON/DATA/ to name component codes also simplifies bead format changes. The technique can be expanded to cover assignment of bead lengths and hook values.

Another solution to the problem is to create the component codes by multiplying the desired contents for a field by the correct power of two to shift the value to its correct field position. A positive number can be shifted left N bits by multiplying 2**N. For example, the component code for



can be created by ICODE = 4*2**30 + 16*2**18 + 5, and the component code for



can be created by ICODE = 8*2**30 + 352.

Since the component code will never contain a negative number, shifting positive numbers N bits by multiplying by 2**N will always be a satisfactory solution to the problem.

DISPLAY ITEM ADDRESSES

The display item address parameter IDDAD is the link to all display buffer editing operations. The application program should provide disposition for the address of each item it displays. Display addresses of highly transient items — such as prompting messages, value register, and some light-buttons — may be kept in programmer-reserved cells in COMMON.

Most display item addresses should be an integral part of the data structure of the application and should reside in components of beads. Display items used only for control or communication can be linked to an application data structure specifically designed for that purpose. For example, light-buttons can be represented in a bead containing a specific identifier, class code, and display address (IDDAD):

IBEAD	POINTER TO NEXT BEAD
IBEAD + 1	CLASS / IDENTIFIER
IBEAD + 2	DISPLAY ADDRESS

Simple subroutines can then be written to

- Display a light-button and splice a bead into the light-button string
- Erase all light-buttons of a class and splice out their beads in a string
- Erase a specific light-button and splice out its bead

MACRO HANDLING

When a programmer writes display macros, he conserves display buffer space and allows more items to be displayed at one time. However, indiscriminate insertion and removal of macros can lead to an inefficient fragmentation of the fixed address area of the 1744 display buffer. Further, these functions represent the greatest operational load of the graphics interface and frequent use of them may affect response time.

The most frequently used macros should be placed at the beginning of the job coding. Transient macros should be removed immediately after use and before other transients are inserted. Groups of transient macros should be removed in the reverse order of their insertion for the fastest response time.

OPTIMUM TASK LENGTH

A prime consideration in programming an Interactive Graphics application is to organize the application as a series of short tasks. Interaction both implies and demands a free flow of information in two directions: from operator to application and vice versa. For this reason, tasks should be concise and well defined; the operator should be able to skip quickly ahead if the interactive processes show an obvious path to the solution of the problem at hand. Similarly, the operator should be able to jump back and forth through the application when divergence occurs until convergence to a solution is assured or until it is apparent that major parametric changes are required. In either case, it is the operator, not the computer or the application, that makes the decisions. Thus, it is obvious that the operator cannot make full use of his decision making capacity if the application programmer does not provide the operator (such as by way of button selection) with a means of exercising that capacity.

A job consisting of many small tasks (as many as 300 tasks) permits SCOPE and the application executive to operate with maximum efficiency and provides the best task execution response.

However, one should not describe 300 extremely short tasks if the logic of the application best suits a configuration with 50 somewhat longer, but logically more correct tasks. A short task on one job might be a long task on another job. The best length for any task is the length consistent with the requirements to perform one phase of a job.

NONGRAPHICS DATA HANDLER USE

The 6000 Basic Graphics Package data handler routines can be used by batch jobs that require a data file with a plex data structure. The programmer should bear in mind, however, that the CM parameter on the job card does not control the allocation of central memory when the data handler is used. The data handler appends the in-core data base to the end of the job's current field length during job execution; thus the data base would begin at the end of the memory field specified by the CM parameter.

Because the CM parameter is arbitrarily large enough to assure space for both the program coding and the loader, a great deal of central memory space may be wasted if the data handler is used.

To eliminate the unneeded space between the regular coding and the data base, the programmer can use either a REDUCE or an RFL control card (see SCOPE Reference Manual).

Figure 7-1 shows a sample deck that uses the RFL card. In this example, the field length of the job is initially $60,000_8$ central memory words to provide space for the compiler. The field length is then changed to $30,000_8$ prior to execution; during execution, eight in-core data handler file blocks are created beginning at RA + $30,000_8$.

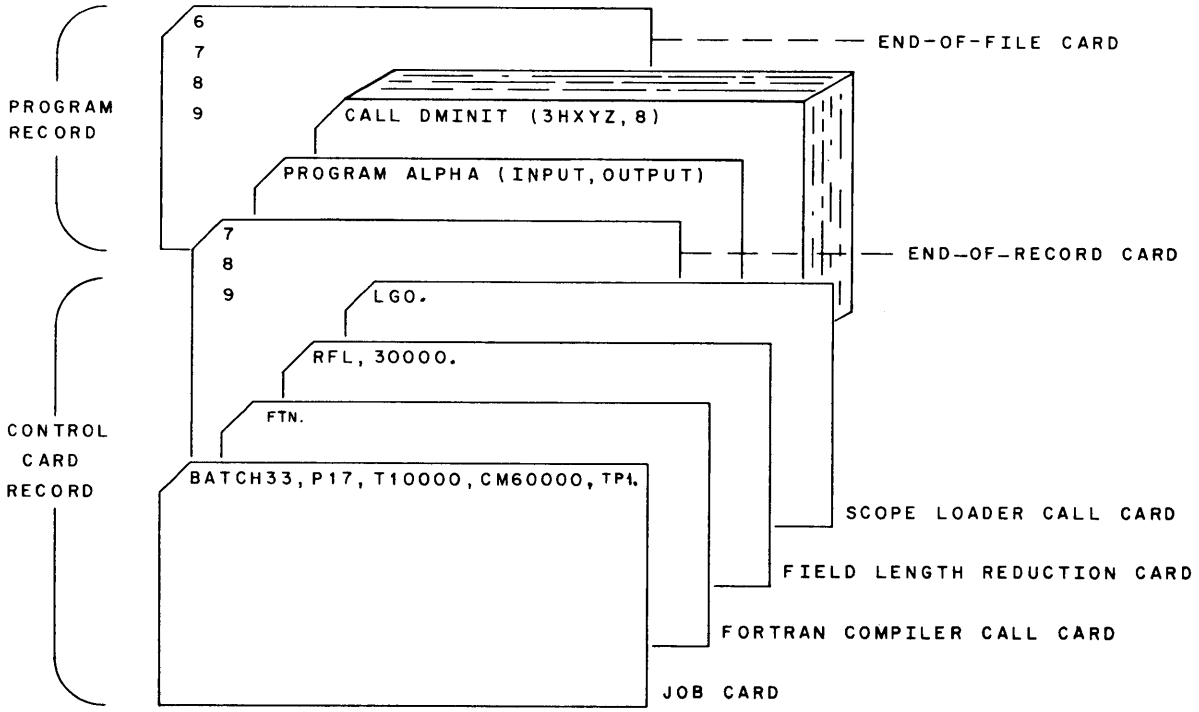


Figure 7-1. Sample Data Handler Batch Deck Using RFL

When the programmer uses an RFL card, he must be careful to leave just enough but not too much space. An easier method is shown by the sample deck in Figure 7-2, which uses a REDUCE card.

In this example, the initial field length is the same, but it is shortened before execution so that it is just large enough to accommodate the application program and the loader. The in-core data base is then appended to that field length during execution, and almost all wasted space is eliminated.

DATA HANDLER COMMON FILES OR PERMANENT FILES

The files created by the data handler during the execution run of a job are local files and are usually destroyed when the job is finished. However, these files can be declared COMMON and, if a DMFLSH has been used it can become a permanent file and subsequently used by other graphics or batch jobs. DMFLSH is used to convert them to permanent files, subsequently used by other graphics or batch jobs.

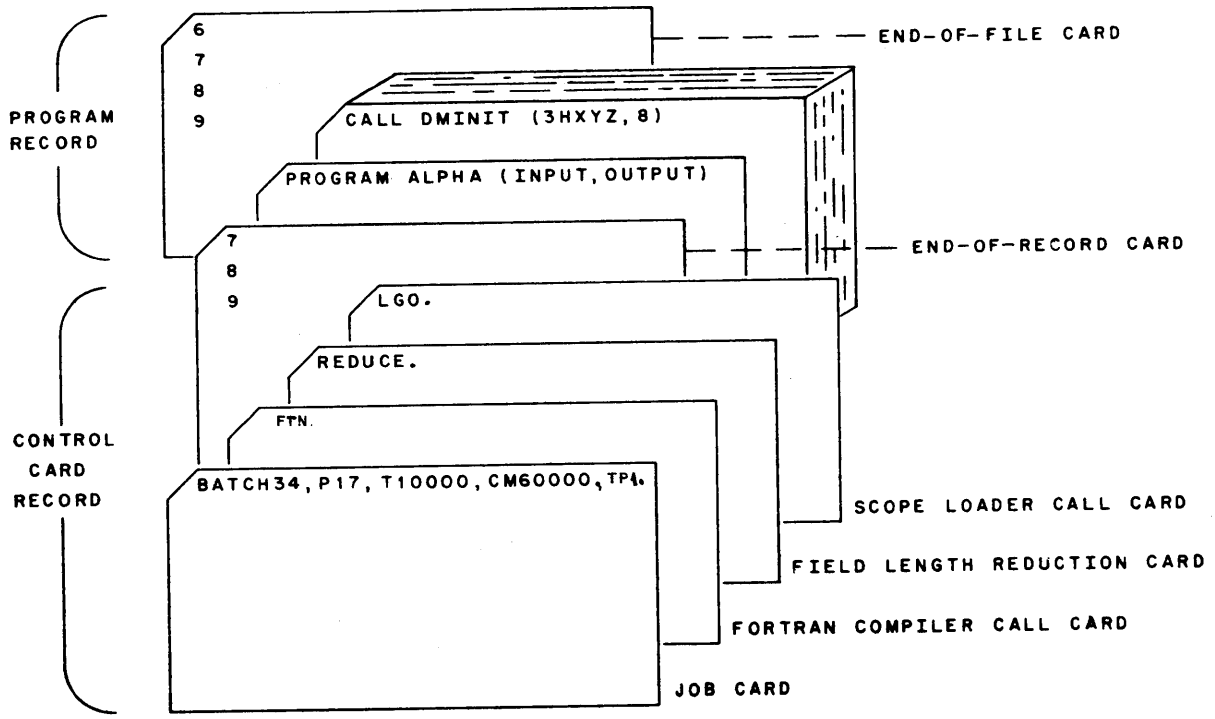


Figure 7-2. Sample Data Handler Batch Deck Using REDUCE

To create a COMMON file for use as a data handler file, the following steps are suggested:

1. Open the file by calling DMINIT.
2. Before allocating space in the file for data, use DMGTBD to obtain space in the file where all bead addresses can be saved; this call should return a bead address of 41000B.
3. On each subsequent call to DMGTBD, save the bead address in the area allocated by the first DMGTBD call.
4. At the end of the job run, call DMFLSH to update the file in mass storage.
5. After the LGO card in the control card record, insert a COMMON card with the name of the data handler IFILE on it.

To use the file during a different run,

1. Insert a COMMON card, naming the proper file, in the control card record before the LGO card.
2. Open the file using a DMINIT call, and assure that the NBSIZE parameter has the same value as during the run that created the file.
3. Get the bead addresses for data from the first part of the file, using a DMGET call with a bead address equal to 41000B (see above).
4. If any new data is stored in the file during the run, call DMFLSH at the end of the job to assure that the mass storage version of the file is up-to-date.

The following programs are examples of using a data handler file as a COMMON file.

Sample program to create the file:

```
JOB1.
FTN.
LGO.
COMMON, DMFILE.
7
8
9
      PROGRAM DMTEST1 (INPUT, OUTPUT)
      COMMON IBD (500), IPTR
C     OPEN FILE DMFILE WITH 4 IN-CORE BLOCKS
      CALL DMINIT (6LDMFILE, 4)
C     GET SPACE ON FILE WHERE BEAD ADDRESSES WILL BE SAVED
      CALL DMGTBD (500, IPTR)
C     IPTR NOW EQUALS 41000B SINCE THIS IS FIRST CALL TO DMGTBD
            •
            •
```

```
      CALL GETBEAD (N1, J1)
            •
            •
            •
      CALL GETBEAD (N2, J2)
            •
            •
            •
      CALL DMSET (ICOMP, IBD(J2), VAL)
            •
            •
            •
      CALL DMFLSH
      END
      SUBROUTINE GETBEAD (NUM, INDEX)
      COMMON IBD(500), IPTR
      ALLOCATE "NUM" NUMBER OF WORDS IN DMFILE
      CALL DMGTBD (NUM, IBD(INDEX))
      IC = 10000000000B + INDEX
C     SAVE BEAD ADDRESS RETURNED IN IBD(INDEX) IN FILE DMFILE
      CALL DMSET (IC, IPTR, IBD(INDEX)
      RETURN
      END
            •
            •
```

Sample program to use the file:

```
JOB2.
FTN.
COMMON, DMFILE.
LGO.
7
8
9
      PROGRAM DMTEST2 (INPUT, OUTPUT)
      DIMENSION IBEAD (500)
C     OPEN FILE DMFILE WITH 4 IN-CORE BLOCKS
      CALL DMINIT (6LDMFILE, 4)
            •
            •
            •
C     SET IPTR = 41000B SO BEAD ADDRESSES CAN BE RETRIEVED FROM
C     DMFILE
      IPTR = 41000B
      ICOMP = 070764000001B
      CALL DMGET (ICOMP, IPTR, IBEAD)
C     ARRAY IBEAD NOW CONTAINS ALL BEAD ADDRESSES SET DURING
C     CREATION RUN
            •
            •
            •
      CALL DMGET (ICOMPA, IBEAD(J), VAL)
            •
            •
            •
      CALL DMFLSH
      END
            •
            •
```

See Figure 2-11 and 2-12 for sample permanent file creation and execution decks. The same suggestions for using a COMMON file apply to using a permanent file.

GLOSSARY

APPLICATION PROGRAMMER – The programmer who writes graphics programs through the FORTRAN interface called the Basic Graphics Package. The programmer is usually also the graphics console user.

ARGUMENT – Parameters entered by the graphics program in a call to the Basic Graphics Package.

ASSOCIATIVE ADDRESS – Bit pattern that forms a parameter for calls to the Basic Graphics Package, i.e., contents of NCON, IDDAD, MAD, IBEAD, NAME, and IFILE.

BASIC GRAPHICS PACKAGE – Collection of FORTRAN-callable subroutines that allow access to all the graphics hardware and the data handler.

BATCH JOBS – Programs that are non-real-time and run in the background of graphics.

BEAD – Group of contiguous computer words that may be related to other beads to make up a data structure. Beads contain components and reside in blocks.

BLOCK – Mass storage logical blocks contain beads and are addressed by count; they reside on mass storage and in core

BOOLEAN OCTAL NUMBER – An octal number represented by 0s and 1s where the digit values are powers of 8. $001\ 000\ 001_2 = 101_8 = 1*8^2+0*8^1+1*8^0 = 64+0+1 = 65_{10}$

BUFFER MEMORY – A storage device attached to the 1744 controller and used by the Interactive Graphics System for storage of byte-streams during off-line display.

BUTTON – Used to initiate an action from the 274 console. There are three kinds of buttons:

- Keyboard key
- Light-button
- Prime button

BUTTON, LIGHT – Software-defined functions displayed on the control surface. They are picked with the lightpen and usually call a task to be executed. Light-buttons are items directly related to graphics program options.

BUTTON, PRIME – Allows a display item that is not defined as a button to activate a task when picked, or, is used to temporarily allow a display item to have input significance other than that written into the ID block of the item.

BYTE – A sequence of 12 adjacent binary digits (bits) operated upon as a unit.

COMMON FILE – A file of information that remains in the system, regardless of whether or not it is attached to a program.

COMPONENT – A specific bit, character, or word space within a bead. Each component has a unique address code.

DATA HANDLER – Package which optimizes the use of mass storage and of in-core data file manipulation.

DATA STRUCTURE – A logical relation used in graphics to store relationships for data retrieval.

DIRECTIVE – An IMPORT word code which informs EXPORT of the type of data that is being sent and/or what type of return data is required.

DISPLAY BUFFER – A core memory buffer in the 1744, used for refreshing displays on the 274 console in an off-line manner.

DISPLAY BYTE-STREAM – Display controller description of the item to be displayed; a serial train of control bytes.

DISPLAY, CORE – A method of graphic display using information stored in computer core memory. Core display is synonymous with on-line display.

DISPLAY ITEM – Any item displayed on the 274 console. Display items, which are byte-streams placed in the floating address area of the buffer memory, usually start with a reset sequence and end with an ID block.

DISPLAY, OFF-LINE – A method of graphic display using information stored in 1744 buffer memory which does not require direct computer intervention except to process display change information. Off-line display is synonymous with buffer memory display.

DISPLAY, ON-LINE – See DISPLAY, CORE.

ERASE – An erase function not only removes a display but also removes the pointer from the associative address table block label. The actual bytes of the item are removed from the display controller.

EXPORT/IMPORT HIGH-SPEED (HS) – A communications system which permits batch or graphics job submission to a 6000 Series computer from a remote computer.

FILE – 1. A collection of related records treated as a unit.
2. A peripheral device used by a computing system for storing data.

FRAME – A programmer-defined rectangular display on the CRT display surface which encloses the working surface. More than one frame can be specified and displayed at one time.

FRAME-SCISSORING – The process of removing the portion of a display item that exceeds the frame limits. A form of microscissoring is done when an item is rescaled such that the item is just a point.

FRAME TIME – Allowed time for any graphics program to remain at a graphics control point. Calculated by the Scheduler routine.

GRAPHICS PROGRAMS – Programs, consisting of many graphics tasks, utilizing the Basic Graphics Package subroutines.

GRAPHICS TASK – Overlay performing one operation, called by a light-button or another graphics task.

GRID, DISPLAY – An area consisting of 4096 addressable points on the H and V axes. The display grid circumscribes the display surface such that any combination of points on the H and V axes can be addressed.

HOOK – A 9-bit pointer inserted into a bead address when stringing beads.

ID BLOCK – An identification block of coded information associated with a display item. (See Section 5.)

INPUT, ALPHANUMERIC – Picking characters from a displayed font or inputting from alphanumeric keyboard.

KEYBOARD – Optional input device. There are two types:

- Function
- Alphanumeric

LIGHT-BUTTON – Software-defined functions displayed on the control surface. They are picked with the lightpen and usually call a task to be executed. Light-buttons are items directly related to graphics program options.

LIGHTPEN – A pencil-like bundle of optical fibers which senses the current vertical and horizontal coordinates of the beam and makes them available to the program in order to identify the item that the operator picked.

LIGHT-REGISTER – Specific area on the control surface provided for operator input of alphanumeric data. These registers may appear anywhere on the screen, according to the application programmer's wish.

MACRO – The display byte-stream for an item which can be displayed in a number of locations on the screen without duplication of the byte-stream.

PICK – The selection of an item with the lightpen or function keyboard.

RESET SEQUENCE – Consists of a reset byte to control beam intensity, lightpen sense, and and blink capabilities; followed by two bytes to establish horizontal and vertical display coordinates to which beam will be set with beam off. In conjunction with the last two bytes, a system-imposed 25 μsec delay permits beam driving circuits to stabilize.

RESIDENT TIME – Actual time a program has run at a control point.

RESPONSE TIME – The time period between a graphics operator command and the answer he receives.

RESULT – The output of parameters by the Basic Graphics Package.

ROLLIN – The function of transferring a graphics program from mass storage to a control point for execution.

ROLLOUT – The function of transferring a graphics program from a control point to mass storage.

SCHEDULER – A PPU program called by EXPORT to rollout or rollin a graphics program.

SCISSOR – The act of dropping an entity from the display when its coordinate parameters exceed the range of the display grid. This is a software function.

SCISSORING, FRAME – Truncating display items to fit a user-defined frame.

SCISSORING, MICRO – The act of removing items too small to be seen from the display. The cutoff point is 0.025 inch.

SINGLE PICK – A classification given to a display item to cause only the last one of this type picked to remain on the queue.

STATUS CODE – An EXPORT data word which informs the IMPORT program what buffers are available for data I/O.

STRING – A serial linking of display items, buttons, or beads.

STRING PICK – A classification given to a display item to cause each item of this type picked to be put on the end of a string of picked items.

SURFACE, CONTROL – The area reserved for light-buttons and light-registers. The area on the cathode ray tube display surface exclusive of the working surface. Programmer-defined.

SURFACE, DISPLAY – A 20-inch diameter area on the cathode ray tube screen utilized for man-machine communications. A light, blue, flicker-free display is presented to the operator due to components of the P7 phosphor coating deposited on the inside surface of the cathode ray tube screen.

SURFACE, WORKING – One of two divisions made on the cathode ray tube display surface. The working surface can be enclosed by a frame (viewing window) which is a displayed graphic.

TASK – A program and its subprograms that perform a series of calculations or logical operations. Graphics tasks are, of necessity, as short as possible to define one phase of a multiphase job.

TRACKING – The 1700 Basic Graphics Package function which maintains cognizance of the position of the lightpen as it moves across the display surface. A core-displayed tracking-cross is used as the light source for the lightpen.

TRACKING-CROSS – A software-displayed item which allows the graphics operator to use the lightpen where otherwise no light exists.

UNIT, DISPLAY GRID – The spacing between the 4096 points on the H or V axes of the display grid. A display grid unit is fixed at 0.005 inch and there are 200 display grid units per inch.

USER, CONSOLE – Person who operates a graphics console and uses an application program.

UTILITY PROGRAMS – Programs which support the graphics system, but are not directly involved in graphics program execution.

APPENDICES

6000 BASIC GRAPHICS PACKAGE ROUTINE INDEX A

GRAPHICS SYSTEM ERROR MESSAGES B

6000 PROGRAMMING DIAGNOSTICS

In addition to the standard FORTRAN compiler, SCOPE loader, and SCOPE execution error diagnostics, the Interactive Graphics System produces several additional diagnostic messages. These diagnostics appear on one or all of the system consoles, and all are entered into the SCOPE dayfile. Dayfile messages pertaining to a specific program are automatically printed with the program's listing. Diagnostics are displayed at the consoles reserved by the program.

The special Interactive Graphics error messages are listed below in alphabetic order. Class 1 messages appear only in the SCOPE dayfile (A) and/or job status (B) displays on the 6612 console screen. Class 2 messages also appear on the 274 console's display screen at screen coordinates (-552, 1600); they occur only during program execution runs. All messages issued by 6000 Basic Graphics Package routines contain the name of the Package call in which the error was made and are prefaced by a message which states the name of the task overlay in which the erroneous call occurred. Class 3 messages are generated by the 6000 software and appear only on the 274 console screen.

6000 INPUT/OUTPUT ERRORS

JANUS also produces some error messages; these appear only in a program's output file.

1700 ABORT ERRORS

Class 4 error messages are produced by the 1700 software and appear on the 274 console screen, but are not sent to the 6000 series machine. If a Class 4 message is associated with a fatal error, it sends an abort flag from the 1700 to the 6000. The Scheduler detects the abort flag and issues the Class 1 error message GRAPHICS ABORT.

The routine sending each message is specified at the end of the paragraph titled Meaning.

Message	Error Type	Class	Meaning	Page References
AEFILE READ ERROR	Fatal	1	Parameters in the file environment table indicate a disk read error when control is returned to AEFILE from SCOPE; sent by AEFILE.	2-22
BAD CALL CODE RETURNED –GIANE	Fatal	2	Buffer returned by IMPORT at end of console alphanumeric input does not contain expected valid identification code; sent by GIANE.	
BAD CALL PARAMETER	Fatal	1	EXPORT has encountered a service request (at RA+76_8 of a graphics job's control point area) with meaningless contents; sent by EXPORT.	
CONTROL CARD ERROR	Fatal	3	During scheduling a bad control card has been detected or an EXIT card is detected; sent by ERPRO.	
BAD NAME CHECK rcrdnam	Fatal	1	The name of record rcrdnam in source file does not correspond to any entry in the file index; sent by AELOAD.	2-20
PP CALL ERROR – AUTO RECALL	Fatal	3	During autocall, a bad PP call is detected; sent by ERPRO.	
BYTE ARRAY EXCEEDS 255 – GUBYTE	Non-Fatal	1	There are only 8 bits in the 1700 Package version of the N parameter, so N in this call cannot exceed 255_{10}; sent by GUBYTE.	6-46
BYTE ARRAY INDEX ZERO – GUBYTE	Non-Fatal	1	The N parameter in this GUBYTE call is zero, so the call is ignored; sent by GUBYTE.	6-38 6-46
ARITHMETIC MODE ERROR	Fatal	3	Arithmetic error causes the job to abort; sent by ERPRO.	
CP TIME LIMIT ABORT	Fatal	2	The central processor uses the maximum amount of time specified, then aborts; sent by ERPRO.	
DISPLAY ITEM BUFFER EXCEEDED –GIDISP	Fatal	2	The total number of bytes in the user's IBUF (excluding GIDISP header bytes and trailing ID bytes) exceeds the 310 decimal maximum; sent by GIDISP.	6-38, 6-49, -50 G-2
DISPLAY ITEM NBYTE EQUALS ZERO – GIDISP	Non-Fatal	1	Since the description buffer is empty, the call is ignored; sent by GIDISP.	6-38, 6-49, -50
DUPLICATE FILE NAMES	Fatal	1	An executive program has detected two COMMON files with the same name; sent by AELOAD or AEFILE.	

Message	Error Type	Class	Meaning	Page References
ECS PARITY ERROR	Fatal	2	There was a parity error during an external storage move; sent by ERPRO.	
EMPTY FILE filenam	Fatal	1	Issued by AEDUMP; the first record in source file filenam indicates that the file was created from an empty random file; sent by AELOAD.	2-23
EOR NOT READ ON TASK LOAD	Fatal	1	Parameters in the file environment table indicate a disk read error during a task call when control is returned to AEXEC from SCOPE; sent by AEXEC.	
EXHS NOT ACTIVE	Fatal	1	There is no communication between EXPORT and IMPORT when an initial graphics roll-out or control point initialization is requested. To continue, the operator may initialize EXHS or drop the job issuing the message; sent by SCH or IGS.	
FORMAT ERROR FIRST DATA RECORD	Fatal	1	The first data record of the input file for the job contains no cards or an illegal file name; names must be seven or fewer characters, and must contain no special characters. Produced by AEXEC.	2-9
GICOPY ADDR ERR, NCON y	Non-Fatal	4	The 6000 Package routine named has sent the buffer translator an invalid IDDAD, IDDADI, or MAD parameter for use on console y; sent by buffer translator.	6-52
GIDISP BUFFER OVERFLOW, NCON y	Non-Fatal	4	Console y controller memory has overflowed because of the named call; sent by buffer translator.	
GIERAS ADDR ERR, NCON y	Non-Fatal	4	The 6000 Package routine named has sent the buffer translator an invalid IDDAD, IDDADI, or MAD parameter for use on console y; sent by buffer translator.	6-46
GIMAC BUFFER OVERFLOW, NCON y	Non-Fatal	4	Console y controller memory has overflowed because of the named call; sent by buffer translator.	
GIMAC CALL IGNORED – NBYTE = 0	Non-Fatal	1	Since the programmer has specified that his description buffer is empty, this call is ignored; sent by GIMAC.	
GIMAC ADDR ERR, NCON y	Non-Fatal	4	The 6000 Package routine named has sent the buffer translator an invalid IDDAD, IDDADI, or MAD parameter for use on console y; sent by buffer translator.	6-48

Message	Error Type	Class	Meaning	Page References
GIMOVE ADDR ERR, NCON y	Non-Fatal	4	See above; sent by buffer translator.	6-53
GITIMV ADDR ERR, NCON y	Non-Fatal	4	See above; sent by buffer translator.	6-56
GIMOVE ADDR ERR, NCON y	Non-Fatal	4	See above; sent by buffer translator.	6-56
ITEM DESCRIPTION BUFFER TRUNCATED			NBYTE is changed because of invalid data. The last parameter in the buffer is truncated or an invalid call code is found; sent by GVAL (called by GIDISP, GIMAC).	6-49
GUAN CALL IGNORED – NC IS ZERO OR NEGATIVE	Non-Fatal	1	Self-esplanatory; sent by GUAN.	6-38, 6-40
GUARCG CALL IGNORED – KSHOW ILLEGAL	Non-Fatal	1	The KSHOW parameter is negative or greater than 5; sent by GUARCG.	6-38, 6-45
GUARCG CALL IGNORED – ZERO RADIUS ARC	Non-Fatal	1	If IH1, IV1 or IH2, IV2 equals IHC, IVC no arc can be generated; sent by GUARCG.	6-38, 6-45
GUMACG ADDR ERR, NCON y	Non-Fatal	4	The 6000 Package routine named has sent the buffer translator an invalid IDDAD, IDDADI, or MAD parameter for use on console y; sent by buffer translator.	6-46
GUSEGA CALL IGNORED – N ZERO OR NEGATIVE	Non-Fatal	1	Self-explanatory; sent by GUSEGA.	6-38, 6-44
IGS CTL PT nn INITIALIZED	Fatal	1	It is initialized from PPU program IGS, where nn is the control point initialized; sent by IGS.	
IGS CONTROL POINT nn RELEASED	Non-Fatal	1	nn is the number of the IGS control point which was released; sent by SCH.	
ILLEGAL CALL TO IGS	Non-Fatal	1	The routine IGS detects an illegal call to itself; sent by IGS.	
ILLEGAL CALL TO SCH	Fatal	1	The Scheduler has been called illegally; sent by SCH.	
ILLEGAL COORDINATE – GITCON	Fatal	2	One of the programmer's tracking-cross corrdinates is beyond the extent of the display grid (not between -2048 and +2048); sent by GITCON.	3-6, 6-55, -56

Message	Error Type	Class	Meaning	Page References
ILLEGAL COORDINATE RETURNED – GITCOF	Fatal	2	One of the tracking-cross coordinates from the last button pick is not within the display grid (is less than -2048 or greater than +2048); sent by GITCOF.	3-6, 6-55, -56
ILLEGAL IBEAD – DMGET	Fatal	2	Programmer's bead address either: • Has an index = 0 • Has a block number = 0 • Has a block number greater than the number of existing blocks Sent by DMGET.	6-6, 6-66
ILLEGAL IBEAD – DMRLBD	Non-Fatal	1	Same as above; sent by DMRLBD.	6-6, 6-65
ILLEGAL IBEAD – DMSET	Fatal	2	Same as above; sent by DMSET.	6-6, 6-65
ILLEGAL IBEAM – GUSEG	Non-Fatal	1	The programmer's beam control parameter is not either 0 or 1; sent by GUSEG.	6-43
ILLEGAL IBEAM – GUSEGS	Non-Fatal	1	Same as above; sent by GUSEGS.	6-41
ILLEGAL ICOMP – DMGET	Fatal	2	The programmer's component code contained either: • Type code = 0 or 9, or greater than 10 • Word or character number greater than the size of the bead specified by the accompanying IBEAD value Sent by DMGET.	6-58, 6-66
ILLEGAL ICOMP – DMSET	Fatal	2	Same as above; sent by DMSET.	6-58, 6-65
ILLEGAL NBLK – DMINIT	Fatal	2	The number of data handler data blocks that the programmer wishes kept in core as copies is either: • Less than the minimum of 2 that the handler needs to function properly • Larger than the number that will fit in core Sent by DMINIT.	6-63
ILLEGAL NBSIZE – DMINIT	Fatal	2	The block size specified in this call is larger than the maximum permissible size of an in-core data base; sent by DMINIT.	6-7, 6-63
ILLEGAL NUMBER OF WORDS REQUESTED – DMGTBD	Fatal	2	The programmer is trying to define a bead with a length (N parameter) ≤ 0 or $\geq 2^{18}$; sent by DMGTBD.	6-64

Message	Error Type	Class	Meaning	Page References
INCORRECT ICODE – GICOPY	Non-Fatal	1	The programmer's reset control code is not a form or value significant to the 1700 version of this routine; a significant ICODE value will be substituted for the one supplied; sent by GICOPY.	6-52
INCORRECT ICODE –GIMOVE	Non-Fatal	1	Same as above; sent by GIMOVE.	6-53
INCORRECT ICODE – GURSET	Non-Fatal	1	Same as above; sent by GURSET	6-39
INCORRECT NCON – GIFID	Fatal	2	The programmer is trying to fetch a single pick ID block from a console other than the one from which the last button pick ID was fetched. The GIFID NCON must always agree with the NCON of the last GIBUT call; sent by GIFID.	6-31
INCORRECT NCON – GIFSID	Fatal	2	Same as above; same as GIFSID; sent by GIFSID.	6-32
INVALID IDDADI – GICOPY	Fatal	2	The user has specified an incorrect associative address of the item to be duplicated by GICOPY; sent by GICOPY.	
ITEM NOT DISPLAYED ON THIS NCON – GITIMV	Non-Fatal	2	Item to be attached to the tracking-cross is not on the console referenced; sent by GITIMM.	
ITEM DESCRIPTION BUFFER TRUNCATED	Non-Fatal	1	When an invalid call code or a truncated call is found in a buffer submitted to GIDISP or GIMAC for output to the 1700, this message is displayed; sent by GVALID.	
ITEM NOT CREATED FOR THIS NCON – GITIMV	Non-Fatal	2	The IDDAD value given does not exist for this console; either the NCON or the IDDAD parameter supplied in this call is wrong; sent by GITIMM.	6-56
JOB ABORT – SYSTEM	Fatal	2	The SCOPE system has aborted the job; sent by ERPRO.	
JOB HUNG IN AUTO RECALL	Fatal	3	A job in autorecall has no PPU activity.	
JOB NOT ATTACHED TO RANDOM TASK FILE	Fatal	1	Either: • AEDUMP could not find the file named in its first parameter field • AEXEC could not find the file named on its graphics COMMON file name parameter card If the file name is both legal and correct as given, then the file has not been attached to the job by a control card; sent by AEDUMP or AEXEC.	2-6, 2-8, 6-23, 6-2

Message	Error Type	Class	Meaning	Page References
JOB WILL BE RERUN	Fatal	2	The 6612 operator has made a rerun request for this job; sent by ERPRO.	
MACRO BUFFER LENGTH EXCEEDED – GIMAC	Fatal	2	The total number of bytes in the user's IBUF (excluding GIMAC header and trailer bytes) exceeds the maximum of 310 decimal; sent by GIMAC.	6-38, 6-48, 6-51 G-1
MACRO NOT CREATED FOR THIS NCON – GITIMV	Fatal	2	The MAD value given does not exist for this console; either the NCON or the MAD parameter supplied in this call is wrong; sent by GITIMM.	
MAD ARRAY INDEX ZERO – GUMACG	Non-Fatal	1	The N parameter given in this call indicates that no macro should be created, so this call is ignored; sent by GUMACG.	6-38, 6-46
NBSIZE DIFFERS FROM PREVIOUS DEFINITION – DMINIT	Fatal	2	DMINIT has been called with a block size different from the block size declared by a previous job using IFILE; sent by DMINIT.	6-63
NBYTE EXCEEDS MBYTE – GUAN	Non-Fatal	1	This graphics utility call has produced more bytes in IBUF than the programmer wants; sent by GUAN.	6-38, 6-39
NBYTE EXCEEDS MBYTE – GUARCG	Non-Fatal	1	Same as above; sent by GUARCG.	6-38, 6-45
NBYTE EXCEEDS MBYTE – GUBYTE	Non-Fatal	1	Same as above; sent by GUBYTE.	6-38, 6-46
NBYTE EXCEEDS MBYTE – GUMACC	Non-Fatal	1	Same as above; sent by GUMACC.	6-38, 6-46
NBYTE EXCEEDS MBYTE – GURSET	Non-Fatal	1	Same as above; sent by GURSET.	6-38, 6-39
NBYTE EXCEEDS MBYTE – GUSEG	Non-Fatal	1	Same as above; sent by GUSEG.	6-38, 6-43
NBYTE EXCEEDS MBYTE – GUSEGA	Non-Fatal	1	Same as above; sent by GUSEGA.	6-38, 6-44
NBYTE EXCEEDS MBYTE – GUSEGI	Non-Fatal	1	Same as above; sent by GUSEGI.	6-38, 6-43
NBYTE EXCEEDS MBYTE – GUSEGS	Non-Fatal	1	Same as above; sent by GUSEGS	6-38, 6-41
NCON ERROR	Fatal	1	EXPORT has detected a service request (RA + 76_8 of a graphics job's control point area) with an invalid NCON parameter; either the console being addressed does not exist, or it is not attached to this job; sent by ERPRO.	6-19

Message	Error Type	Class	Meaning	Page References
NC RETURNED GREATER THAN MAXIMUM– GIANE	Non-Fatal	1	System error; more characters were returned than were requested; sent by GIANE.	6-34
NC TOO LARGE – GIANE	Fatal	2	NC can be no greater than 80.	6-34
NC TOO LARGE – GIANS	Fatal	2	The programmer is willing to accept too many characters; the maximum is 80; sent by GIANS.	6-34
NC ZERO OR NEGATIVE – GIANE	Non-Fatal	2	Self-explanatory.	
NC ZERO OR NEGATIVE – GIANS	Non-Fatal	2	Self-explanatory.	
NO ACTIVE GRAPHICS CP.	Fatal	1	The Scheduler cannot find a graphics control point to which the program can be assigned. The system operator has not assigned a control point for graphics use, so the job cannot be executed; no programming error has occurred; sent by SCH.	
NO DATA HANDLER FILE OPEN – DMDMP	Fatal	2	The programmer is trying to dump a non-existent IFILE; either: • DMINIT has not yet been called • DMINIT has not been called since the last DMFLSH call Sent by DMDMP.	6-61, 6-63, 6-64
NO DATA HANDLER FILE OPEN – DMGET	Fatal	2	The programmer is trying to obtain data in a non-existent IFILE; see above. Sent by DMGET.	6-61, 6-63, 6-64
NO DATA HANDLER FILE OPEN – DMGTBD	Fatal	2	The programmer is asking for space in a non-existent IFILE; see above. Sent by DMGTBD.	6-61, 6-63, 6-64
NO DATA HANDLER FILE OPEN – DMRLBD	Fatal	2	The programmer is trying to clear space in a non-existent IFILE; see above. Sent by DMRLBD.	6-61, 6-63, 6-65
NO DATA HANDLER FILE OPEN – DMSET	Fatal	2	The programmer is attempting to place data in a non-existent IFILE; see above. Sent by DMSET.	6-61, 6-63, 6-65
NO INITIAL POINT GENERATED – GUSEG	Fatal	2	GUSEG has been called without a previous GUSEGI or GUSEGS call to initialize the figure that the programmer wants to generate; sent by GUSEG.	6-42, 6-43

Message	Error Type	Class	Meaning	Page References
NO TASK NAME IN BUTTON ID – AETSKR	Non-Fatal	1	Produced by AEXEC when AETSKR has been called; IDWA and IDWB of the button in the FETCH queue do not contain information that can be used to load a task; either: • IDWA = 0 • The ID block ends short of IDWA • The bit pattern in IDWA and IDWB does not match a task name in the index	6-20, 6-48, 6-49
NO TC CO-ORDINATES THIS NCON – GITCOF	Non-Fatal	1	No tracking-cross coordinates can be returned because the NCON of the last button picked does not match the NCON supplied in this call; sent by GITCOF.	6-55
OPERATOR DROP	Fatal	1	The 6612 operator used a type-in to abort an IGS job; sent by GABT.	
PP CALL ERROR – ABORT	Fatal	2	Self-explanatory; sent by ERPRO.	
PREFIX TABLE FORMAT ERROR – AEFILE	Fatal	1	AEFILE has detected an illegal prefix table while creating the task file from the overlay scratch file.	
PROGRAM NAME NOT IN FILE CATALOG	Fatal	2	AETSKC cannot find the required task in the directory of the job's graphics task file; sent by AETSKC.	2-20, 6-20
QUEUE TABLE FULL	Fatal	1	The Scheduler has no room in its graphics input queue for this job, so the job cannot be assigned to a control point. The job should be run again when there are fewer graphics jobs in the system; sent by SCH.	
RETURN ADDRESS OVERLAYED OR MISSING – AE	Non-Fatal	1	Either AETSKC has never been called or the return address of AETSKC has been overwritten by a task load since the last call. After issuing the message, AERTRN exits to AETSKR; sent by AERTRN.	
TASK taskname ADDRESS nnnnnn		2	Issued by the AEXEC error processor before all fatal and nonfatal error messages; taskname contains the name of the task overlay in which the error occurred. nnnnnn is the octal return address of the routine which discovered the error. It will normally occur in the user's program; sent by AEXEC.	

Message	Error Type	Class	Meaning	Page References
TASK ******** ADDRESS nnnnnn		2	AEXEC issues this when it appears that the contents of the reservation word in AEXEC for the current task have been destroyed or when jobs are being run outside applications executive interface; sent by AEXEC.	H-2, H-3
TOO MANY DATA HANDLER FILES CREATED – DMINIT	Fatal	2	DMINIT has been called to create more than the installation-specified number of data handler files; sent by DMINIT.	6-63
TOO MANY FILES ATTACHED TO A GRAPHICS JOB	Fatal	3	More than eight files are attached to the job which the Scheduler is attempting to roll out; sent by SCH.	
TOO MANY IGS CONTROL POINTS	Non-Fatal	1	The operator has assigned more than two control points to IGS; sent by IGS.	
1700 ABORT	Fatal	1	Either a Class 4 error or one of the system problems mentioned in the IGS Operator's Guide has caused the 1700 to abort the job. Error correction may have to be done through the 1700; sent by SCH.	

CHARACTER CODE EQUIVALENTS C

6000 Internal Display Code	Printed Character (Standard 6000 set)	1713 Teletypewriter	274 Display Character†††	Alphanumeric Keyboard††††	1700 Hexadecimal Internal Code†	Octal Equivalent of Hex.	6000 Hollerith (punched card rows)††
01	A	A	A	A	41	101	12, 1
02	B	B	B	B	42	102	12, 2
03	C	C	C	C	43	103	12, 3
04	D	D	D	D	44	104	12, 4
05	E	E	E	E	45	105	12, 5
06	F	F	F	F	46	106	12, 6
07	G	G	G	G	47	107	12, 7
10	H	H	H	H	48	110	12, 8
11	I	I	I	I	49	111	12, 9
12	J	J	J	J	4A	112	11, 1
13	K	K	K	K	4B	113	11, 2
14	L	L	L	L	4C	114	11, 3
15	M	M	M	M	4D	115	11, 4
16	N	N	N	N	4E	116	11, 5
17	O	O	O	O	4F	117	11, 6
20	P	P	P	P	50	120	11, 7
21	Q	Q	Q	Q	51	121	11, 8
22	R	R	R	R	52	122	11, 9
23	S	S	S	S	53	123	0, 2
24	T	T	T	T	54	124	0, 3
25	U	U	U	U	55	125	0, 4
26	V	V	V	V	56	126	0, 5
27	W	W	W	W	57	127	0, 6
30	X	X	X	X	58	130	0, 7
31	Y	Y	Y	Y	59	131	0, 8
32	Z	Z	Z	Z	5A	132	0, 9

Standard FORTRAN Characters

†8-bit ASCII, used for communication with 1713 Teletypewriter
††0, 11 is equivalent to 11, 8, 2 and 0, 12 is equivalent to 12, 8, 2
†††CLEAR, SPC, BACKSPACE, TAB and EOM have input control significance only and are not available for software alphanumeric pick processing.
††††The only legal EOM characters are RETURN and the characters in the alphanumeric font. BKSP, SPC and CLEAR are not legal EOM characters.

CHARACTER CODE EQUIVALENTS (Cont'd)

6000 Internal Display Code	Printed Character (Standard 6000 set)	1713 Tele-type-writer	274 Display Charac-ter†††	Alpha-numeric Keyboard††††	1700 Hexa-decimal Internal Code†	Octal Equiv-alent of Hex.	6000 Hollerith (punched card rows)††
33	0	0	0	0	30	60	0
34	1	1	1	1	31	61	1
35	2	2	2	2	32	62	2
36	3	3	3	3	33	63	3
37	4	4	4	4	34	64	4
40	5	5	5	5	35	65	5
41	6	6	6	6	36	66	6
42	7	7	7	7	37	67	7
43	8	8	8	8	38	70	8
44	9	9	9	9	39	71	9
45	+	+	+	+	2B	53	12
46	-	-	-	-	2D	55	11
47	*	*	*	*	2A	52	11, 4, 8
50	/	/	/	/	2F	57	0, 1
51	(	(	(	(	28	50	0, 4, 8
52	)	)	)	)	29	51	12, 4, 8
53	$	$	$	$	23	43	11, 3, 8
54	=	=	=	=	3D	75	3, 8
55	blank	space		space	20	40	space
56	,	,	,	,	2C	54	0, 3, 8
57	.	.	.	.	2E	56	12, 3, 8
60	≡	#	backspace	←	5F	137	0, 6, 8
61	[	[	?	?	5B	133	7, 8
62	]	]			5D	135	0, 2, 8
63	:	:	:	:	3A	72	2, 8
64	≠				27	47	4, 8
65	→	@	tab	TAB	40	100	0, 5, 8
66	∨(OR)	/	clear	RUBOUT	7F	177	0, 11

Standard FORTRAN Characters

†8-bit ASCII, used for communication with 1713 Teletypewriter
††0, 11 is equivalent to 11, 8, 2 and 0, 12 is equivalent to 12, 8, 2
†††CLEAR, SPC, BACKSPACE, TAB and EOM have input control significance only and are not available for software alphanumeric pick processing.
†††† The only legal EOM characters are RETURN and the characters in the alphanumeric font. BKSP, SPC and CLEAR are not legal EOM characters.

CHARACTER CODE EQUIVALENTS (Cont'd)

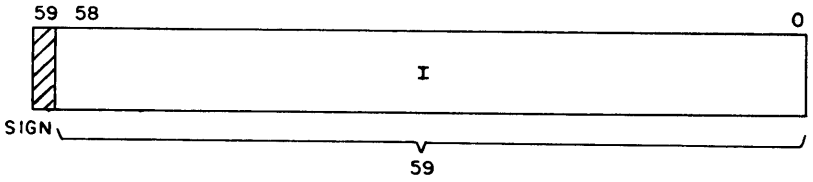
6000 Internal Display Code	Printed Character (Standard 6000 set)	1713 Teletypewriter	274 Display Character†††	Alphanumeric Keyboard††††	1700 Hexadecimal Internal Code†	Octal Equivalent of Hex.	6000 Hollerith (punched card rows)††
67	∧(AND)	%	EOM	RETURN	3F	77	0, 7, 8
70	↑	↑		↑	23	43	11, 5, 8
71	↓	↓		↓	5C	134	11, 6, 8
72	<	<		<	3C	74	0, 12
73	>	>		>	3E	76	11, 7, 8
74	≤	&		&	26	46	5, 8
75	≥	?			5E	136	12, 5, 8
76	¬(NOT)	←		←	7C	174	12, 6, 8
77	;	;		;	3B	73	12, 7, 8

†8-bit ASCII, used for communication with 1713 Teletypewriter
††0, 11 is equivalent to 11, 8, 2 and 0, 12 is equivalent to 12, 8, 2
†††CLEAR, SPC, BACKSPACE, TAB and EOM have input control significance only and are not available for software alphanumeric pick processing.
††††The only legal EOM characters are RETURN and the characters in the alphanumeric font. BKSP, SPC and CLEAR are not legal EOM characters.

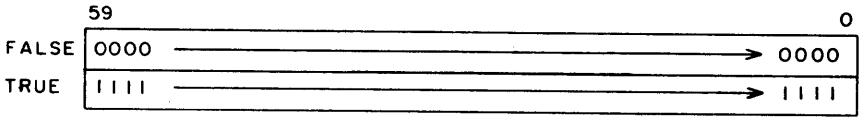
6000 SERIES CENTRAL MEMORY WORD ORGANIZATION

D

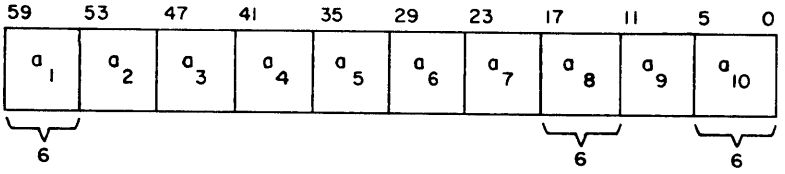
INTEGER



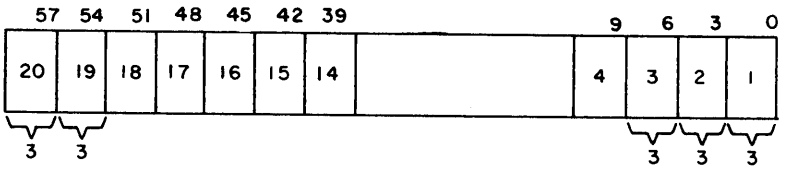
LOGICAL



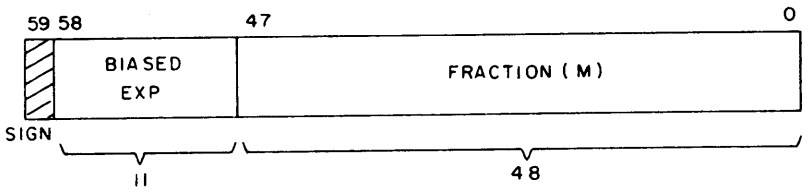
HOLLERITH BCD AND DISPLAY CODE



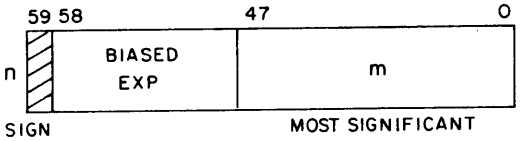
OCTAL

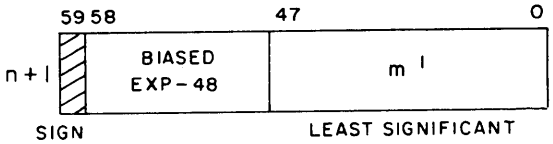


REAL

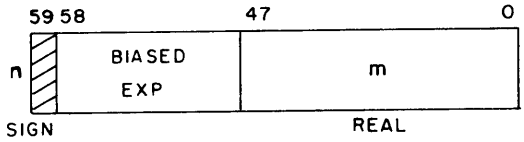


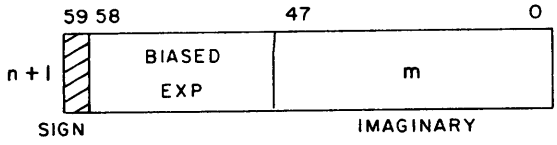
DOUBLE PRECISION





COMPLEX





HEXADECIMAL/OCTAL CONVERSION TABLE E

Hexadecimal	Octal	Hexadecimal	Octal	Hexadecimal	Octal
0	00	28	50	50	120
1	01	29	51	51	121
2	02	2A	52	52	122
3	03	2B	53	53	123
4	04	2C	54	54	124
5	05	2D	55	55	125
6	06	2E	56	56	126
7	07	2F	57	57	127
8	10	30	60	58	130
9	11	31	61	59	131
A	12	32	62	5A	132
B	13	33	63	5B	133
C	14	34	64	5C	134
D	15	35	65	5D	135
E	16	36	66	5E	136
F	17	37	67	5F	137
10	20	38	70	60	140
11	21	39	71	61	141
12	22	3A	72	62	142
13	23	3B	73	63	143
14	24	3C	74	64	144
15	25	3D	75	65	145
16	26	3E	76	66	146
17	27	3F	77	67	147
18	30	40	100	68	150
19	31	41	101	69	151
1A	32	42	102	6A	152
1B	33	43	103	6B	153
1C	34	44	104	6C	154
1D	35	45	105	6D	155
1E	36	46	106	6E	156
1F	37	47	107	6F	157
20	40	48	110	70	160
21	41	49	111	71	161
22	42	4A	112	72	162
23	43	4B	113	73	163
24	44	4C	114	74	164
25	45	4D	115	75	165
26	46	4E	116	76	166
27	47	4F	117	77	167

Hexadecimal	Octal	Hexadecimal	Octal	Hexadecimal	Octal
78	170	AA	252	DC	334
79	171	AB	253	DD	335
7A	172	AC	254	DE	336
7B	173	AD	255	DF	337
7C	174	AE	256	E0	340
7D	175	AF	257	E1	341
7E	176	B0	260	E2	342
7F	177	B1	261	E3	343
80	200	B2	262	E4	344
81	201	B3	263	E5	345
82	202	B4	264	E6	346
83	203	B5	265	E7	347
84	204	B6	266	E8	350
85	205	B7	267	E9	351
86	206	B8	270	EA	352
87	207	B9	271	EB	353
88	210	BA	272	EC	354
89	211	BB	273	ED	355
8A	212	BC	274	EE	356
8B	213	BD	275	EF	357
8C	214	BE	276	F0	360
8D	215	BF	277	F1	361
8E	216	C0	300	F2	362
8F	217	C1	301	F3	363
90	220	C2	302	F4	364
91	221	C3	303	F5	365
92	222	C4	304	F6	366
93	223	C5	305	F7	367
94	224	C6	306	F8	370
95	225	C7	307	F9	371
96	226	C8	310	FA	372
97	227	C9	311	FB	373
98	230	CA	312	FC	374
99	231	CB	313	FD	375
9A	232	CC	314	FE	376
9B	233	CD	315	FF	377
9C	234	CE	316		
9D	235	CF	317		
9E	236	D0	320		
9F	237	D1	321		
A0	240	D2	322		
A1	241	D3	323		
A2	242	D4	324		
A3	243	D5	325		
A4	244	D6	326		
A5	245	D7	327		
A6	246	D8	330		
A7	247	D9	331		
A8	250	DA	332		
A9	251	DB	333		

RE-ENTERING A GRAPHICS TASK OVERLAY F

A graphics task overlay consists of a FORTRAN program and its associated subroutines in absolute format. Each task is entered by an unconditional jump to the entry address of the overlay, and normally no provision is made to return to the statement following the task call.

AERTRN

Under certain circumstances, the programmer may wish to return from a graphics task to the statement following the CALL AETSKC card which caused entry to the task; he might do this if he wanted to call several tasks in a row. A routine called AERTRN is provided for this purpose.

When the programmer wants to return from one task to another, he places a card with the format

```
CALL AERTRN
```

in the task he wishes to return from. When this card is encountered, control is passed to the return address of the last executed return jump to AETSKC.

Note that AERTRN does not provide for reloading the task that called AETSKC; it provides only the jump to pass control and the record of the last call to AETSKC. The programmer must insure that the tasks do not overload each other, and that the AERTRN call occurs whenever the return feature is desired.

EXAMPLES

The following examples show how secondary overlays and the C parameter on the overlay card may be used to set up a task file so that AERTRN can be used.

C PARAMETERS

The standard FORTRAN overlay card has the format

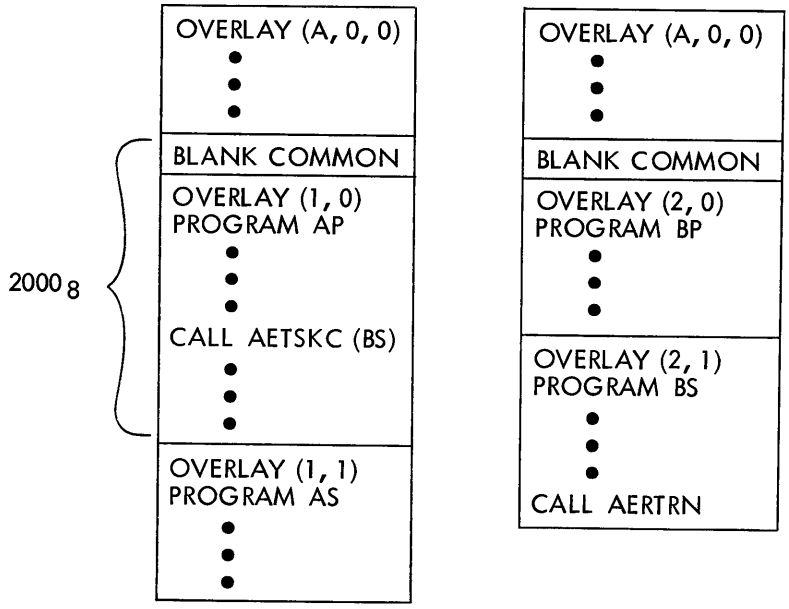
```
OVERLAY (lfn, p, s, Cnnnnnn)
```

where lfn, p, and s have the meanings given in the overlay card definition of Section 2. The quantity nnnnnn after the letter C in the parameter field is an octal value that specifies the first word address of the overlay with respect to the beginning of blank COMMON; i.e., the overlay coding is loaded and entered at a location nnnnnn words after the beginning of the program's blank COMMON area. This C parameter cannot be used on the zero-level overlay card, but is optional on all other overlay cards.

Since the first word address of blank COMMON is constant for any given overlay or task file, all overlays with the same C parameter will have the same first word address in core.

For example, assume that AP and BP are primary overlays, and AS and BS are secondary overlays; all four have been written into a task file by AEFILE.

These four overlays would appear in core as:



Program AS has been relocated with respect to the last word address plus one of the program AP because they have the same primary level number. Program BS has been relocated with respect to the last word address plus one of program BP because they also have the same primary level number.

OVERLOD, the standard SCOPE overlay loader, will not allow overlays (1, 0) or (1, 1) to call overlay (2, 1). Primary overlays and overlays with the same primary number may call each other; no other calls are allowed. However, a call to AETSKC allows any of the four overlays to call any of the others by using their program name.

If a programmer places a task call to BS from AP, part of the called task will be loaded over the calling task. If a call to AERTRN is then made at the end of BS, AERTRN will return control to the core address following the last AETSKC call, but the return will have chaotic results because the core locations that contained the code which the programmer wished to execute have been overlayed by the beginning of program BS. The following paragraphs describe one method of avoiding this problem.

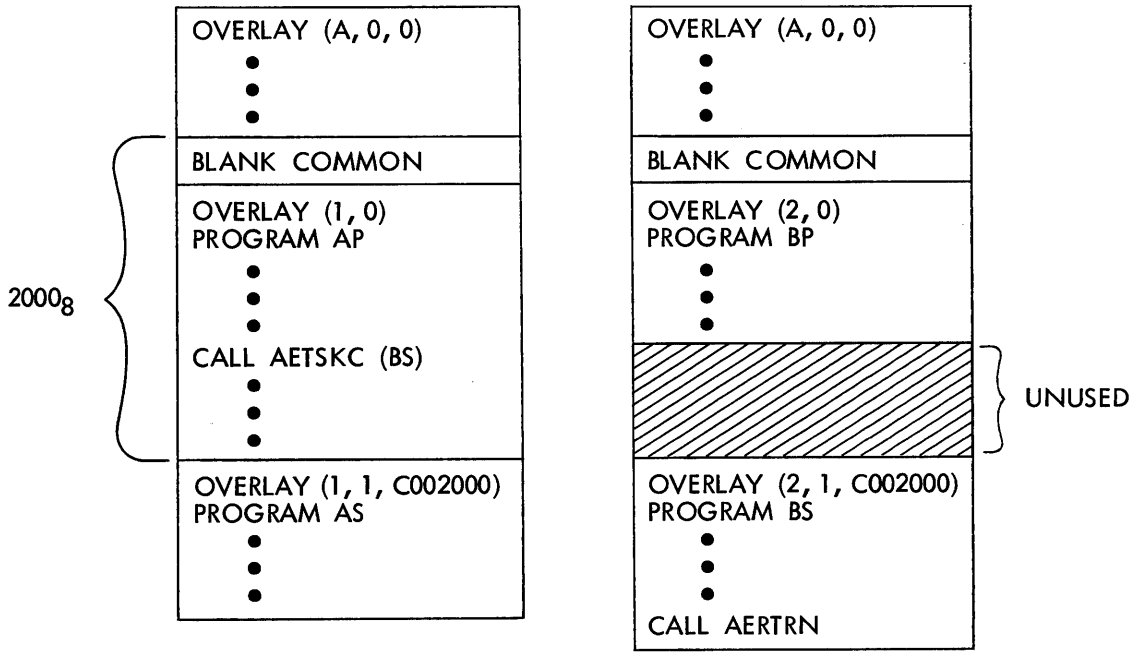
Assume that overlay (1, 1) is loaded 2000_8 words from the first word address of blank COMMON. If the secondary overlay cards are written with the C parameter so that they appear as

OVERLAY (1, 1, C002000)

and

OVERLAY (2, 1, C002000)

the routines in the overlays will have the same first word address and will appear in core as



Now each of the programs is free to call the others and to use a CALL AERTRN card to return to the address following the last call to AETSKC.

It is up to the programmer to keep track of the overlay core relationships when using AERTRN. A task which calls another with the expectation of returning should be located so that the two do not overlay each other. AERTRN provides limited capability for constructing tasks which may be called by AETSKR and also entered as subroutines. If logic requires the

use of AERTRN in some cases and AETSKC or AETSKR in others, a flag may be set in blank COMMON by the calling task and interrogated before each return is executed.

An error message, RETURN ADDRESS OVERLAYED, will be sent to the dayfile and a task return will be executed only if the return address of AERTRN is within the overlay calling AERTRN.

Note that linkage of external symbols is not provided for by the GPSL Loader between overlays with different primary level overlay numbers. If overlays 2.1 and 2.0 have subroutine linkages in common, the overlay 2.1 will probably not run correctly unless 2.0 is in core at the same time. AERTRN should primarily be used in secondary overlays with the same primary number while the primary is in core.

SYSTEM PACKING OF IBUF DESCRIPTION BUFFERS G

Nine graphics utilities routines of the 6000 Basic Graphics Package place item description bytes in IBUF; in addition, both GIDISP and GIMAC place header and trailer bytes into the description buffer before sending it to the 1700 buffer translator through EXPORT.

Table G-1 lists all of the routines that place bytes into IBUF and gives the number of bytes packed by each; all 12-bit bytes are packed five to a 60-bit central memory word, starting in byte zero. The first byte generated would be packed into bits 59 through 48. After the fifth byte is packed into bits 1 and 0 of the CM word, a new word is started.

TABLE G-1. IBUF/1744 BYTE
COMPARISON, ITEM DESCRIPTION BYTE GENERATORS

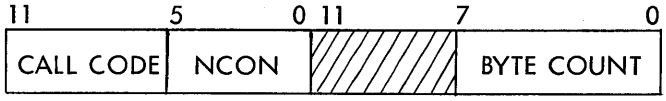
Routine	(Octal) Call Code	Number of Bytes Packed	Bytes Packed in 1744	Explanation
GUAN	02	$1 + \frac{NC + 1}{2}$	NC + 2	NC is the character number parameter in the call
GUARCG	06	6 + 4 * KSHOW	3/inch (large) 6/inch (small)	KSHOW is the arc segment number parameter in the call
GUBYTE	08	1 + L	–	L is the macro address number parameter in the call
GUMACG	07	1 + 2 * L	2	
GURSET	01	3	3	
GUSEG	04	3	4 + (2 or 3/inch)	
GUSEGA		4 + 3 * N (4 Bits)	–	Calls GUSEGI once, GUSEG N times, where N is the line segment number
GUSEGI	11	4	–	
GUSEGS	03	6	Same as GUSEG	
GIDISP	01	10	2 + 3 to 7, depending upon parameter	Eight trailer and two header bytes; explained below
GIMAC	05	4	3	Two header and two trailer bytes; explained below

Each graphics utilities call packs a call code for the corresponding 1700 Basic Graphics Package routine into the upper four bits of the first 12-bit byte it places in IBUF; if a graphics utilities routine is called with NBYTE equal to zero, the routine will leave two bytes empty at the beginning of the next unfilled central memory word in IBUF. The two empty bytes are usually used by GIDISP or GIMAC for the two header bytes which each packs in IBUF.

NOTE

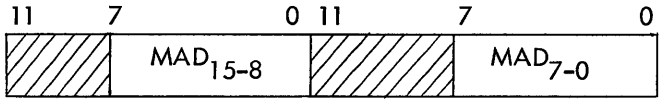
The call codes of all GU routines are in the upper 4 bits of a 12-bit byte and are within a GIDISP, GIMAC, or GIPLOT buffer. The call codes of all GI routines are in the upper 6 bits of a 12-bit byte at the beginning of a 60-bit central memory word.

These bytes have the structure



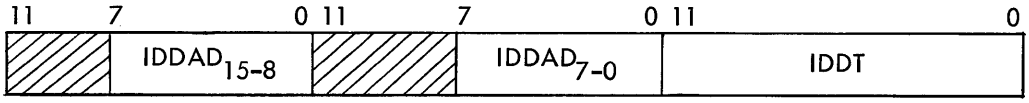
where the byte count excludes the header and trailer bytes.

The two trailer bytes packed by GIMAC are placed in IBUF immediately after the last item description byte. The first of these two bytes contains bits 15 through 8 of the lower 16 bits of MAD, right-justified; the second byte contains bits 7 through 0 of MAD, also right-justified, as shown:

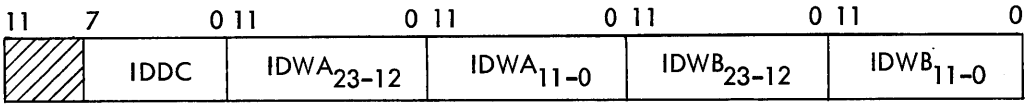


GIDISP places a variable number of trailer bytes (three to eight) in IBUF, immediately following the last packed item description byte. The number packed depends on the number of parameters present in the GIDISP calling sequence before a minus zero parameter or a right parenthesis is encountered.

The three trailer bytes always packed by GIDISP contain bits 15 through 8 of IDDAD right-justified in the first, bits 7 through 0 of IDDAD right-justified in the second, and IDDT in the lower 8 bits of the third, as follows:



In a full calling sequence, five more bytes would be packed. These would contain: IDDC in the lower 8 bits of the first additional byte, bits 23 through 12 of IDWA in the second, bits 11 through 0 of IDWA in the third, bits 23 through 12 of IDWB in the fourth, and bits 11 through 0 of IDWB in the fifth. These bytes appear as follows:



If IDDT and IDDC are not defined, a minus zero value is stored in their respective bytes. GIDISP terminates packing with the first minus zero parameter. If a right parenthesis terminates the calling sequence before IDWB, the first missing parameter is replaced by a minus zero and packing is terminated.

Both GIMAC and GIDISP issue a fatal error diagnostic if the IBUF description buffer resulting from the call is longer than 64 central memory words (including header and trailer bytes).

Both routines process a non-fatal error and issue an informative dayfile message if the NBYTE parameter is equal to zero when the routine is called.

OMISSION OF AEXEC FROM PROGRAM CODING H

Although the procedure is not recommended, the application executive AEXEC program can be omitted from a graphics program. If the programmer does not use AEXEC, he must either provide substitutes for each of its routines or else resign himself to the use of an inordinately large amount of central memory by his job. The large amount of central memory is necessary because AEXEC must reside in the zero-zero overlay since it is designed to be called only once. There is no return from AEXEC. It could not be loaded as needed since it opens the random task file, calls Scheduler, loads the one-zone overlay and transfers control to it. This appendix provides an outline of the structure and functions of AEXEC so that a programmer can write replacement routines if he wishes.

STRUCTURE OF AEXEC

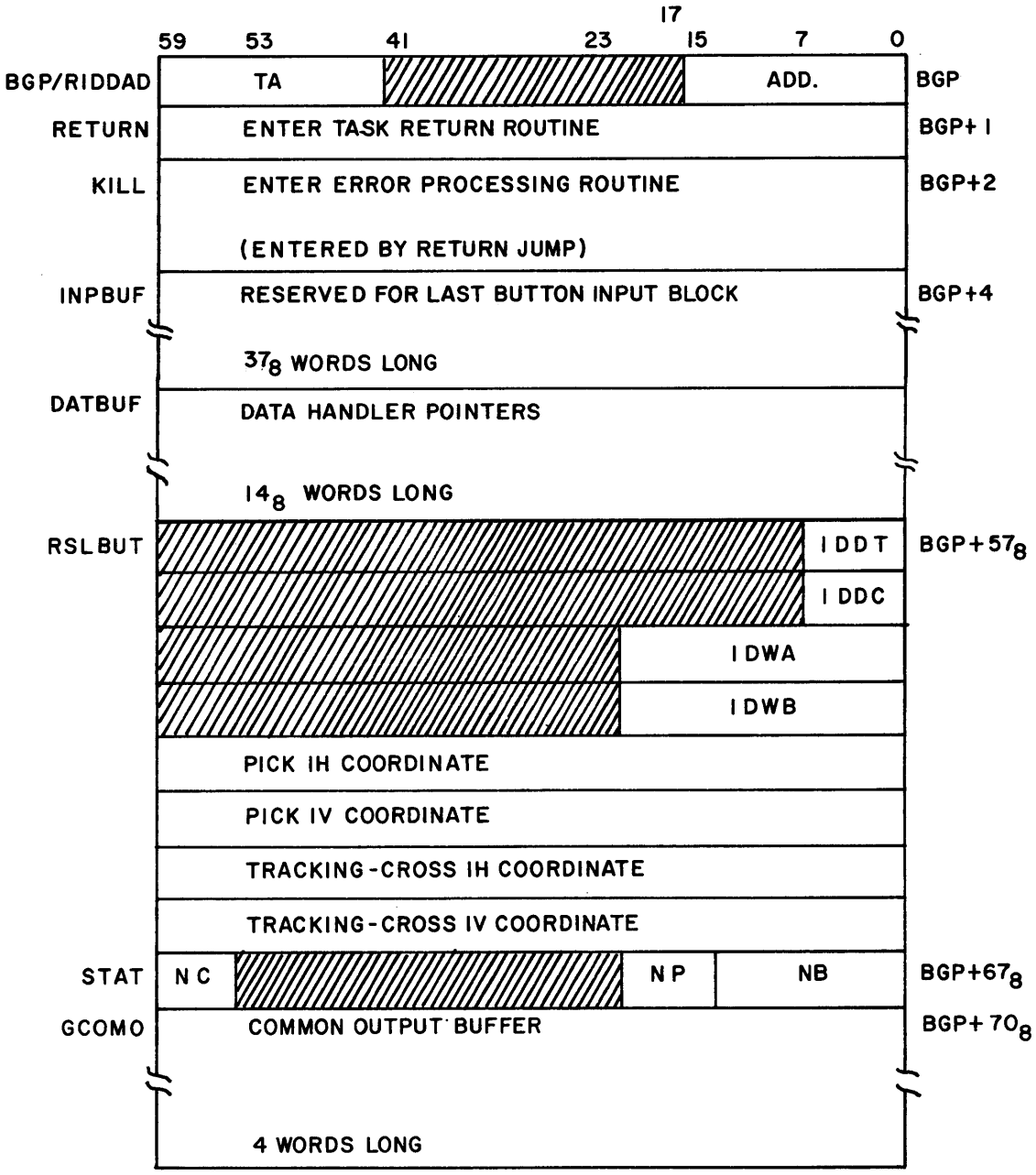
The first 74_8 words in AEXEC are entry points and buffer areas shared by the 6000 Basic Graphics Package routines (see Figure H-1).

AEXEC has seven formal entry point names:

AERTRN
AETSKC
AETSKR
BGP
DATBUF
AEXEC
RIDDAD

The entry point AEXEC is called only once during a job and is not referenced by any of the routines in the 6000 Basic Graphics Package. AEXEC is the entry point used by the CALL AEXEC card in the program's zero-level overlay and provides access to coding that provides AEFILE with the file names it uses during a task file creation run; the coding associated with entry point AEXEC also initiates loading and execution of the first overlay in the task file during an execution run.

The entry points AETSKR, AERTRN, and AETSKC are used by the respective 6000 Basic Graphics Package routines. These entry points provide unconditional jumps to appropriate subroutines within the AEXEC program.



TA=ADDRESS OF DISPLAY CODE NAME OF CURRENT TASK IN MEMORY
ADD=RIDDAD/MAD COUNTER
NC =LAST NCON ARGUMENT OF GINCJB OR GIBUT
NB =GIFSID POINTER (NUMBER OF BYTES RETURNED)
NP =GIFSID POINTER (NUMBER OF PICKS RETURNED)

Figure H-1. AEXEC Communications Area

The entry points BGP, DATBUF, and RIDDAD are used by the 6000 Basic Graphics Package routines to share access to common pointers.

The last four entry points described above are all linked to the presence of AEXEC in the program's zero-level overlay. If AEXEC is not called there and the application program contains no subroutines with entry points named BGP, RIDDAD, and DATBUF, then the entire AEXEC program will be loaded from the system library into every overlay that contains Package routines which reference any of these entry points.

If the application program does contain a subroutine (in place of AEXEC) with the proper entry points, all 6000 Package routine references to the points are linked to the locations listed in Table H-1.

TABLE H-1. 6000 PACKAGE EXTERNAL LINKAGES

Entry Point Name	Purpose	Type of Reference
FORTRAN-Callable:		
AETSKC	Load an overlay by name	Relocatable
AERTRN	Return from task to calling address plus one	Relocatable
AEXEC	Initialize Application Executive	Relocatable
Not FORTRAN-Callable:		
BGP	Unused to reference first 74_8 words of AEXEC relatively	Relocatable
RETURN	Perform task return (AETSKR)	Relative to BGP
KILL	Process errors and messages	Relative to BGP
INPBUF	Reserve last button input block	Relative to BGP
RIDDAD	Reserve task name address and IDDAD/MAD counter	Relative to BGP
RSLBUT	Reserve last button ID parameters	Relative to BGP
STAT	Reserve NCON and single/string pick counters	Relative to BGP
GCOMO	Common EXPORT output buffer	Relative to BGP

CREATING ALPHANUMERIC DISPLAY FONTS I

For certain applications, the programmer may wish to provide the console user with a display font other than the two supplied in the 6000 Basic Graphics Package (see Section 6, GFONTA and GFONTN). The following discussion covers some of the more important points that a programmer should consider when creating his own display font.

FONT CHARACTER RECOGNITION

The 1700 Basic Graphics Package recognizes a sequence of display generation bytes followed by a one-word ID as a display font character. When the character is picked with the lightpen and GIANS has been called, the ID word is queued on an alphanumeric string so that it can be sent to the application program when a GIANE call occurs. Each 8-bit ID word in the 1700 is an ASCII character and is converted to 6000 display code before being sent to the 6000 application program.

Because of this processing, the application programmer can create font characters by supplying the one-word ASCII ID through a call to GUBYTE. For example, the three calls:

```
CALL GURSET(IH, IV, ICODE, IBUF, NBYTE, MBYTE)
CALL GUAN(1LA, 1, IBUF, NBYTE, MBYTE)
CALL GUBYTE(101B, 1, IBUF, NBYTE, MBYTE)
```

create an alphanumeric font of one character, A, at screen coordinates IH and IV. The ASCII code for A is 101_8 or 41_{16}.

The call

```
CALL GIDISP(NCON, IBUF, NBYTE, IDDAD, -0)
```

then displays this one character font. After the font appears on the screen, the call

```
CALL GIANS(NCON, 10, IH1, IV1)
```

creates a light-register at screen coordinates IH1 and IV1; this register can contain as many A's as are picked up to a limit of 10.

If the character A is picked once and GIANE is called, the parameters returned to the call will be

NC = 1

IBCD = Abbbbbbbbb

where the letter b indicates a blank.

SPECIAL CHARACTERS

Two special characters are defined for the 1700 Package. These two characters, backspace and clear, allow the console operator to remove characters which have been queued since the call to GIANS and before the next call to GIANE occurs.

BACKSPACE

Any display followed by a one-word ID of 137B (or $5F_{16}$) is defined as a backspace character. When such a character is picked with the lightpen, the last picked character in the light-register is erased from the display and the underline is restored; the ID of the erased character is also removed from the buffer of queued alphanumeric information.

CLEAR

Any display followed by a one-word ID of 177B (or $7F_{16}$) is defined as a clear character. When such a character is picked with the lightpen, all of the characters currently in the light-register are erased and the entire underline is restored; in addition, the ID's for all of the erased characters are removed from the buffer of queued alphanumeric information.

Backspace and clear have no other effect on alphanumeric picking.

RESET SEQUENCE

When a GURSET call is used in the definition of a font character, the ICODE s bit (bit 2^6) must be set. The s bit of the reset sequence is the enable lightpen bit; if it is not set, the character's ID word is not read when a pick is made, and the character consequently cannot be entered into the light-register or queued for processing by the 6000 series computer.

A font character can be generated without a reset sequence by using a GUAN call with NC set equal to one, but a no-operation instruction must precede the GUAN call in the character's IBUF. This no-op may be supplied by a GUBYTE call of one byte, where the byte is a positive zero value.

CONSERVING ID WORD SPACE

The ID words IDDT, IDDC, IDWA, and IDWB of the GIDISP call or calls which display font characters need not be referenced; the 1744 buffer space they normally occupy can be conserved by truncating the parameter list with a closing or right parenthesis after IDDAD.

DYNAMIC ADDITION OF CHARACTERS

Characters may be added to an existing console display font by successive calls to GIDISP at any time; duplicates of the same character, i.e., characters with the same ASCII code ID words, may be present in a font.

SAMPLE FONT CREATION ROUTINES

The following subroutine creates a display font containing:

0 1 2 3 4 5 6 7 8 9 X

```
      SUBROUTINE NFONT (NCON, IBUF, NBYTE, MBYTE, IDDAD)
      DIMENSION IBCD (10)
      DATA (IBCD(I), I=1,10)/1L0,1L1,1L2,1L3,1L4,1L5,1L6,1L7,1L8,1L9/
      CALL GURSET (0,-600,103B,IBUF,NBYTE,310)
      ICON1 = 60B
      ICON2 = 71B
      DO 1 I = ICON1,ICON2
      J = I -57B
      CALL GUAN (IBCD(J), 1, IBUF, NBYTE, MBYTE)
C     THE PRECEDING CALL GENERATES ONE OF THE FONT CHARACTERS
      CALL GUBYTE (I, 1, IBUF, NBYTE, MBYTE)
C     THE FOLLOWING CALL PROVIDES SPACING BETWEEN CHARACTERS AND
C     COULD BE REPLACED BY A GURSET CALL
     1CALL GUAN (1L , 1, IBUF, NBYTE, MBYTE)
      CALL GUAN (2L , 2, IBUF, NBYTE, MBYTE)
      CALL GUBYTE (0, 1, IBUF, NBYTE, MBYTE)
      CALL GUAN (1LX, 1,IBUF, NBYTE, MBYTE)
      CALL GUBYTE (130B, 1, IBUF, NBYTE, MBYTE)
C     THE THREE PRECEDING CALLS CREATE AND IDENTIFY THE CHARACTER X
C     AS AN END-OF-MESSAGE CHARACTER FOR USE IN GIEOM ASSIGNMENT
C     THE FOLLOWING CALL DISPLAYS THE FONT
      CALL GIDISP (NCON, IBUF, NBYTE, IDDAD, -0)
      RETURN
      END
```

If the programmer wishes to generate each character with a separate GUAN call, he must provide a no-op instruction before each GUAN call. CALL GUBYTE(0,1,IBUF,NBYTE, MBYTE) provides the appropriate no-op.

The programmer can also create display font characters of any size he wishes; he need not use the size characters that are defined by the 1700 Basic Graphics Package alphanumeric

macros. For example, the three following calls create a circle with a center at IHC and IVC, and an initial/termination point at IH and IV. This circle is queued as an alphanumeric O when picked with the lightpen.

```
CALL GURSET (IH, IV, ICODE, IBUF, NBYTE, MBYTE)
CALL GUARCG (1, IHC, IVC, IH, IV, IH, IV, IBUF, NBYTE, MBYTE)
CALL GUBYTE (117B, 1, IBUF, NBYTE, MBYTE)
```

Note that the ASCII code equivalent of O is 117B ($4F_{16}$).

The programmer can create a true/false font with coding like the following:

```
      CALL GURSET (IH1, IV1, ICODE, IBUF, NBYTE, MBYTE)
      CALL GUAN (4HTRUE, 4, IBUF, NBYTE, MBYTE)
      CALL GUBYTE (124B, 1, IBUF, NBYTE, MBYTE)
C     THE PRECEDING CALLS CREATE THE WORD TRUE BEGINNING AT IH1/IV1
C     AND QUEUE AN ALPHANUMERIC T(=124B) WHEN IT IS PICKED
      CALL GURSET (IH2, IV2, ICODE, IBUF, NBYTE, MBYTE)
      CALL GUAN (5HFALSE, 5, IBUF, NBYTE, MBYTE)
      CALL GUBYTE (106B, 1, IBUF, NBYTE, MBYTE)
C     THE PRECEDING 3 CALLS CREATE THE WORD FALSE BEGINNING AT IH2/IV2
C     AND QUEUE AN ALPHANUMERIC F (=106B) WHEN IT IS PICKED
```

INDEX

COMMENT SHEET

MANUAL TITLE 6000 Series Interactive Graphics System Version 2 Reference Manual

PUBLICATION NO. 17303600 **REVISION** D

FROM: NAME: _____

BUSINESS
ADDRESS: _____

COMMENTS:

This form is not intended for use as an order blank. Your evaluation of this manual is welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be noted below. Please include page number references and fill in the publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer Engineers are urged to use the TAR.

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Systems Publications
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

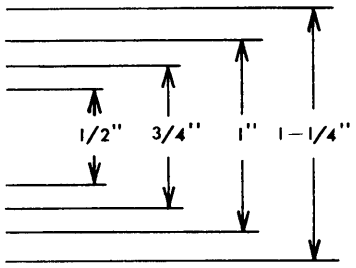
FOLD

FOLD

STAPLE

STAPLE

CONTROL DATA



→→ CUT OUT FOR USE AS LOOSE-LEAF BINDER TITLE TAB

274 INTERACTIVE GRAPHICS SYSTEM REFERENCE MANUAL

CONTROL DATA
CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

PRINTED IN U.S.A.

Pub.No. 17303600