A GENERAL PURPOSE GRAPHICS SYSTEM FOR A SMALL COMPUTER

Timothy O'Brien McNeil, B.Eng.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Engineering.

Department of Electrical Engineering,
McGill University,
Montreal, Québec.

March 1973.

ABSTRACT

This thesis presents an implementation of Bell Telephone Laboratories' BELLGRAPH software system on McGill's disc refreshed computer graphics display. By modifying and installing an existing software package, it was possible to take advantage of forty man-years of effort and experience in the construction of a sophisticated, interactive graphics system complete with its own high level graphical programming language. The special considerations and problems encountered in transforming a large operating system designed to drive a core refreshed display into one capable displaying on disc refreshed hardware are described. Certain features of a disc refreshed graphics not attainable in systems refreshing from main memory, such as improved background processing are also outlined.

ABSTRACT

Cette these préscute une realization sur le systeme graphique regeneration par disque à l'Université McGill, du système de programmation BELLGRAPH qu'ont developpé les laboratoires du Bell Telephone. En modifiant et en installant un systeme developpé il a été possible de profiter de l'investissement d'une quarantaire d'homme-années d'effort et d'expérience dans la construction d'un système graphique interactif avancé qui comporte un language evolvé de programmation graphique. Les considérations et les problèmes rencoutrés en transformant ce vaste système de programmation conçu pour un système graphique regeneration par la memoire centrale pour un système regeneration par disque sont présentés. Certains traits, qui ne sont possibles que sur un système regeneration par disque, tel que le traitement amelioré de programmes non prioritaires, sont esquissés.

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to the many people who contributed to this project.

The guidance, assistance, and patience of my thesis advisor, Dr. M.D. Levine is much appreciated.

Thanks are due to Dr. A. Malowany who stimulated many ideas during our innumerable discussions.

I would like to thank Larry Rosler, Marty Welt, and Elain Yamin at Bell Telephone Laboratories for providing the BELLGRAPH software and assistance in its implementation.

Thanks are due to Juhan Lemitt, who wrote all the PDP-8 programs, and maintained the graphics hardware.

I am indebted to Chris Thompson at the Montreal Neurological Institute who provided machine time on their busy PDP-12.

The many discussions on many diverse topics with Doug Skuce and Ron Poulsen helped the author maintain his sense of humour throughout it all.

I especially thank my future bride, Louise, for her encouragement and understanding. She also did an excellent job in typing this thesis.

The research was supported by the National Research Council.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

The lack of effective communication between man and computer has limited the more intimate use of machines in almost all human endevours. To be truly effective people should be able to deal directly with the machine doing their banking, reserving their airline seat or calculating the stress in their mechanical design.

Major efforts have been made towards making computers user oriented. In the last decade high level programming languages such as ALGOL and PL/1 have been developed with an "English-like" syntax and powerful diagnostic support. Timesharing systems provide each user his own terminal through which he can communicate using a conversational language such as APL.

A current area of interest is getting the computer to comprehend data in a form more natural to humans than to machines. Optical readers can now be obtained which input data directly from a typewritten page instead of going through punched cards, magnetic tape, or a keyboard. Systems which recognize hand writing are also being built experimentally. The computer comprehension of pictorial data has produced some of the most rewarding results. Using cathode ray tubes (CRT) or television as output devices, computer graphics systems have drawn pictures ranging from two dimensional graphs of mathematical results to three dimensional colour scenes moving in real-time. For example, animated cartoons have been generated by machine through the use of computer graphics.

Another highly exploited use of computer graphics is the broad field of computer-aided design. An engineer using a light pen sketches a machine part on the CRT screen, checks its interaction with adjoining pieces and computes the stress within it, altering his original idea as he goes until a finished design results. The computer then prepares the paper tape used to control an automated machine tool used to cut the part. Similar design systems exist for the automated manufacturing of integrated circuits.

A third example of the use of computer graphics is pattern recognition. High energy physicists spend long hours studying bubble chamber photographs in an attempt to discover sub-atomic events. Physicians pore over X-ray photographs or microscopes to achieve a diagnosis. Weather forecasters use pictures sent by sattelite in arriving at their predictions. A machine which would compress the data stored in a picture into a few descriptive statements would be a powerful tool in many areas.

This thesis is concerned with the development of the general purpose graphics system at McGill, capable of being used for both computer-aided design and pattern recognition problems. In Chapter II, a brief review is given of existing systems in an attempt to familiarize the reader with this field, and provide a basis for the design considerations of McGRAPH. The remainder of the thesis then describes a general purpose graphics system for picture synthesis using a small computer. Throughout the thesis the special problems introduced by picture analysis are also mentioned and discussed.

CHAPTER II

EXISTING GRAPHICS SYSTEMS

2.1 The Sketchpad System

"Sketchpad",[1] a graphics system built by I.E. Sutherland at MIT in 1963, dramatically demonstrated the computer's ability to generate and display complicated drawings. Using toggle switches, pushbuttons, knobs and a light pen one could draw figures on a CRT screen and then rotate, magnify and move them around. For example, by depressing the "DRAW" pushbutton the computer traces out a straight line stretching like a rubber band from some initial point to the current position of the light pen. Circle arcs are constructed in a similar manner by depressing the "CIRCLE CENTRE" pushbutton to define its position and then choosing a point on the arc and pressing "DRAW" to define its radius. The length of the circle arc is controlled by the light pen position (see Figure 2.1). To illustrate Sketchpad's operation, let us construct an equilateral triangle by inscribing it in a circle. First we draw a circle and any triangle using the techniques described above. By pointing to a corner and depressing the "MOVE" pushbutton we can drag the corner onto the circle. After all three corners have been constrained to the circle, each side can be made equal by turning on the "EQUAL" toggle switch and pointing to each line. The circle can be deleted via the pushbutton "DELETE". If we wished to make a pattern using this triangle, we could copy it with the

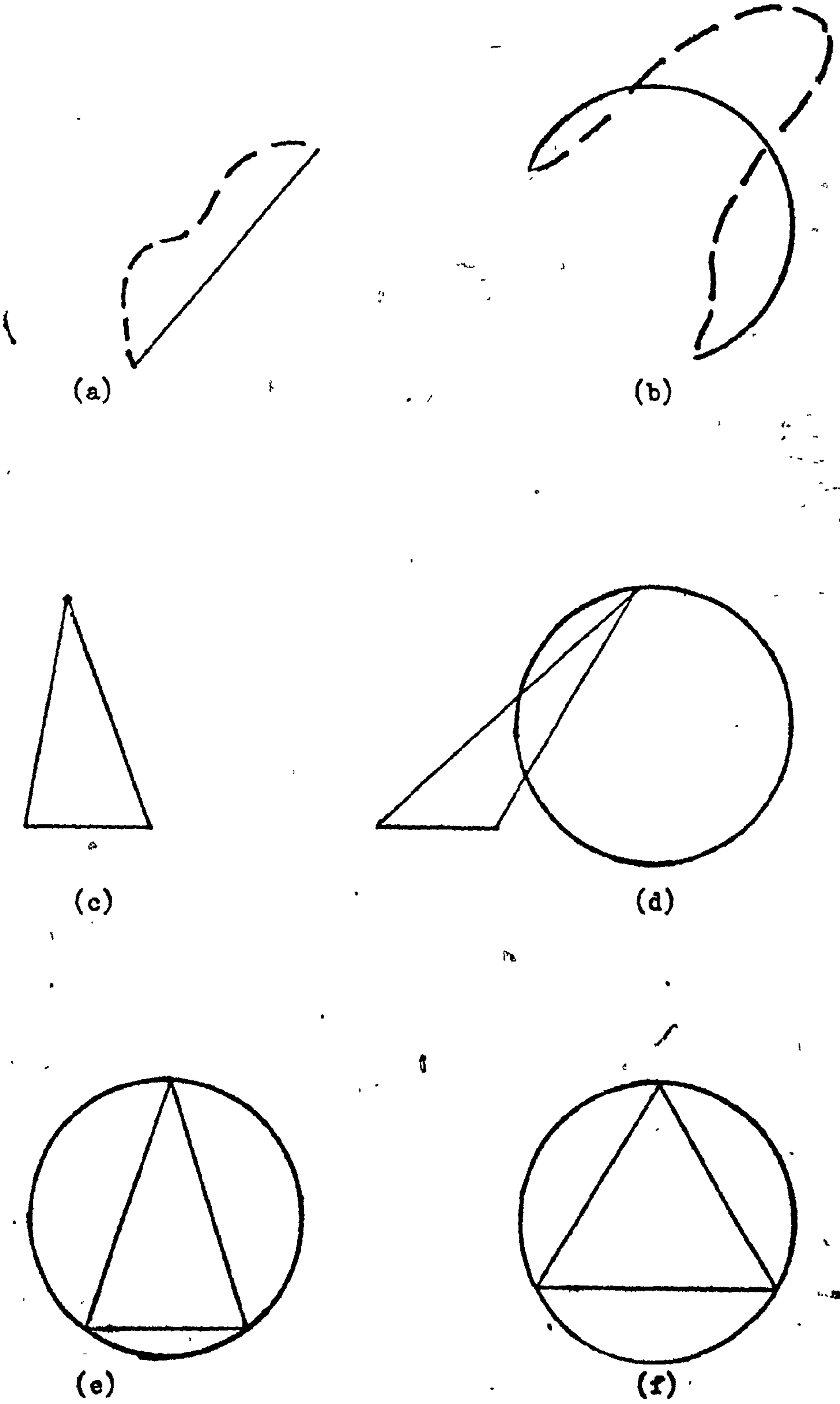
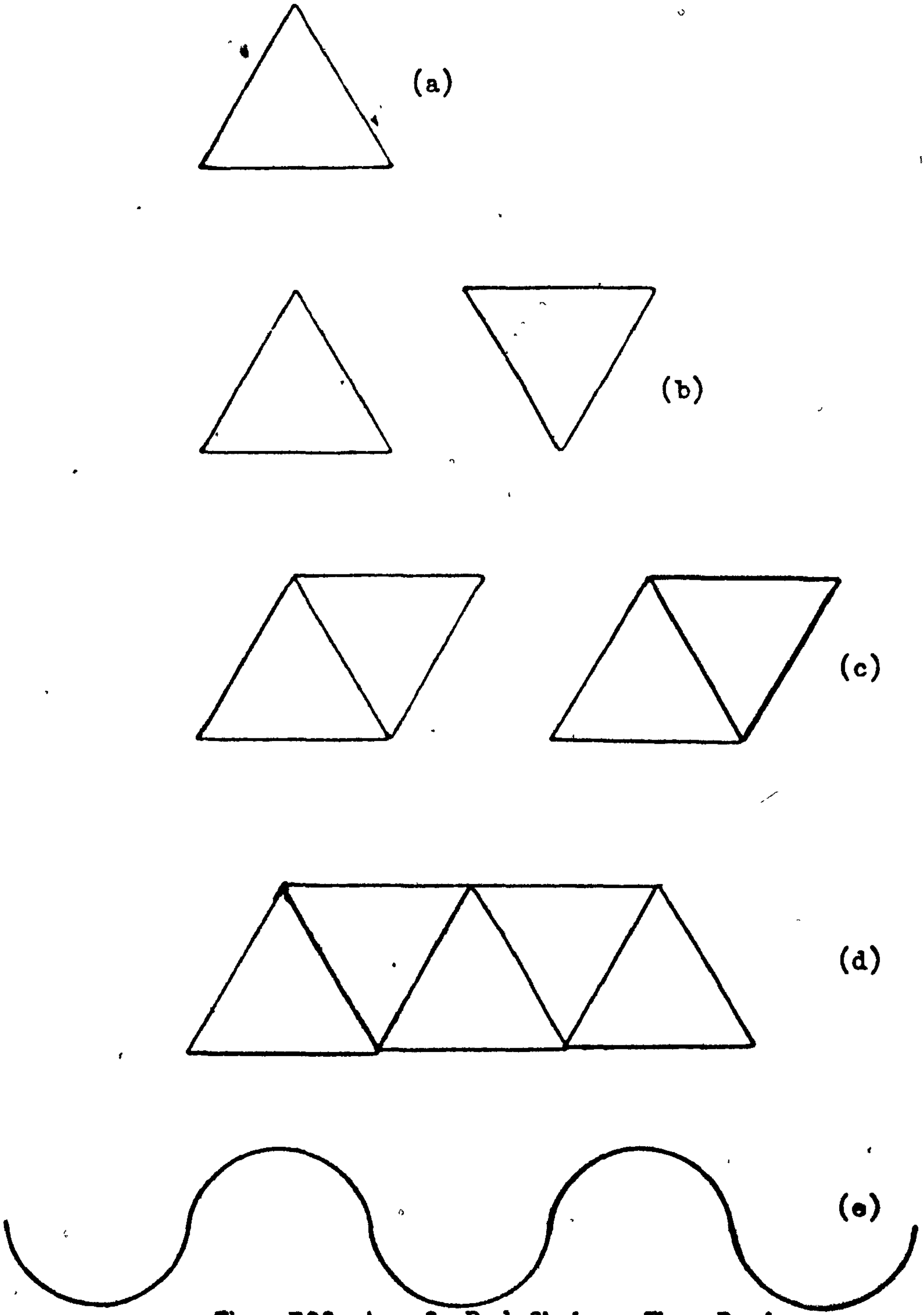


FIGURE 2.1. DRAWING SEQUENCE TO CONSTRUCT EQUILATERAL TRIANGLE USING SKETCHPAD.

"COPY" pushbutton, rotate the copy using one of the knobs and attach it to the first. We could then copy this composite drawing and attach it to its twin (see Figures 2.2 a - d) and so on, building up a complicated pattern from an initial simple object. To change the pattern to that in Figure 2.2 e, we call back the original triangle and redefine it as a semicircle.

From this simple example we can see a number of concepts which have become standard in subsequent systems. For instance, the light pen to indicate position and move objects. A number of other positional devices also have appeared such as the RAND tablet, joystick and tracking mouse each having their own strengths and weaknesses but effectively performing the same function. Pushbuttons are still a popular input device for indicating choice although light pen sensitive "light buttons" have been universally accepted. A light button is a short piece of text (e.g., MOVE, DRAW, ERASE) displayed on the screen along with the picture which when hit by the light pen interrupts the computer and indicates the operator's choice. They have an obvious advantage over pushbuttons since they require no extra hardware and are more easily interpreted by the user as the text displayed describes the function attached to it. Also, they provide greater flexibility since light buttons can appear and disappear depending upon the choices available to the operator. Another con-



The Effect of Redefining The Basic Primitive From a Triangle to Semicircle.

FIGURE 2.2. DRAWING SEQUENCE TO GENERATE PATTERN USING SKETCHPAD.

cept generally found in interactive graphics systems is the description of pictures in terms of its component parts. A facility is usually available for selectively moving, copying, rotating or erasing any picture or subpicture. This hierarchical description of drawings has profound impact on the manner in which data is stored in the computer. We shall elaborate on this topic in the section on software.

Sketchpad evoked basic relationships between picture parts through the use of toggle switches. In our example, the sides of the triangle were made equal by turning on the "EQUAL" switch. Other constraints such as PARALLEL, VERTICAL, HORIZONTAL were also available. Additional constraints could be added by writing assembly language programs and including them in the system. Sutherland[1] gives an example of tracing out the movement of three connected bars under the conditions of two joints fixed and a driving force on the third. This ability to define interrelationships (besides topological) and to simulate some action based on a given condition, is an important feature. In many instances, the user defines a model (e.g. circuit diagram, molecular configuration) using the graphics console and studies its response to given inputs. The picture drawn is merely a convenient representation of the operator's problem in a notation familiar to him. It is the computer's responsibility to translate this into a form which it can use in performing the required computations. This capability, essential to all interactive graphics systems is also discussed in the section on software.

To appreciate how a graphics terminal functions, however, we must first look at the hardware available, then investigate the programs used to draw pictures and input operator attention signals.

2.2 Graphics Hardware

2.2.1 System Configuration

Graphics terminals used for computer-aided design consist of a CRT and some positional input device (light pen, joystick etc.) connected to a small local computer which in turn is interfaced to a large remote time-sharing system. In the early systems, such as GRAPHIC-1[6] and DAC-1,[7] all real-time responses plus picture analysis computation were handled by the remote time-shared machine, and the local processor provided merely a terminal interface. The slow response time of these systems was a source of user annoyance and greatly limited their usefulness. The DAC-1 system attempted to alleviate user impatience by special messages, and audible bells reassuring the operator that the system was working on his problem and would return shortly. With the advent of powerful mini-computers, the computing load shifted away from the large central processor to the local computer. By handling the real-time interactions locally and transmitting only data requiring computation, one can significantly improve the response time and cut communication costs. In this configuration the central computer, freed of all graphical bookkeeping becomes basically a large bulk storage and computation unit.

2.2.2 Types of Display Hardware

There are three major classes of displays currently being used in graphics terminals: calligraphic, TV raster and storage tube. A brief comparison of these displays is presented here.

(i) Calligraphic Display

Modern calligraphic systems have a display processing unit (DPU) connected between the local computer and CRT (see Figure 2.3). This processor accepts display commands from a direct memory access channel (DMA), and translates them into analog voltages which position and intensify the beam. Hence, the picture is "painted" on the screen in a sequence specified by a command list in core. The power of these DPU's varies widely from

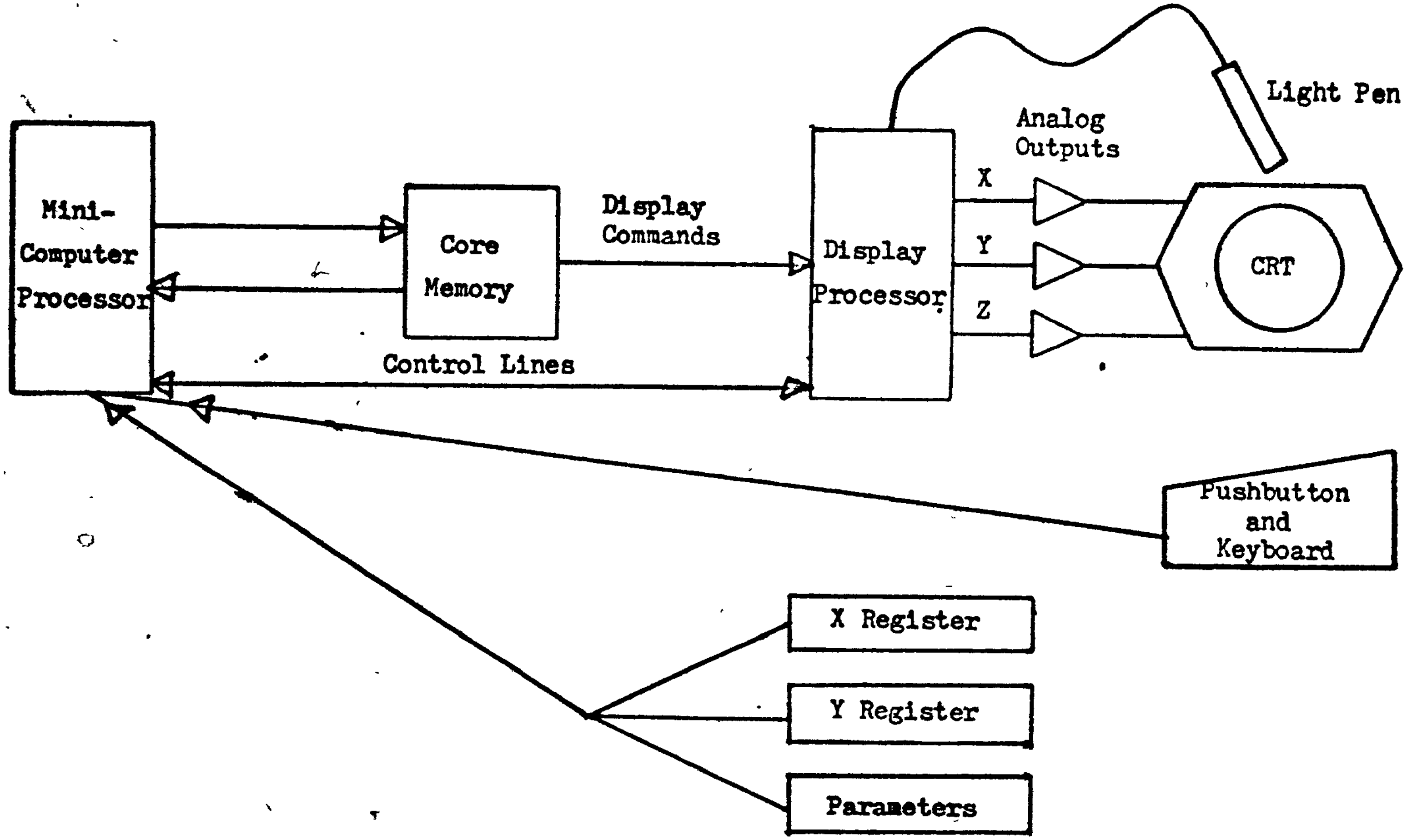


FIGURE 2.3. CALLIGRAPHIC DISPLAYS.

simple beam movements to providing selective blinking, dashed lines, windowing or three dimensional projections. Some DPU's also have an array of registers which can be read by the computer. An example of such a set might be:

1. Display status register (blink on or off, intensity, scale etc.).

2. X and Y co-ordinates.

3. Core address of current command.

Sophisticated DPU's have a display subroutine command so that different instances of the same picture part need not be repeated in core. This is a costly option, however, since to be truly effective the DPU must stack display parameters (position, intensity scale, orientation, etc.), as well as return addresses. When hardware costs decrease sufficiently to make subroutining feasible it will have major impact on the design of graphics software especially its data structures. Other useful additions are "windowing" circuits preventing lines which go off screen from wrapping around and appearing on the other side, and 3-D projection generators to translate X, Y, Z beam movement commands into perspective views of a 3-D object.

Calligraphic displays normally use the light pen as its positional input device. A program can immediately retrieve position and picture part identification from the DPU on a light pen interrupt since it occurs at the precise instant the display is drawing the part hit. Other positional input devices provide only X, Y co-ordinates which must be mapped into part identification.

The disadvantage of calligraphic displays is their cost. Since they are continuously refreshed, a segment of high speed memory (minimum 4K) must be dedicated to the display. Flicker also can become objectionable if the drawing becomes too complex. The DPU is another high cost item being a special complicated piece of hardware.

(ii) T.V. Raster

A second popular display uses a T.V. monitor with a standard broadcast raster scan (see Figure 2.4). This is a less expensive system since most of the hardware driving the screen, such as deflection yokes and amplifiers etc., is mass produced. A pleasant by-product of T.V. raster graphics is its ability to generate solid objects instead of line drawings. I.E. Sutherland's group at the University of Utah has produced some beautiful computer generated 3-D pictures of solid objects in colour.[3]

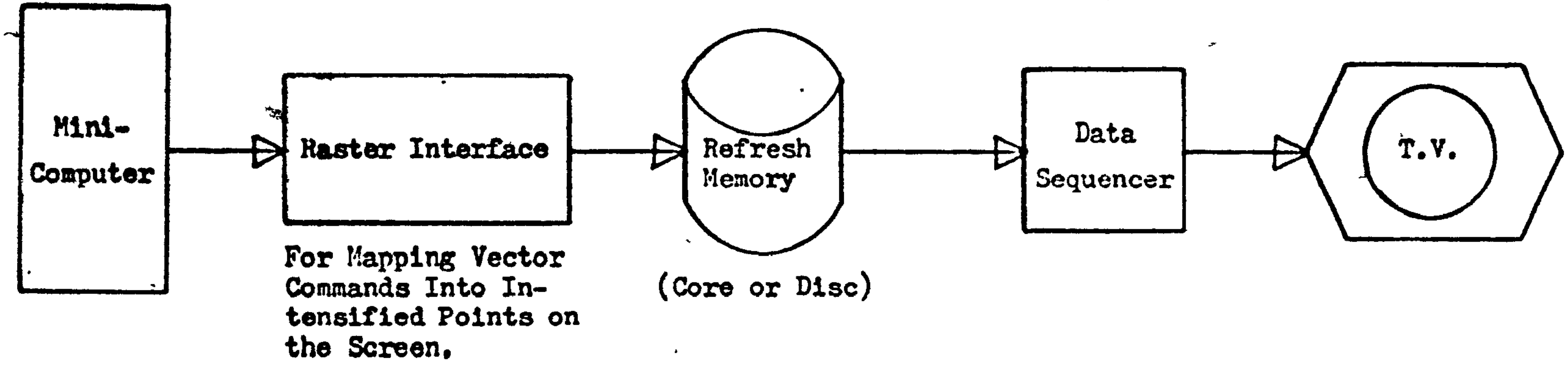


FIGURE 2.4. T.V. RASTER DISPLAYS.

The screen of the T.V. monitor is comprised of an array of points (normally about 480 x 480) and is refreshed by a continuous sweep of a memory, which contains the intensity (and perhaps colour) of each point.

The major disadvantage of T.V. raster graphics is the large amount of storage required for the array, and the problem of mapping lines and curves into it. Time taken to put up a new picture is measured in minutes due to this transformation problem if done by software. Micro-programmable hardware has been built to perform this mapping in a matter of seconds[8] but of course, this decreases the cost advantage of T.V. Also, even at this rate animation such as moving a tracking cross or picture is impossible in real-time. Another difficulty with T.V. is the information available from input devices. The X-Y co-ordinate positional devices such as joysticks, RAND tablets, etc. are favoured here since a light pen will not give position directly due to the interleaved scan. To determine which picture part is selected, a search of the data structure is required.

(iii) Storage Scopes

Storage scopes are capable of drawing very complicated scenes without flicker, and require no memory of their own. Their disadvantages are their slow drawing speed and inability to provide selective erasure or animation. In interactive systems using storage tubes as graphics terminals, a special processor called a "smart terminal" is connected between the display and large remote computer (see Figure 2.5). This processor handles the operator's real-time requests by drawing a cross (X) through an object to be deleted and adding a new instance to simulate movement of an object. In this mode, the screen soon becomes cluttered with antifacts of past revisions forcing the user to request a cleanup. The "smart terminal" then sends an update list to the central machine and waits for a fresh display. In the meantime the operator can go for a coffee.

2.2.3 Hardware Summary

The choice of display depends very much on the application. Where fast response, fine resolution, and animation are required, calligraphic displays are used. In applications where costs override these human factor niceties and resolution capabilities, T.V. is the next choice. T.V.'s also provide the best means of displaying solid objects. Other systems require a high resolution output device and do not need graphic interaction. In this case a storage screen provides the best service.

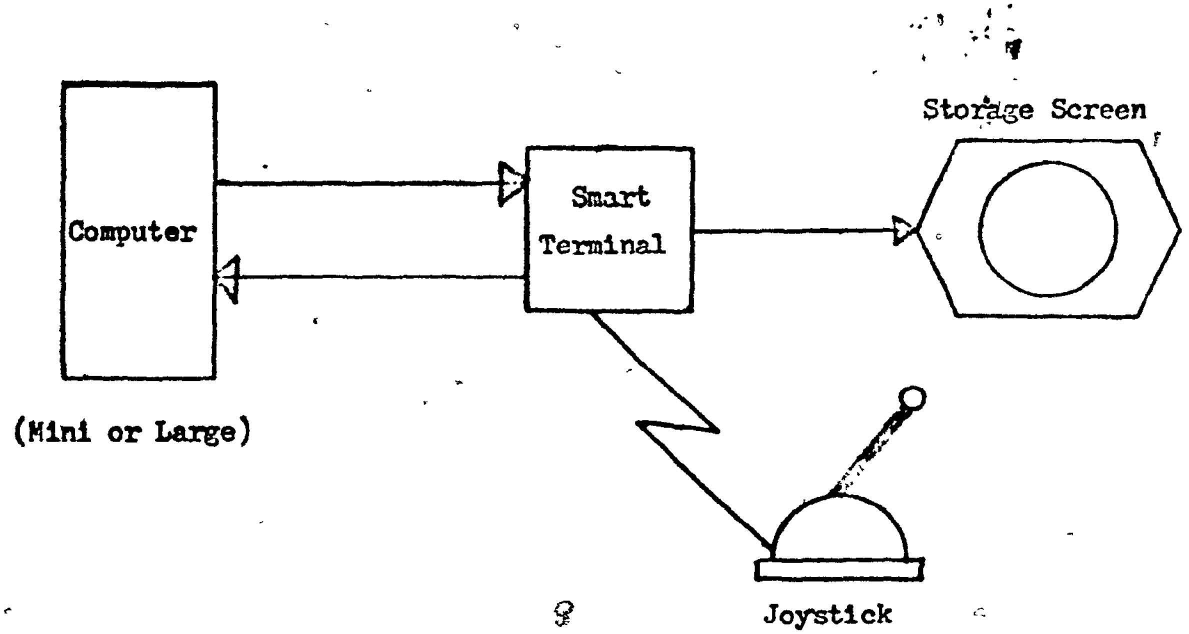


FIGURE 2.5. STORAGE SCOPE GRAPHICS.

2.3 Graphics Software

2.3.1 Introduction

Software implies the data structure used to organize stored information, the operating system programmes used to access this data and handle input/output devices, and programming languages which allow the user to write programs to solve his problem. In this section, we shall briefly review the software used in interactive graphics systems with a view to making some judgement statements concerning what is required to build a good graphics display. All the systems mentioned here drive a calligraphic display since this affords the best interaction between man and machine. The use of other displays would alter significantly the software design.

First, we shall illustrate by an example how a graphics program builds its data structure from the operator's light pen inputs. We then shall present other data structures capable of the same function but having added flexibility. The discussion of data structures is followed by a review of graphics languages and their use.

2.3.2 Data Structures

A characteristic of graphic displays is their ability to generate large complex drawings starting from a few primitives, or in the limit, from a blank screen. During the drawing process, many parts may be drawn, moved and discarded, as the operator sees fit. It is evident a dynamic memory management scheme along with a flexible data structure is needed to represent the picture as it expands and shrinks. To get a feel for how this is done, let us look at the data structure in "Sketchpad" and see how it is manipulated during the drawing exercise described at the beginning of this chapter.

All geometric objects in Sketchpad are defined by their component parts. For example, in the triangle constructed at the beginning of this chapter, we define the line "L1" by points "P1" and "P2" and likewise "L2" in terms of "P2, P3" and "L3" in terms of "P3, P1". Triangle T1 is in turn defined by lines "L1, L2, L3". The data structure preserves these hierarchical relationships using a system of rings implemented as two-way lists. Each object is represented in memory by an n-component block of contiguous registers (Figure 2.6) containing the block type, (e.g. point, line) pointers to the next member of its ring, and in the case of point blocks, its X, Y co-ordinate position. For example, in Figure 2.6, line block "L1" heads a two-directional list containing point blocks "P1" and "P2".

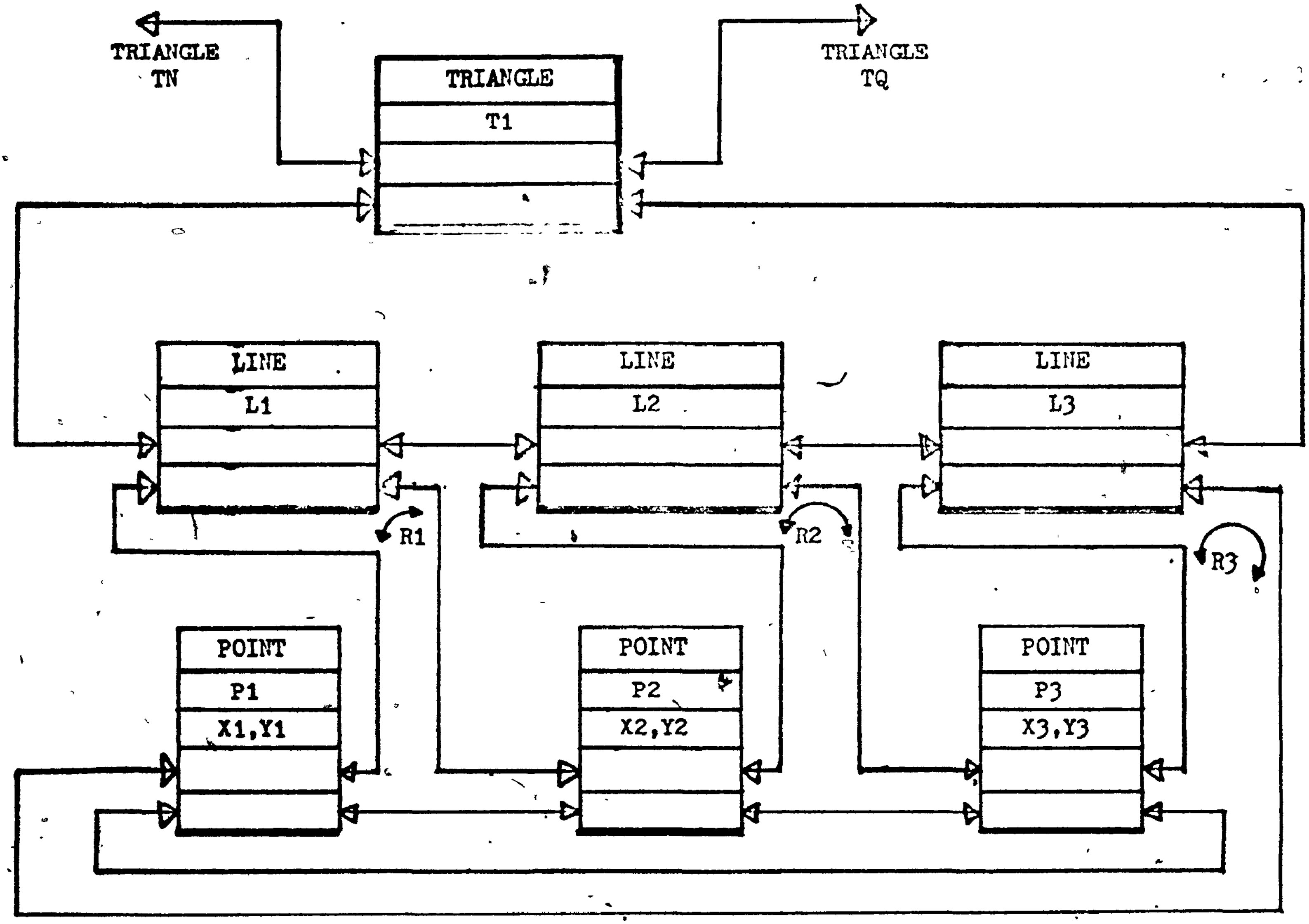


FIGURE 2.6. SKETCHPAD DATA STRUCTURE.

Similiarly the triangle block "T1" heads a ring composed of line blocks "L1, L2, L3". Note each block has a pair of pointers for each ring, one pointing forward and one pointing back. When adding or removing a block to a ring the pointers in the preceding and following blocks must be changed. These blocks are easily found using the forward/reverse pointer pairs.

Although not included in Sketchpad, a third pointer to the start of the ring is a useful device and is sometimes implemented. For example, let's select triangle "T2" in our display by hitting line "L2" with the light pen. To find the triangle to which "L2" belongs, we must chase the pointers around the ring until the ring head identifying the triangle is reached. If each block had a third pointer to the ring head, this information is found directly. This scheme of pointer triplets creates overhead which is usually intolerable in minicomputers. Hence, compromises are made. For example, CORAL[5] uses two pointers per ring in each block. One is used as a forward pointer. The other is used as a ring head pointer in every second block and a back pointer in every other one. (See Figure 2.7).

Note that in these two systems no mention is made of the DPU commands needed to drive the display. Sketchpad used a program to traverse the data structure and generate a contiguous list of X, Y positions of in-

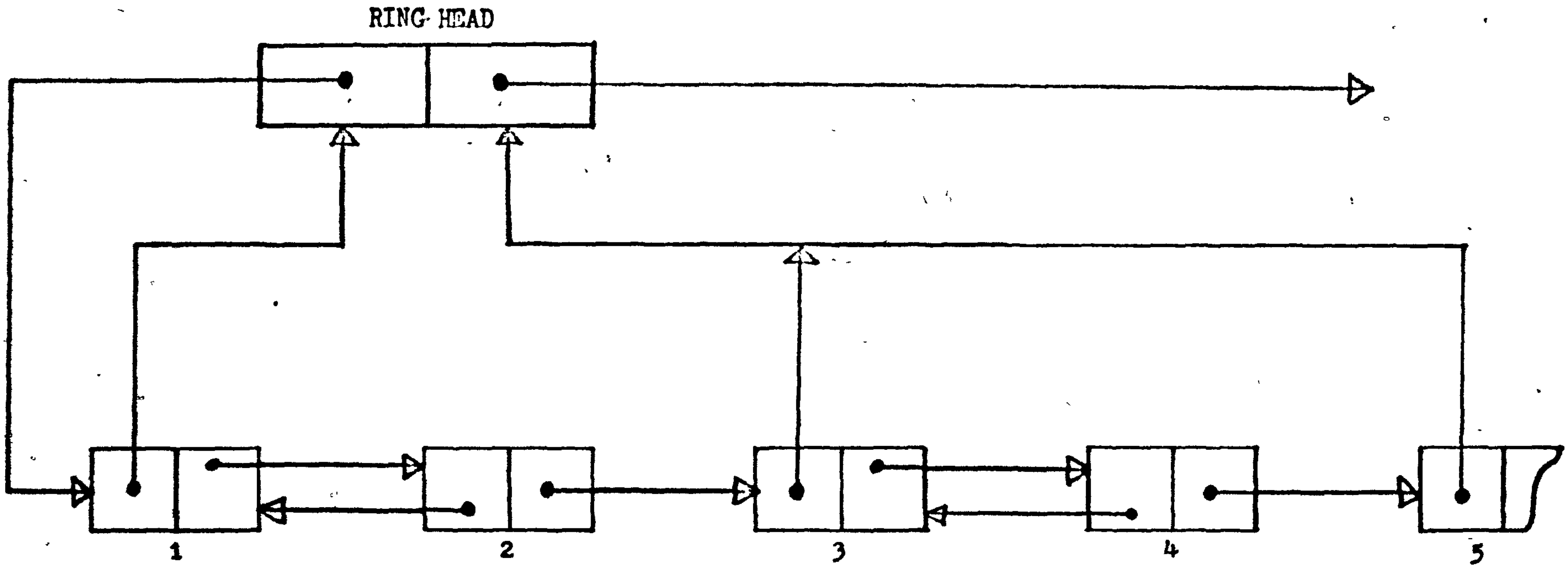
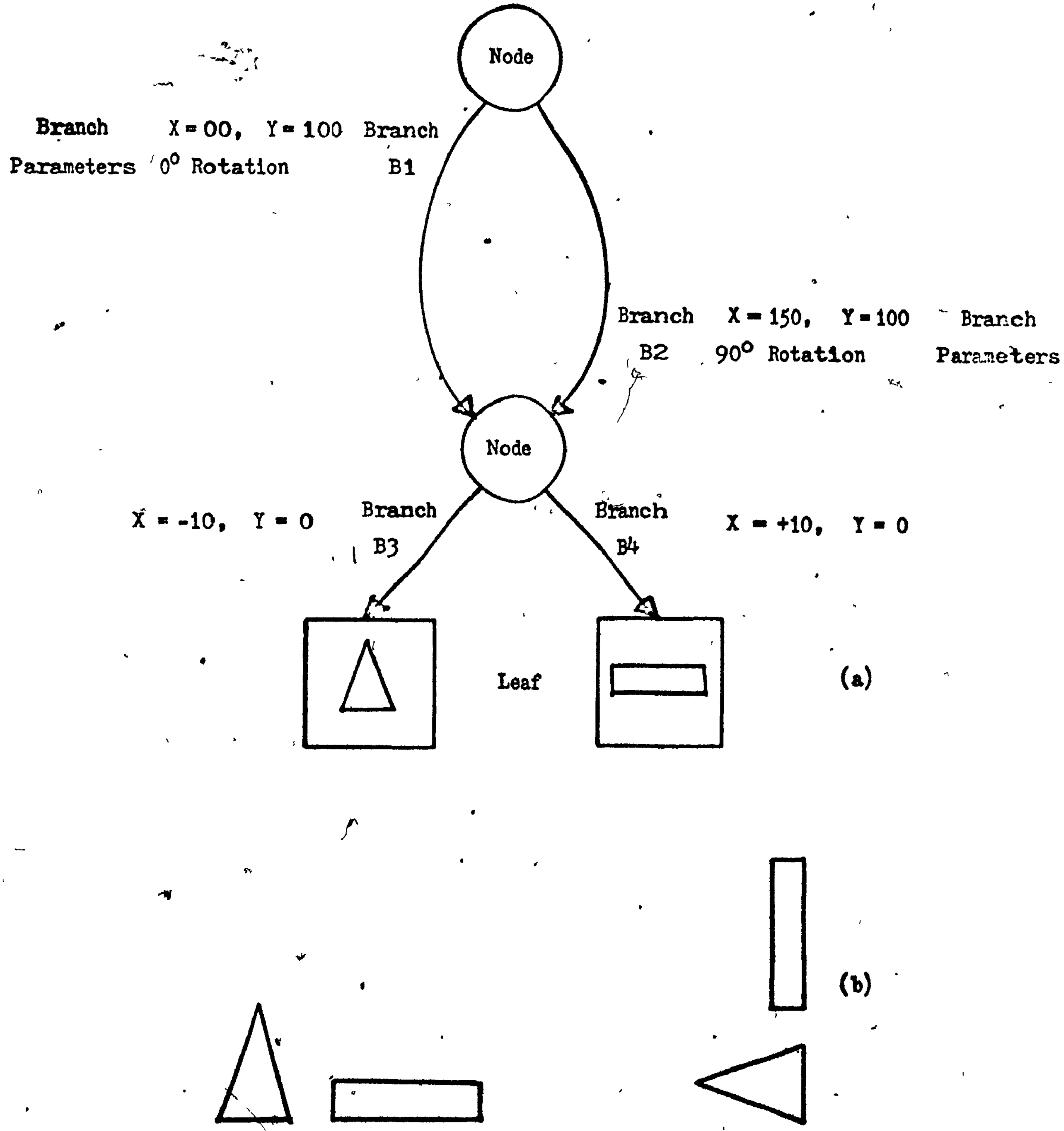


FIGURE 2.7. CORAL DATA STRUCTURE.

tensified points. The rudimentary DPU read these values and generated the corresponding image. An alternate approach is to imbed the DPU commands in the data structure itself. BELLGRAPH's[11] data structure does this in a clever fashion which effectively provides display subroutining.

BELLGRAPH's data structure is a directed graph (with no closed loops) comprised of a set of nodes connected by branches (see Figure 2.8). The branches have a direction associated with them, i.e. they point from one node to another. The terminal nodes called leaves have special significance in that they are the only blocks containing displayable commands. Each node represents a particular sub-part in the picture and each branch represents a particular instance of the node to which it points. The display parameters associated with each instance (e.g. position, scale, orientation) are stored in the branch block. As the display program traverses the data structure, the branch addresses are stacked, which effectively stacks the display parameters, thereby achieving true display subroutining as mentioned in the previous section. Figure 2.8 illustrates the effect of changing the orientation parameter in a branch block. Unlike Sketchpad, the basic primitives (i.e. leaves) can be defined to be more than just single lines thereby reducing considerably storage requirements. In Figure 2.8 the smallest indivisable picture elements are triangles and rectangles.



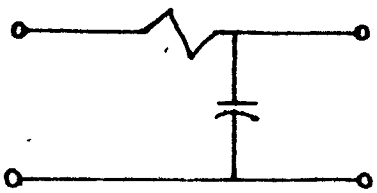
Resulting Display of Above Data Structure.

FIGURE 2.8. BELLGRAPH DATA STRUCTURE.

One of the major problems confronting designers is to find a data structure containing the graphic data such that it can be quickly supplied to the display processor for display refresh, and yet contain sufficient information for the background analysis program. For example, Figure 2.9 illustrates two representations of the same circuit. One is used as input to the display programs the other for frequency response calculations.

BELLGRAPH approaches this problem by including non-display data blocks, connected to each branch, used to store information concerning the instance.

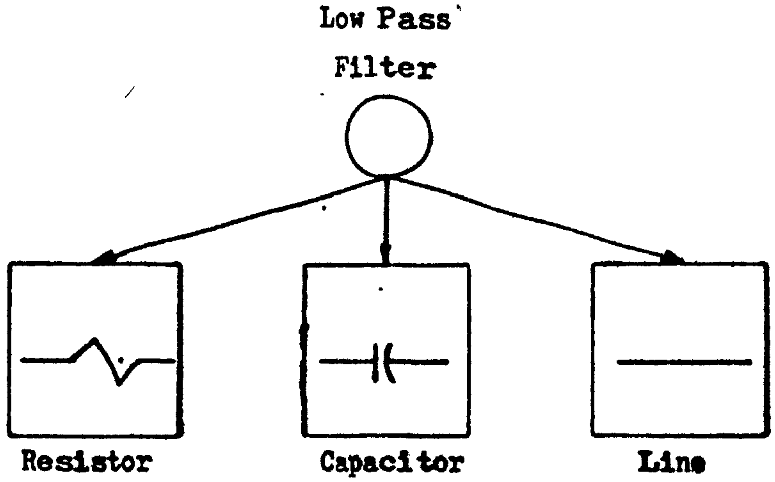
Another technique is to use two data structures. One, stored at the central computer, defines the organization of the pictures and stores information pertinent to the analysis programs. The other data structure is maintained at the remote station and is used solely for display and attention handling. The UNIVAC graphics system[12] for example, has a data structure in the main computer which contains all information concerning the display data. A condensed version of this structure giving the organization of the currently displayed picture, is concurrently maintained at the terminal along with another data structure used for display and operator selection.



Graphic Representing of Low Pass Filter

$$H(jw) = \frac{1/jwc}{R + 1/jwc}$$

Mathenatical Representation of Low Pass Filter Frequency Response



Computer Graphics Data Representation

FIGURE 2.9. DIFFERENT REPRESENTATIONS OF ONE PROBLEM.

2.3.3 Data Structure Summary

There are very many data structures used in computer graphics each having its own special features. We have left the reader to explore the many structures described in the literature and have instead attempted to illustrate the important features common to most. Two good surveys of graphic data structures are given by Gray[9] and Williams.[10]

We have presented the basic concepts universal to graphics data and can now arrive at the attributes of a good data structure. That is, a data structure should do the following:

(1) Store a picture in computer memory and preserve the hierarchical relationships between its parts.

(2) Permit rapid access to all levels of the structure enabling the user to select any picture part for removal or modification.

(3) Have a mechanism for allowing the data to expand and contract as the user operates on his picture.

(4) Permit rapid searching, deletion, or addition to keep up to the operator's requests.

(5) Provide storage and access for other non-display data associated with any picture part.

2.3.4 Graphics Languages

There is even less agreement concerning graphics languages than there is on data structures. Each year more papers are published presenting yet another graphics language. A possible cause for this proliferation of languages is the application of graphics to many diverse problems, each with its own constraints and criteria.

In the following discussion we will emphasize those languages used in computer-aided design, and later touch upon the special requirements of general picture synthesis and analysis as required in pattern recognition systems. This discussion of graphics languages will probably be clearer if we first present a sample computer-aided design problem, then apply a number of languages to it and judge their utility.

Consider a circuit designer using a graphics terminal to develop a particular device. First he must define the primitives he wishes to use (e.g. resistors, capacitors). Depending on the system he would do this off-line by inputing a deck of cards at the central computer or use the light pen to draw them on-line. In either case he must use a language of sorts. Note that when we talk of programming languages, especially graphics language we refer to a method of describing a sequence of operations to be executed on a set of data entities.

The alphabet of a graphics language need not be a set of characters, but rather includes pushbuttons, light buttons, and light pen movements. The syntax of the language gives the order in which these various functions are performed. For example, the statement in Sketchapd to define a circle centre is

> "move tracking cross to the desired location
> and depress CIRCLE CENTRE pushbutton".

The syntax is "select a point and depress pushbutton". The statement to draw a line has the same syntax. The two dimensional language of Sketchpad is rigidly defined by the system programs driving the display. It has a simple syntax of three or four statement structures. Sibley[24] extends this concept of two dimensional language by proposing one which enables the user to specify his own procedures within the language. He states,

> "Thus the description of a procedure (akin to writing a program) is done by motions of a light pen on a screen or pen on a tablet. These motions are neither the act of physical writing (using a character recognizer) nor defining a procedure by drawing its flow chart. In fact the motions are very similar to those a user carries out when he is executing a procedure using drafting equipment."

Sibley illustrates this point using such a language to generate procedures for drawing geometric objects.

Returning to our designer who now has his primitives defined, he is ready to build a model. Once defined, the system saves his primitives for future use. Note the general purpose graphics system also handles problems posed by logic designers and, structural engineers or chemists, each of who has his own set of primitives.

To be truly interactive the system should permit the user to copy, rotate, scale and connect together his fundamental primitives using the positional input device (e.g. light pen, joystick, etc.), and input component values via a keyboard. He can now analyze his model by selecting preprogrammed functions such as D.C. steady state response or A.C. response. At any time he may return to the other two modes and add new primitives and alter his model. Depending on the system, he makes his choices using light buttons, pushbuttons, the keyboard or some mixture of these. If the existing functions are not sufficient the operator may wish to define new ones using a graphics language on-line while he is at the terminal. The language should allow him to merely state an algorithm or procedure and not force him to prepare a complete program.

We can see from our example, an off-line programming language is needed to:

1. Define the picture synthesis procedures off-line.

2. Define analysis procedures which can be selected by the user on-line.

A nice feature is another language available to the operator to prepare additional algorithms or procedures on-line.

More precisely, the off-line language provides the system analyst a method of constructing programs which drive the I/O devices, manage memory, and handle the data structure. The off-line programs written once, and modified infrequently, define the capabilities of the graphics system and the syntax/semantics of the on-line language.

For example, in the first case there might be a technique for including a generalized light button, that is, one can specify the text to be displayed in a menu and programme the accompanying function. Using this, the system analyst may include a "MOVE" light button and then write a routine which permits the light pen to drag an object across the screen. Now, when the user turns on the graphics terminal, "MOVE" appears. This becomes a function in his language. Which language is the graphics language? Should languages be provided to the system analyst building a graphics system or should he be forced to use assembler? Should languages be provided to the user which allow him to develop his own algorithms and procedures besides selecting existing procedures? Perhaps the same language should be used for both, so that the system, through use, will bootstrap itself to higher capabilities. Note this implies a common language for both picture synthesis and analysis.

Roberts,[20] Kulsrud,[16] and Miller and Shaw[15] all agree that the same graphics language should be used for both synthesis and analysis. Miller and Shaw states:

> "The arguments for treating analysis and synthesis problems together, i.e., using a common description scheme, are generality, simplicity, and the universal use of common description languages in science. We also note that most picture analysis applications have (and need) an associated generative system and vice versa; there are also many situations where both a synthesis and analysis capability are equally important, for example, in computer-aided design."

If such a language became universally accepted, a growing pool of portable software could accumulate enabling successive workers in the field to build upon existing programs, independent of the computer and graphics devices used. To gain portability many of the familiar high level languages such as FORTRAN were modified to support graphics.[13,14,16,17] For example, the language GRAF[13] is an extended FORTRAN containing an additional variable type called a "display variable" and a set of functions for operating on them. The statement

DISPLAY A, J, Q(17) TR1(7, 4, 5)

declares "A, J, Q and TR1" as display variables or display variable arrays. Pictures are drawn using functions such as PLACE, LINE, CHAR, and PLOT with display variables specified in their argument lists. In our sample computer-aided design problem, the circuit designer using GRAF would code his primitives off-line using these functions. Menus of light buttons perform program branching through use of the attention poling function "DETECT", followed by a list of conditional transfers (i.e. "IF" statements).

It is immediately evident that a heavy programming burden is placed on the user albeit in FORTRAN. Another disadvantage with this approach is the necessity for anticipation of inputs. In a true interactive system one would like to be able to interrupt the processor and restart at another point at any time. For example, if the user finds an error in his model while the computer is analysing it, he would like to abort the computation and restructure his model.

The "AIDS"[4] language attempts to circumvent this problem by formulating an automata approach to programming. Instead of a flowchart, the programmer builds an automaton model (see Figure 2.10) which defines a set of states and transitions between them.

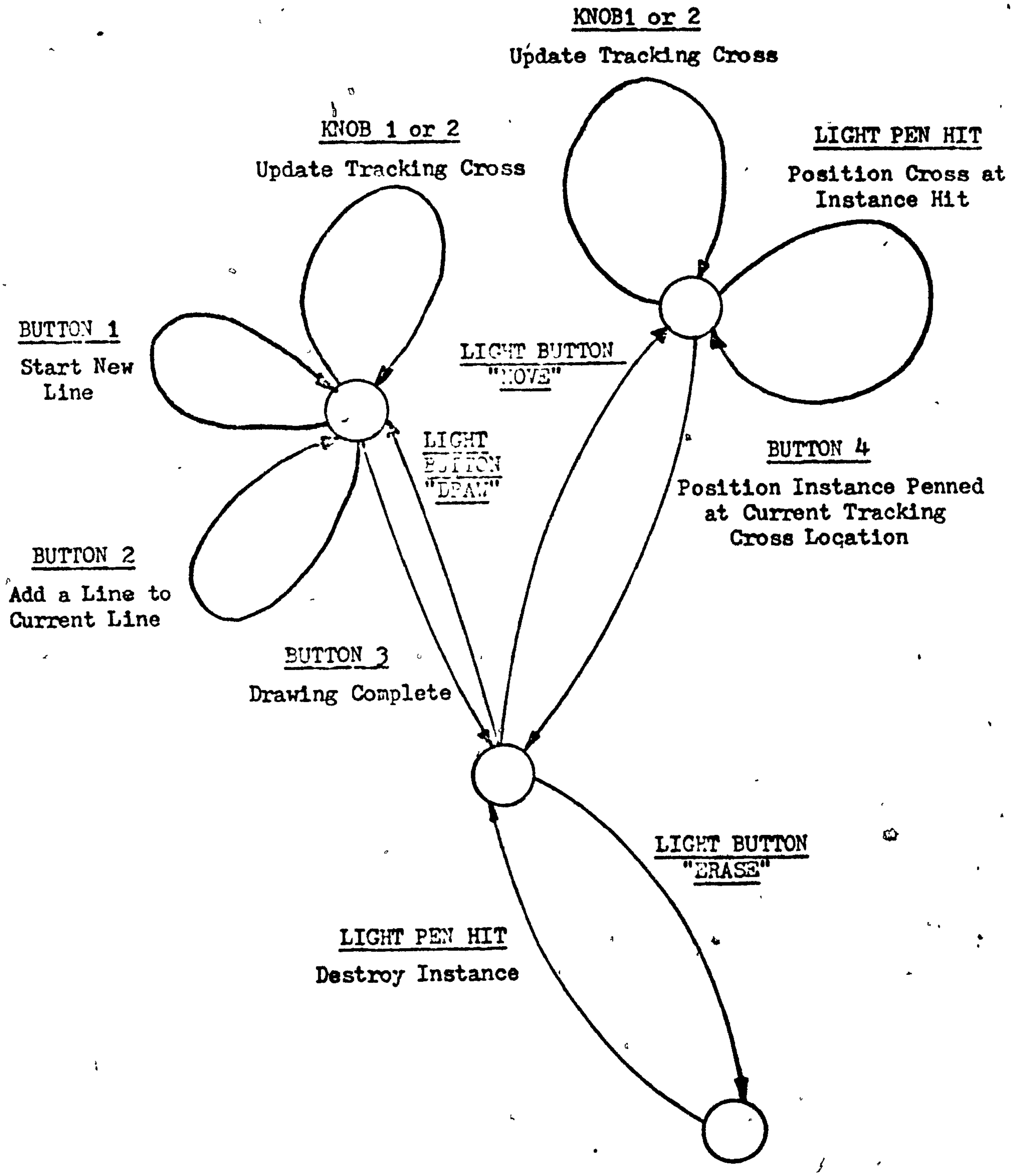


FIGURE 2.10. AN AIDS PROGRAM TO DRAW AN OBJECT, MOVE IT TO ANY DESIRED LOCATION, AND/OR DELETE IT FROM THE SCREEN.

The programmer can then write:

```
WHEN IN STATE n, IF condition a,
THEN .... response
```

where "n" is an integer identifying the state, "condition a" is an input device or the real-time clock which is activated by the operating system when the program is in state "n". The phase ".. response" is a FORTRAN statement or group of FORTRAN statements.

This system also supports a hierarchical graphical structure similar to BELLGRAPH's.[11] AIDS programs are first processed by a pre-compiler which extracts the graphic and interactive statements and passes the necessary information to the operating system. The remaining FORTRAN statements are compiled using a standard FORTRAN compiler.

H.E. Kulsrud[16] proposes the use of a meta-compiler or compiler-compiler system for the study of graphics languages. A meta-compiler accepts commands which define the syntax and semantics of a new language and produces a compiler. Language statements in this new language are then processed and executed using this compiler. To avoid the input anticipation problem, incremental compilation is used to produce open-ended programs which can run at all points in time. That is, procedures are compiled line-by-line and then executed on demand. In a sample system built by Kulsrud, each operator or function in the test language evoked a subroutine from a system library. These subroutines may be written in any language. Therefore, input/output functions could be coded in assembler for fast response, and the computational operations in FORTRAN for quick implementation. A multilingual basis for a common supra language appears a promising approach to the study of graphics language.

There are many other languages too numerous to discuss here. We shall let the interested reader review the literature.[17,24-29]

2.3.5 Graphics Language Summary

All graphics languages, although differing in attributes and capabilities depending upon their special applications, permit some form of picture synthesis. That is, each must have the basic capability to:

1. Position points, lines, and text on the screen.
2. Draw copies of any displayed picture part.
3. Move picture parts around on the screen and rotate and scale them.
4. Provide display management.

To be interactive the system must:

5. Accept text and positional information from a keyboard and/or light pen, joystick, etc.

6. Not anticipate inputs.

7. Be flexible enough to require little preplanning.

8. Provide rapid response.

9. Provide for user errors, allowing him to correct errors in a simple straight forware manner.

We now can itemize those options which make a graphics language powerful or high level. First, we require the picture be described in memory in hierarchical levels, which can be quickly accessed in real-time. Also, it should allow the user to input and test his own algorithms and procedures without having to write a complete program. The language should use notation familiar to the user. A subroutine capability and a subroutine library initially containing a basic set of functions would be useful. This makes the language effectively open-ended, allowing the user to expand it as the need arises. A language permitting easy extension of its capabilities is applicable to many classes of problems and hence ensures greater general acceptance.

In Chapter III we will introduce the "GRIN" language developed at Bell Laboratories and later moved to McGill. This language is specifically designed to synthesize pictures and contains many of the desirable features above. In Chapter IV, we will discuss some of these features and the problems encountered in transfering GRIN from its home environment to McGRAPH. In the Conclusions, proposed future work to enlarge "GRIN's" capabilities is presented.

CHAPTER III

THE McGRAPH SYSTEM

3.1 Introduction

McGRAPH is a general purpose computer graphics system to be used for research in a variety of areas ranging from logic and circuit design to image processing and pattern recognition. Because of its many different uses McGRAPH's software must be sufficiently flexible to support a wide range of applications and yet powerful enough to remove the user from the programming chores of I/O communication and data management. The system should provide the user a simple direct method of constructing graphical representations of his problem and enable him to define interactive problem-solving procedures tailored to his particular application. These procedures should be easy to build and modify, allowing the user to experiment with various approaches.

The task of effectively interfacing graphics hardware to problem solving is not trivial and is usually underestimated. Ninke[22] used the triangle shown in Figure 3.1 to illustrate the programming effort needed to achieve a good interface. The base of the triangle represents the device dependent programs needed to drive the display hardware. System support programs for file management and multiprogramming comprise the rest of this foundation. Upon this base he proposed a

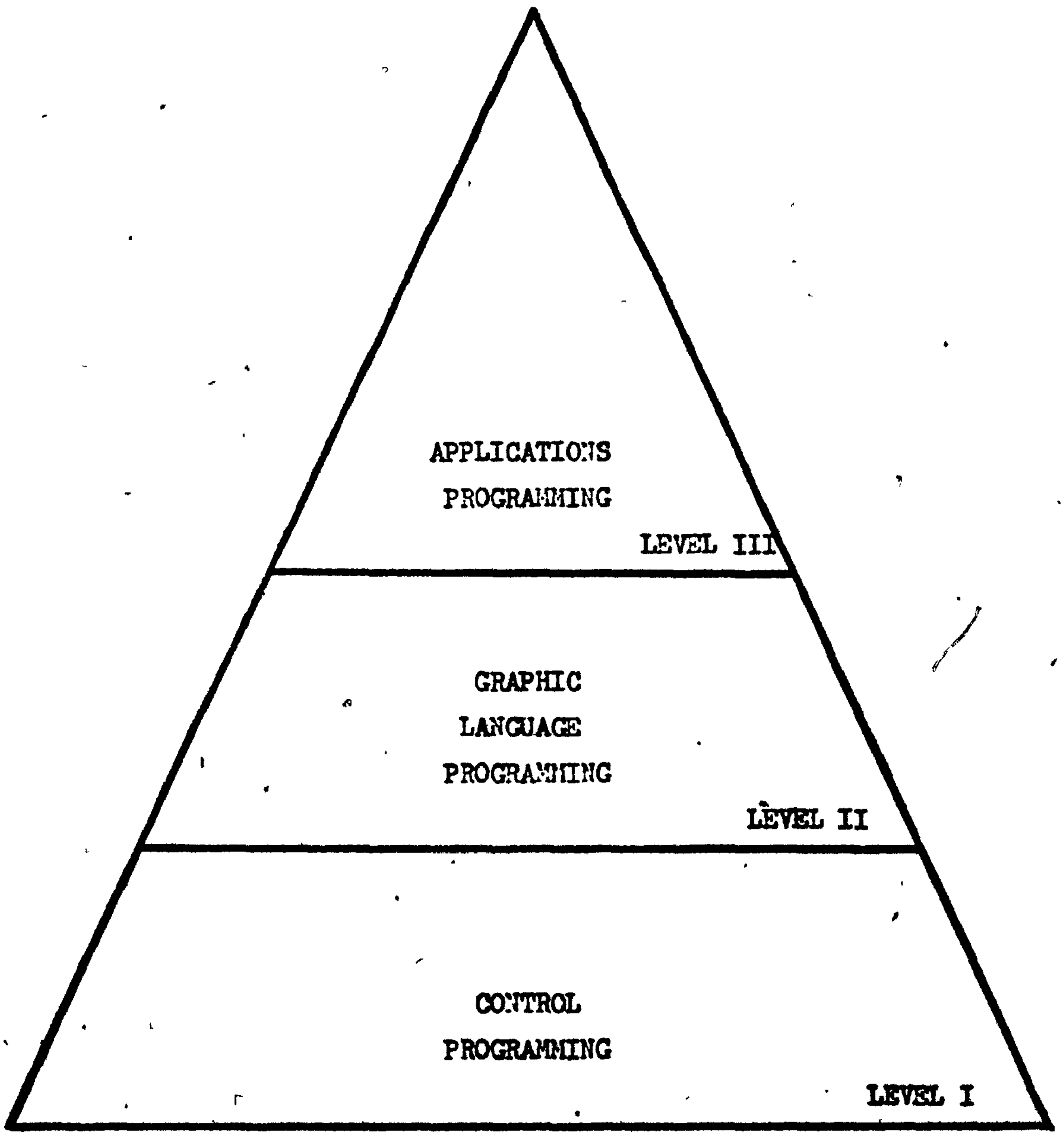


FIGURE 3.1. PROGRAM PLAN TO ACHIEVE A GRAPHIC INTERFACE CAPABILITY.

programming language for generating the application oriented graphics system. Note that this language is not used explicitly by the problem solver. Its function is to aid in the construction of a specific graphics system applied to a given class of problems.

Programs written in this language define the procedures and features of the graphics system. For example, these programs could display a menu of light buttons available to the user and provide the corresponding functions. In a more powerful system, the system programs could interpret statements presented by the user in an on-line graphics language. Ninke places these application programs at the apex of the programming effort triangle (Figure 3.1).

The McGRAPH system software described in this thesis consists of support at levels I and II. A executive program (Level I) handles all the I/O devices and performs memory and file management. The graphics language "GRIN" (level II) operating under this monitor aids programmers in constructing interactive graphics systems.

The rest of this chapter is an introduction to McGRAPH. The next section describes the hardware configuration used. The following sections provide an overview of its software.

3.2 McGRAPH's Hardware

Figure 3.2 is a block diagram of the devices comprising McGRAPH. The heart of the system is a calligraphic display refreshed from a disc. The other components of the graphics system are a light pen and joystick for positional input and two computers (24K word PDP-15 and 4K word PDP-8) with a teletypewriter attached to each. Other peripherals include two one quarter million word discs on the PDP-15, high speed paper tape punch and reader on both machines, as well as two DEC tape drives on each. There is also a T.V. monitor refreshed from the display disc for outputting grey level pictures. Another CRT, with a Polariod camera attached, is connected to the graphics display processor for taking pictures. All graphics devices are connected to the PDP-8, which in turn passes data to and from the PDP-15 along a core to core link. The PDP-8 is used as a programable I/O device controller. It accepts commands from the PDP-15 and channels data to the appropriate output device. Correspondingly, data from input devices is preprocessed and sent to the PDP-15. By inserting the PDP-8 between the main frame and the devices, one can emulate a powerful device controller and experiment with its command repetoire to achieve maximum throughput. For example, by performing input data compression and output data expansion in the PDP-8, PDP-15 I/O processes are quicker, leaving more time for computation.

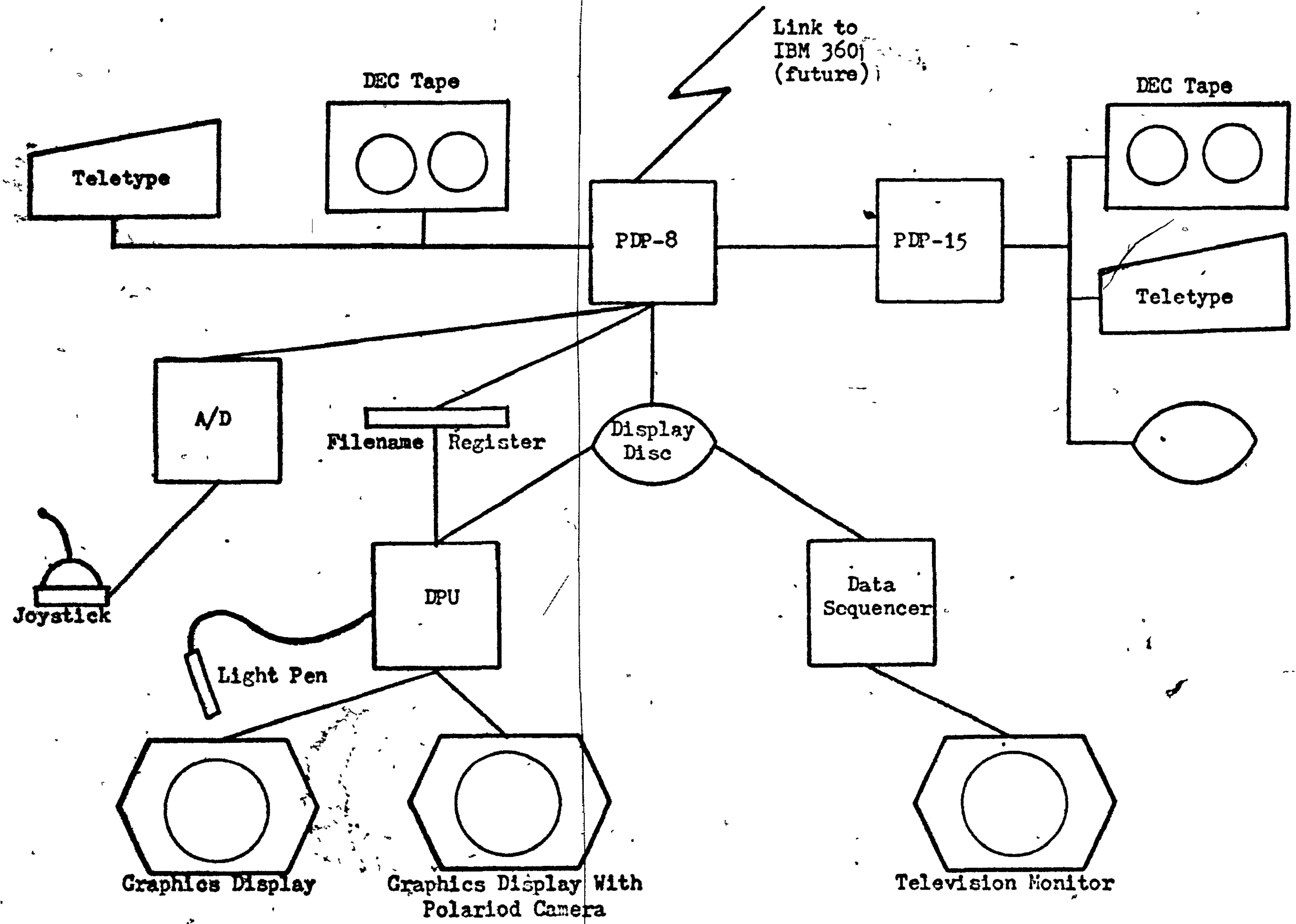


FIGURE 3.2. McGRAPH HARDWARE.

3.3 The Display Processor

As the display disc revolves, a selected track read head continuously transmits a list of 8 bit words stored on the track to a display processing unit (DPU). The display processor interprets this data as a string of commands for driving the X, Y and Z inputs of a CRT. These beam intensity and move commands trace out the required picture. Screen refresh is performed on each revolution of the disc; that is, 30 times a second. Therefore, a picture corresponding to one full track of data (12K commands) can be displayed without flicker. It is possible to display more complex scenes with some flicker by displaying the contents of two or more tracks through the use of a "branch to another track" DPU command. The beam control DPU commands allow one to:

1. Position a point.
2. Draw a vector with 3 bit displacements in X and Y.
3. Adjust the scale in either X or Y.
4. Rotate an image about the X or Y axis.
5. Adjust the intensity.

Scale, position and intensity commands may load in a new value, or add or subtract it to/from the existing one. This is especially useful in the case of point and vector moves, for it allows one to select an origin or starting position of a picture using absolute X, Y point moves, and then trace out the picture using relative vector and point moves. Now, if at any time we wish to move this picture to a new position, only the first X and Y absolute point moves need be changed.

Display data is passed back to the PDP-8 from the display processor by the HEADER DPU command. The HEADER is a two word (16 bit), command containing a 12 bit constant called a filename I.D. Each time a HEADER is executed this filename I.D. is stuffed into a 12 bit register which can be read by the PDP-8. The contents of this register remain static until the next HEADER. The graphics software places a HEADER before each string of commands corresponding to a seperate picture entity. For example, the DPU command list for each instance of a BELLGRAPH leaf is preceded by a HEADER containing a unique filename I.D.

The filename I.D. is used to identify picture parts on a light pen strike. On a light pen interrupt, the PDP-8 reads the filename I.D. register and passes its contents to the PDP-15. Programs in the PDP-15 then determine which object was selected and takes the appropriate action. Chapter IV discusses the picture identification technique in more detail.

The HEADER command is also used to synchronize the PDP-8 programs with the display disc. A HEADER command contains one bit, which when set, causes a PDP-8 interrupt each time the HEADER is executed.

In this section we have provided a brief overview of the operation and instruction set of the display processor. A complete description of the DPU is found in R. Fabi's thesis "The Design and Construction of a Disc Oriented Graphics System".[30]

3.4 Disc Display Refresh

Refreshing a calligraphic display from a disc is a novel method of obtaining powerful graphics without using a large central computer.

The disc relieves the need of refreshing from core, thereby freeing main memory for other tasks such as background computing. In all interactive graphics systems, the time between input attentions from the user is orders of magnitude greater than that needed to display one refresh cycle. Therefore, in systems refreshing from main memory the computer spends a lot of its time preforming a highly repetitive routine. That is, the computer must repeatedly traverse the data structure and output commands to the DPU. This procedure reduces the amount of time available for other computations.

In an effort to recover this lost time, some systems build a contiguous list of display commands in main memory from one pass of the data structure. Using this approach one sacrifices the memory required for this list. By placing the list on a disc instead of in main memory, we free all the computer's resources (including its DMA channel) from the refresh load. Another advantage of discs is their large storage. Further system optimization could be achieved by putting a number of pictures on the disc and allowing the DPU to do simple display switching. The display could then be changed in response to input attentions without requiring computer intervention. For example, pictures connected to individual light buttons could be selected using hardware.

The disadvantage of disc refreshed graphics is the cost of a dedicated disc and controller. In McGRAPH's environment, however, a disc is essential for refreshing the T.V. monitor. The cost of the four extra tracks used for graphics is negligible compared to dedicated core memory.

3.5 McGRAPH's Software

The McGRAPH software is designed to achieve a general purpose interactive graphics system easily adapted to any given application. The system performs basic display handling such as:

1. Drawing pictures from a file assembled off-line.
2. Drawing pictures on-line using a light pen.

3. Moving, rotating, scaling, and copying picture parts.

4. Decompose (or construct) pictures into (from) component parts.

5. Provide the user a means of calling attention to any of these parts by light pen or joystick.

6. Accepting inputs from keyboard, pushbuttons, light pen and joystick.

7. Provide display and memory management.

To make it adaptive a graphics language is provided. This language has the following features:

1. Enables one to program all of the above display handling functions.

2. Is capable of performing some arithmetic.

3. Include conditional branching so that program flow can be altered by arithmetic results.

4. Enables subroutining.

5. Is open-ended. That is, allows expansion of the language by incorporating new functions.

As Ninke[22] pointed out, development of such a system and language involves typically 40 man-years of work, an unreasonable expenditure in the case of McGRAPH. An alternate approach was to acquire an existing system and modify it to suit the requirements of McGRAPH's disc based graphics hardware.

BELLGRAPH and its associated language GRIN, developed by Bell Laboratories at Murray Hill fits the above design requirements very well. The fact that BELLGRAPH also uses a PDP-15 is especially attractive. Also, although BELLGRAPH is used in conjunction with a large remote central computer, the PDP-15 programs are sufficiently comprehensive for stand-alone operation. In fact, the PDP-15 resident software completely supports the functions of level I and II in Figure 3.1.

The PDP-15 programs consist of device handlers for a PDP-15 disc, teletype, and high speed tape reader as well as all graphics devices such as a DEC 339 display processor with light pen, console keyboard and eight back lighted pushbuttons. There is a core resident on-line monitor or executive system used for memory management, and GRIN program interpretation. A library of subroutines and associated dictionary are kept on disc for use by the executive system. A number of other PDP-15 programs have been supplied for off-line support of the system. For example, there are debugging aids and linking loaders for GRIN programs. One off-line set of programs labelled G2LIBE has been implemented on McGRAPH to build and edit the disc resident library. An assembler for GRIN programs which executes on an IBM 360 is also available. Another GRIN assembler residing in the PDP-15 has not been implemented yet. The remainder of this chapter introduces each of these software packages. Chapter V is devoted to the IBM 360 assembly system.

Two basic problems required solution in order to install this BELLGRAPH software on McGRAPH. First, the hardware control programs had to be altered to drive our disc refreshed system. Second, the monitor programs were changed to account for addressing differences between McGRAPH's 24K computer and BELLGRAPH's modified 8K processor. Further discussion of these problems and their solutions is given in Chapters IV and V.

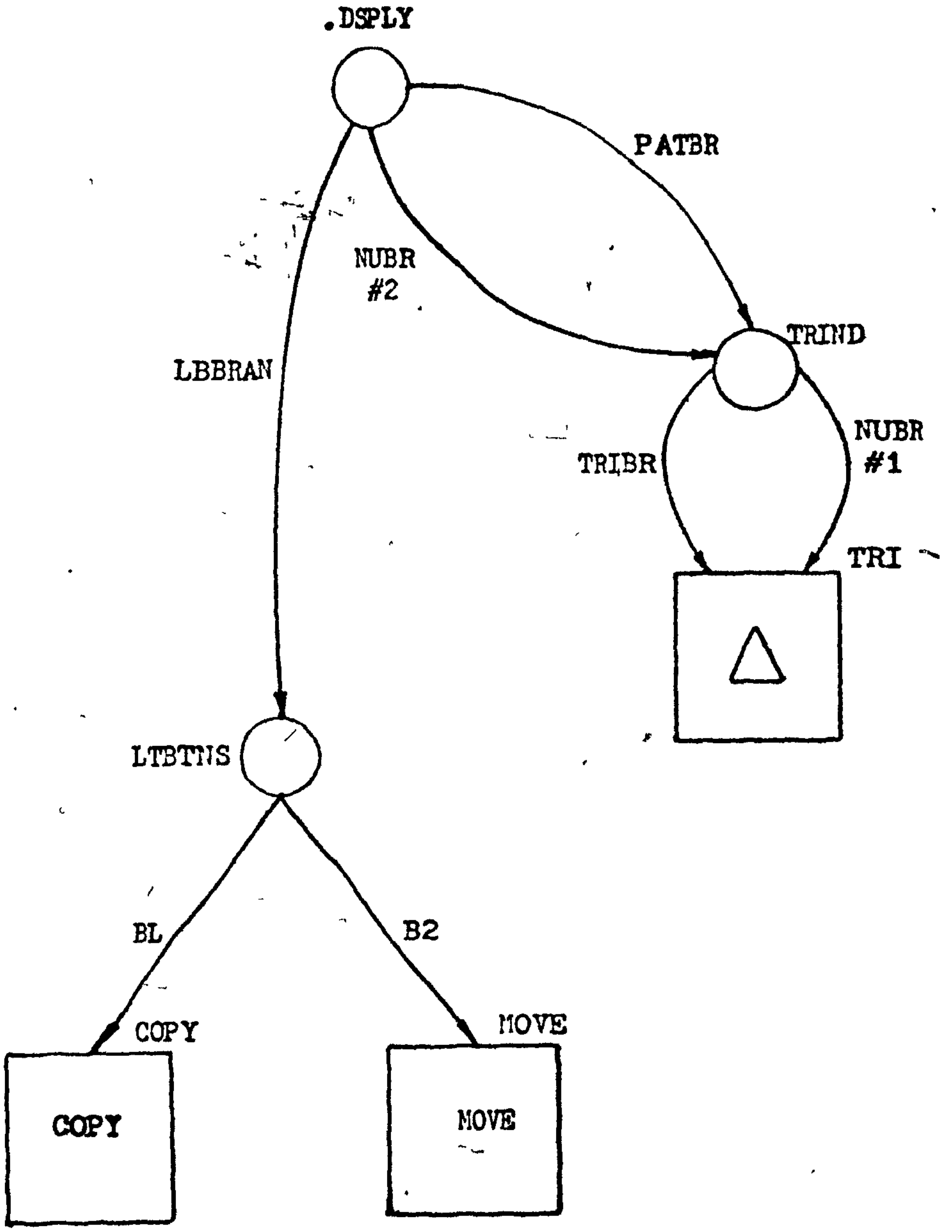
3.6 The GRIN Language

Perhaps the best way to introduce GRIN is by use of an example. Let us write a GRIN program (Figure 3.3) to draw a pattern using triangles as in Chapter II. The associated data structure is given in Figure 3.4. Statements 1 to 4 produce a leaf labelled TRI containing commands to draw the basic triangle. Statements 5 to 8 define two leaves used in the light button menu. The menu is created by joining these two leaves to a common node labelled "LTBTNS" in statement 9. The branches (B1, B2) from this

```
Statement No.              GRIN2 Statement                  Comments

  1              LEAF      TRI                     LEAF TRI CONTAINS COMMANDS
  2              VECTOR    #0,20                   TO DRAW AN EQUILATERAL
  3              VECTOR    10,-20                  TRIANGLE
  4              VECTOR    -20
  5              LEAF      COPY                    LEAF COPY CONTAINS THE TEXT
  6              TEXT      (C,O,P,Y)               'COPY'
  7              LEAF      MOVE                    THIS LEAF CONTAINS 'MOVE'
  8              TEXT      (M,O,V,E)
  9              TREE      LBTNS((B1,,,COPY,COPG)(B2(0-30),MOVE,MOVEPG)
                                                   CONNECT LEAVES TO NODE 'LBTNS'
                                                   USING BRANCHES 'B1,B2'
 10              BRANCH    LBBRAN,,(950,0),LTBTNS
 11              NODE      TRIND
 12              BRANCH    PATBR,,,,TRIND
 13              BRANCH    TRIBR,TRIND,(500,500),TRI,.LPON
 14              NEWDSP    (LBBRAN,PATBR)          DISPLAY TRIANGLE AND LIGHT BUTTONS
 15  HOME        WHICH     .LBTRA
 16              GOTO      HOME
 17  COPG        WHICH     .IGNOR                  THE COPY PROGRAM
 18              COPYBR    .CURBR,NUBR             CREATE A NEW COPY OF THE
 19              GOTO      HOME                    CURRENT BRANCH AND GO TO HOME
 20  MOVEPG      MOVE      .WHICHB
 21              WHERE     .HOME
```

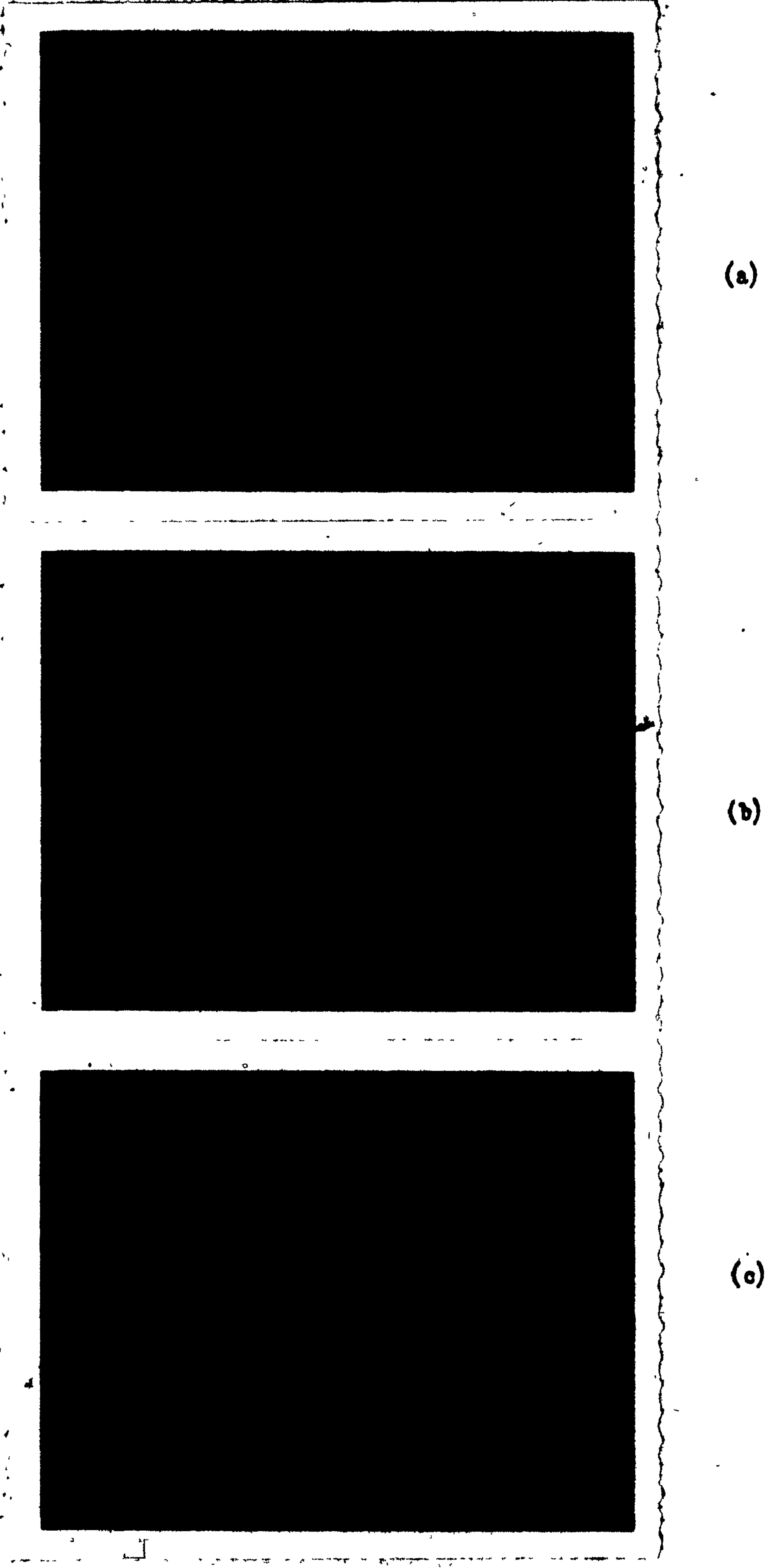
FIGURE 3.3. A SAMPLE GRIN2 PROGRAM.



Data Structure Generated by the GRIN Program of Figure 3.3.

FIGURE 3.4 (a)

(continued)



Display Sequence Resulting From GRIN Program in Figure 3.3.

FIGURE 3.4 (b).

node represent the light buttons. Each branch (B1, B2) has a relative co-ordinate position and transfer address associated with it. For example, light button "MOVE" is 30 units below "COPY", the menu's origin, and has transfer address "MOVEPG". That is, on a light pen hit of "MOVE" the program will transfer to MOVEPG. Statement 10 connects a branch labelled LBBRAN to the menu node and specifies its position (i.e. 950, 0). Similarly we connect a node to the triangle leaf via branch TRIBR and a branch PATBR to this node in statements 11, 12, 13. The entire picture consisting of the menu and triangle is defined by joining branches "PATBR" and "LBBRAN" to the display node in statement 14.

The picture is displayed by the "WHICH" function of statement 15. The .LBTRA argument specifies that when a light button is picked control should pass to its associated program. If a non-light button were picked control would pass to statement 16 which transfers back to 15, thereby forcing the user to select a light button.

Assume the COPY light button is picked. Control is passed to the WHICH function in statement 17 which waits this time for a non-light button selection (the .IGNOR argument). The triangle will blink when hit with the light pen providing the user feedback. Three extra light buttons, MORE, OK, and LESS, will appear at the bottom of the screen.

The operator can traverse up and down the tree using these light buttons. For example, by picking MORE, branch PATBR will be selected. Now if there was another leaf connected to node TRIND, both leaves would blink. The operator exits the WHICH function via the OK light button. Statement 18 creates a new branch to the triangle which creates a new instance on top of the original. The new triangle can now be moved by selecting the MOVE light button. The MOVEPG program's WHERE function displays a tracking cross used to define the position of the new triangle instance. But first, the MOVE function in statement 20 is executed specifying the branch to be moved. In this case it is WHICHB, that is the last branch picked by the WHICH function. The argument HOME in the WHERE function specifies control should pass back to HOME (i.e. statement 15) when a key on the console keyboard is struck. The next time "COPY" is selected the operator can choose to copy one of the two instances displayed, or by invoking the "MORE" option of the WHICH command, both triangles can be reproduced and moved as a seperate subpicture. Figure 3.4 illustrates the sequence of displays in copying first the original triangle and then the resulting pair.

The above example is given to illustrate some of the capabilities of GRIN and introduce its syntax. A complete description of the language plus many tutorial examples is given in the BELLGRAPH Programmer's Manual available to any potential user of McGRAPH. Before leaving the discussion of GRIN, however, two general comments should be made.

First, new or potential GRIN programmers approach the seemly complicated argument lists with trepidation. However, given a few hours to write a sample program using the programmer's manual the sequence of arguments for the more common commands is quickly learnt. There has been a concious effort made in the design of the language to minimize differences between argument lists of two functions containing the same information. One simple example of this is the TREE function argument list which is composed of a list of BRANCH function argument lists. Such hierarchies of argument lists is prevalent throughout the language. The reason GRIN's syntax is that of an assembler language is to minimize construction time of the assembler and compilation costs of GRIN programs. A standard macro assembler (such as BAL on OS/360) using a library of macros to define the function calls is easier to implement and less expensive to run than a high level language compiler. Due to its smaller size an assembler is also easier to install on a minicomputer such as a PDP-15. The GRIN assembler is discussed fully in Chapter V.

3.7 The Operating System (G2SYS)

GRIN programs are prepared on paper tape by the assembler and read into the PDP-15 under control of the resident monitor or executive system. The GRIN programs consist of a mixture of their PDP-15 machine instructions, calls to subroutines in the monitor system, and data. The monitor subroutines control device interrupts and manage dynamic memory allocation. When a GRIN function statement is executed the monitor checks core for a contiguous free space, loads the corresponding language statement subroutine from disc, adjusts its relocatable references and passes control to it. This subroutine may in turn evoke other disc resident subroutines. These subroutines also may generate data blocks such as leaves, nodes and branches. When core space is needed, the executive system automatically deletes as many blocks as necessary. The programmer can override the executive's choice of delete candidates by special memory management functions in GRIN.

Details of the memory management scheme are given in the BELLGRAPH Programmer's Manual.

3.8 Graphics Device Simulator (G2SIM)

A graphics device simulator programs has been written to translate control commands for the DEC 339 into equivalent McGRAPH instructions. In this first phase of constructing the McGRAPH software, it was decided to minimize changes to the supplied BELLGRAPH software. One method of doing this is to simulate a DEC 339 display processor and associated devices using the McGRAPH hardware. Later, with a working system as back-up, one can alter the BELLGRAPH programs themselves to take advantage of the features of disc refreshed graphics.

To simulate the graphics devices, the definition of all their IOT instructions in the assembler were changed to subroutine calls to the simulator program G2SIM. This technique reduced the number of programming modifications to BELLGRAPH. A few changes were required, however, to account for fundamental differences between core refreshed and disc refreshed displays.

The display routine in the executive for example, was rewritten so that it makes only one pass through the display data. The IOT's in this routine evoke G2SIM for translating the display commands and writing them on the PDP-8's disc. It was found more advantageous to incorporate the tracking cross into the PDP-8 programs and allow the PDP-15 to access it as another device. This reduces the complexity and size of the "WHERE" and

"DRAW" function subroutines considerably. As experience with the system grows other potential reductions in the PDP-15's display load will be discovered.

A detailed description of the simulator program (G2SIM) is presented in Chapter IV.

3.9 The Library Editor System (G2LIBE)

G2LIBE is an off-line support program, supplied in the BELLGRAPH package, to edit the disc resident library. This program accepts typed commands to change one or more words on the disc, add new subroutines to the library from paper tape, and print out the contents of the library or dictionary. A user's guide to this program is given in **the available** BELLGRAPH documentation (memo by L. Rosler "Mass Storage Management Software for the GRAPHIC-2 and ADEPT Terminals, Issue 2-Case 39991-20).

Two changes have been made to this program. First, the library is stored on hardware disc #1 instead of #0 so that Digital Equipment Corporation (DEC) operating system residing on disc #0 is undisturbed. Second, the "Q" command to the library editor has been altered to return control to the DEC operating system (V5A or DOS) rather than to the BELLGRAPH executive G2SYS.

3.10 Summary

In this chapter we introduced McGRAPH, first by outlining its hardware and then by discussing its accompanying software package. In this chapter we have briefly described the function of each software segment. In the remaining chapters, we will investigate in detail, the method of operation of the graphic simulator program, and the PDP-15 assembler. The memory management system and GRIN language are discussed fully in the BELLGRAPH Programmer's Manual provided in the accompanying documentation. Following this we shall present our conclusions and suggest future improvements.

CHAPTER IV

GRAPHIC DEVICE CONTROL PROGRAMS

4.1 Introduction

The major obstacle in adapting the BELLGRAPH programs to McGRAPH is the very different display hardware of the two systems. BELLGRAPH repeatedly scans the data structure and outputs commands to the DPU refreshing the screen. McGRAPH need send this information only once to the display disc. BELLGRAPH programs expect, and use, a lot of real-time information at the time of a light pen hit. For example, the X and Y co-ordinates, core address of current display command, current display parameters, as well as software status such as the path taken through the data structure are immediately available upon an interrupt, A twelve bit filename I.D. is the only immediate real-time information given by McGRAPH.

Rather than rewrite large sections of the BELLGRAPH programs and lose all the advantages of an existing working system, we decided to write an additional set of programs to interface the BELLGRAPH software to the McGRAPH hardware. The effort required to interface BELLGRAPH is less than that to extensively modify it. This allows us to more quickly install the BELLGRAPH graphics system without sacrificing any of its powerful features. This chapter is devoted to describing these interface programs.

During the construction of the interfacing programs, a number of situations originally handled by the BELLGRAPH programs were found to be better suited within these interfacing programs. For example, when a vector move goes on or off the screen, BELLGRAPH expects an interrupt. The edge interrupt handler then turns on or off the beam intensity, preventing the picture from wrapping around and appearing on the other side of the screen. By placing this function within the simulator program we save the time consumed ~~in~~ servicing an interrupt, the space occupied by the edge handler in the BELLGRAPH managed memory, and space on the display disc occupied by non-intensified vector moves. This, plus other changes made to the BELLGRAPH system will be discussed in a later section of this chapter.

4.2 Basic Requirements

Having decided upon interfacing the BELLGRAPH monitor system to the McGRAPH hardware via software simulation programs, what specific tasks must these programs perform?

First, the BELLGRAPH programs are expecting inputs from two devices not available on McGRAPH: a console keyboard, and a set of eight lighted pushbuttons. Next, the DPU instructions for drawing vectors and points, and for setting DPU parameters (i.e. scale, intensity, orientation), must be translated into the McGRAPH instruction set. As shown later, this translation is not one to one, or indeed linear. The resulting McGRAPH instructions resulting from decoding one DPU command are dependent on the past commands as well as the current one. Some DPU commands have no equivalent in the McGRAPH instruction set, and their action must be simulated entirely by software. Details of the translation procedures are given in a later section of this chapter. There are a total of 36 input/output transfer (IOT) commands for controlling the graphics devices. The action of each of these must be interpreted in the context of off-line (disc) display refresh. For example, the "RLPD" (resume with light pen disabled) is used in BELLGRAPH to start up the display after a light pen hit. In McGRAPH's environment, the display does not stop on a light pen hit. Another IOT is "WBCS" (write display buffer and single step). Single step has no significance to McGRAPH. These IOT's need to be redefined in their new environment.

In writing the interface programs three additional design objectives were stipulated. The programs should be modular, with the physical device control being entirely contained in a few modules. Later changes or additions to the hardware will then impact on only these few modules. The inputs and outputs of the interface package should be well defined and generalized. In other words, it should be independent of BELLGRAPH, making application to other systems possible. For example, this set of programs could be incorporated into a set of FORTRAN subroutines to provide graphics to FORTRAN users. (Of course, one would lose the memory management and hierarchical picture representation supplied by BELLGRAPH). Table 4.1 gives a list of transfer vectors used by the interface programs to communicate with BELLGRAPH. When used in other applications these pointers are set to other locations external to the simulator packages for use by the calling routine. For example, in a FORTRAN environment using the DEC standard operating system, the labels in Table 4.1 would be declared external globals and removed from the data area. The MACRO-15 assembler would assign its own memory locations for these transfer vectors and the linking loader would supply their values.

The third objective is a provision to test the graphics hardware. That is, hardware check-out should be possible by using a simple procedure such as typing in a few commands. A similar method of debugging new functions of the interface programs is also desirable.

TABLE 4.1

LIST OF EXTERNAL REFERENCES IN BELLGRAPH/McGRAPH INTERFACE PACKAGE

DSPLY (120)* Pointer to a word containing the starting address of the display routine (used during the light pen search).

LFEND (336) Pointer to a end-of-leaf flag. This flag is set to -1 on an end-of-leaf trap interrupt.

AXEDGE (334) Pointer to the X-edge register. This register contains the number of times the X register overflowed or underflowed. On an overflow it is incremented. On an underflow it is decremented.

AYEDGE (335) Pointer to the Y-edge register. On a Y register overflow it is incremented. On a Y register underflow it is decremented. When both the X-edge register and Y-edge register contain zero the beam is on the screen.

AMARGN (407) Pointer to margin. On a carriage return or margin return the X-co-ordinate is reset to this margin value.

AWHERX (230)
AWHERY (231) Pointers to latest co-ordinates of tracking cross. These are updated by G2SIM each time the tracking cross interrupts the PDP-15.

*NOTE: The octal addresses in parentheses give the value of the transfer vector when interfaced to BELLGRAPH.

4.3 System Implementation

Figure 4.1 gives the data flow paths between the various programs comprising the McGRAPH software system. The IOT commands in the executive call the simulator subroutine (G2SIM). These calls set and reset flags maintained in software registers in G2SIM (e.g. the light pen enable flag) or initiate transfers of display processor commands from the executive to the command translator (G2TRAN). The translator sends a list of display commands to the PDP-8 along with some control commands (e.g. start or stop blink).

The time sequence of display commands sent by the executive is mapped into a spatial sequence on the disc. This is best illustrated by example. Consider the following DPU instructions sequentially sent to the command translator:

```
POSITION BEAM AT 500,500
DRAW LINE "L1" WITH DX = 10, DY = 50
SET BLINK ON
DRAW LINE "L2" WITH DX = -10, DY = 50
TURN OFF LIGHT PEN
DRAW LINE "L3" WITH DY = -77
TURN ON LIGHT PEN, TURN OFF BLINK
END OF LEAF TRAP
```

The resulting list of McGRAPH display disc commands would be the following:

```
    ABSOLUTE POINT MOVE 500,500
    VECTOR MOVE DX = 1, DY = 5
    VECTOR MOVE DX = 1, DY = 5
    (Repeated 10 times)
*   START BLINK FILE
    VECTOR MOVE DX = -1, DY = 5
    (Repeated 10 times)
*   IGNORE LIGHT PEN FILE
    VECTOR MOVE DY = -7
    (Repeated 11 times)
*   ALLOW LIGHT PEN HITS
*   STOP BLINK FILE
```

Commands, controlling the display status at a given time in the refresh cycle, (marked here by *) are included as special non-display files on the refresh disc, and have unique filename I.D. numbers. The graphics device monitor program in the PDP-8 continually scans the filename I.D. register and takes the appropriate action on these special control I.D.'s. For example, between START BLINK files and STOP BLINK files the graphics monitor program increases and decreases the beam intensity using one of the D/A's connected to the PDP-8, causing all pictures stored between these two files to blink at approximately 2 Hz.

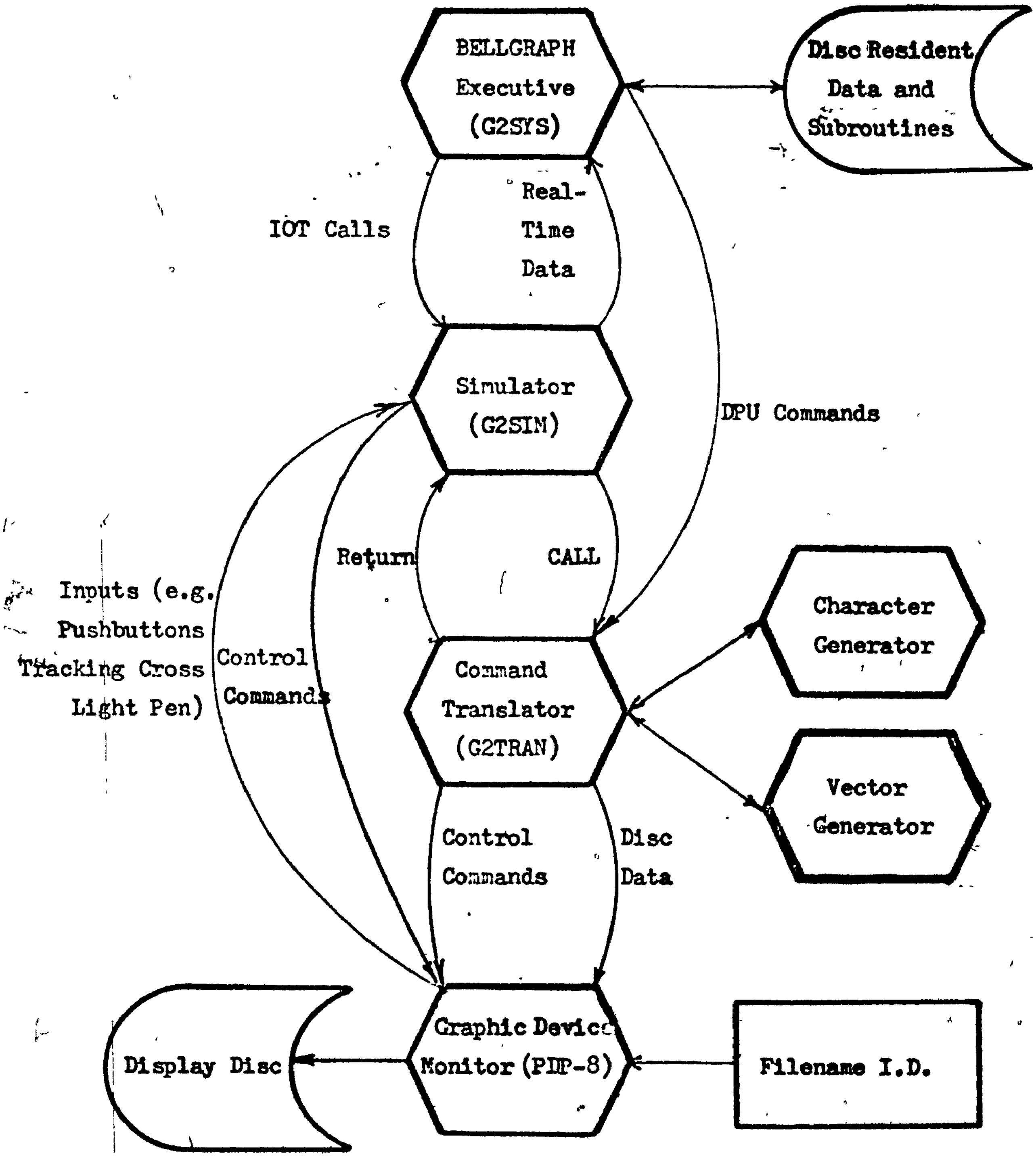


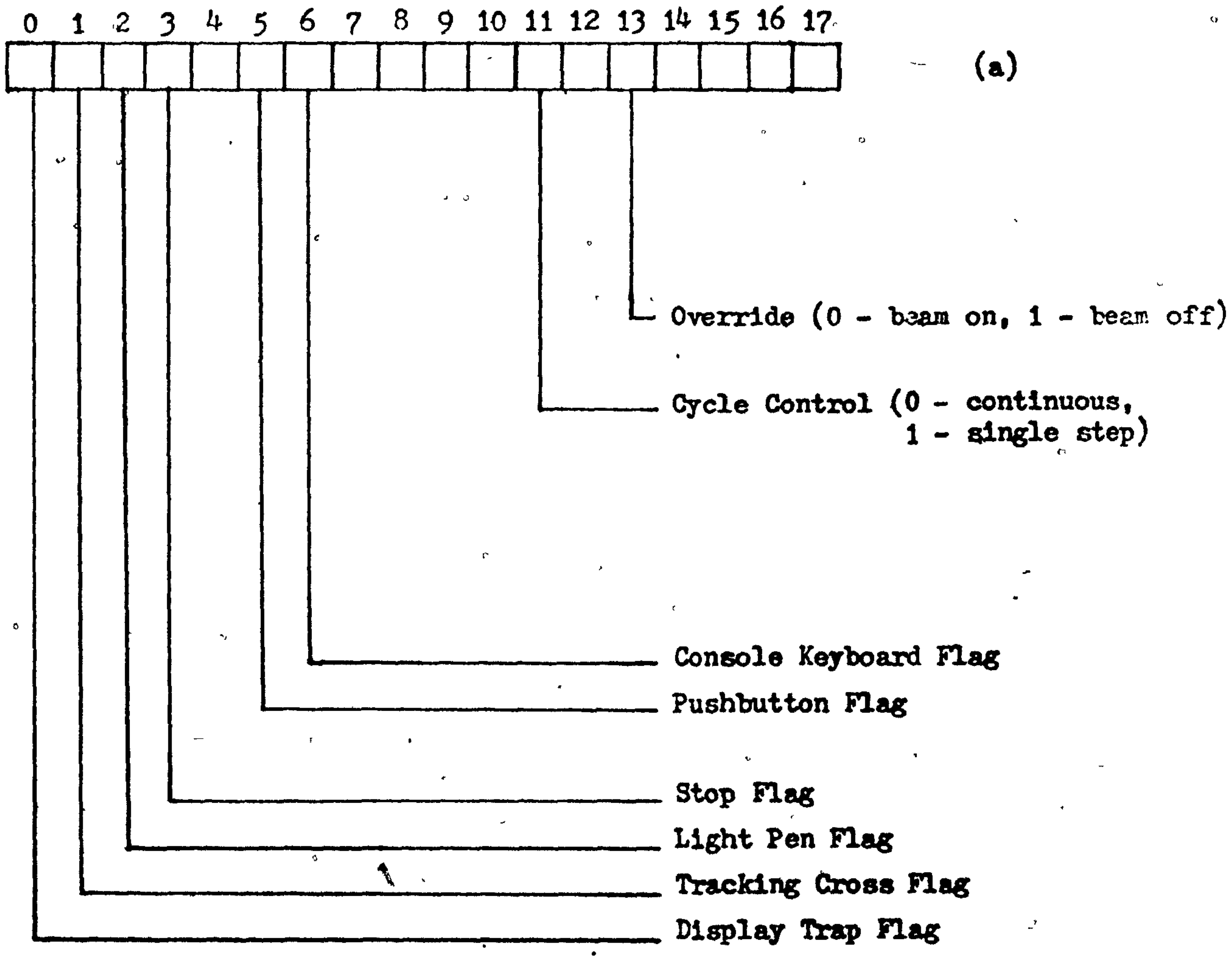
FIGURE 4.1. DATA FLOW IN McGRAPH SOFTWARE SYSTEM.

4.4 GRAPHIC-2 Simulator Program

The GRAPHIC-2 simulator program provides an interface between the BELLGRAPH executive and the McGRAPH display hardware. This program simulates the action of a modified DEC 339 display processor, with console keyboard and eight lighted pushbuttons. A description of the DEC 339 processor is given in the memo entitled "GRAPHIC-2 - Hardware Organization" found in the accompanying BELLGRAPH documentation. All the major data registers in the DEC 339 processor have their software equivalents in the McGRAPH interface programs. Figure 4.2 illustrates the format and label of each of the software registers. Note the minor differences (e.g. the omission of the edges flags in the status registers) between these and their hardware prototypes. The differences are due to changes made to the software system for more efficient operation.

The objective of the simulator program is to minimize programming changes in the BELLGRAPH software. However, there are some instances where direct simulation of the DEC 339 processor results in clumsy, time consuming procedures due to the different refresh policy of McGRAPH. In these cases additional functions were added to the interface package, and the BELLGRAPH programs were correspondingly altered. The following paragraphs describe these additional functions and the BELLGRAPH changes currently implemented. All the changes have reduced the size and complexity of the modified BELLGRAPH programs.

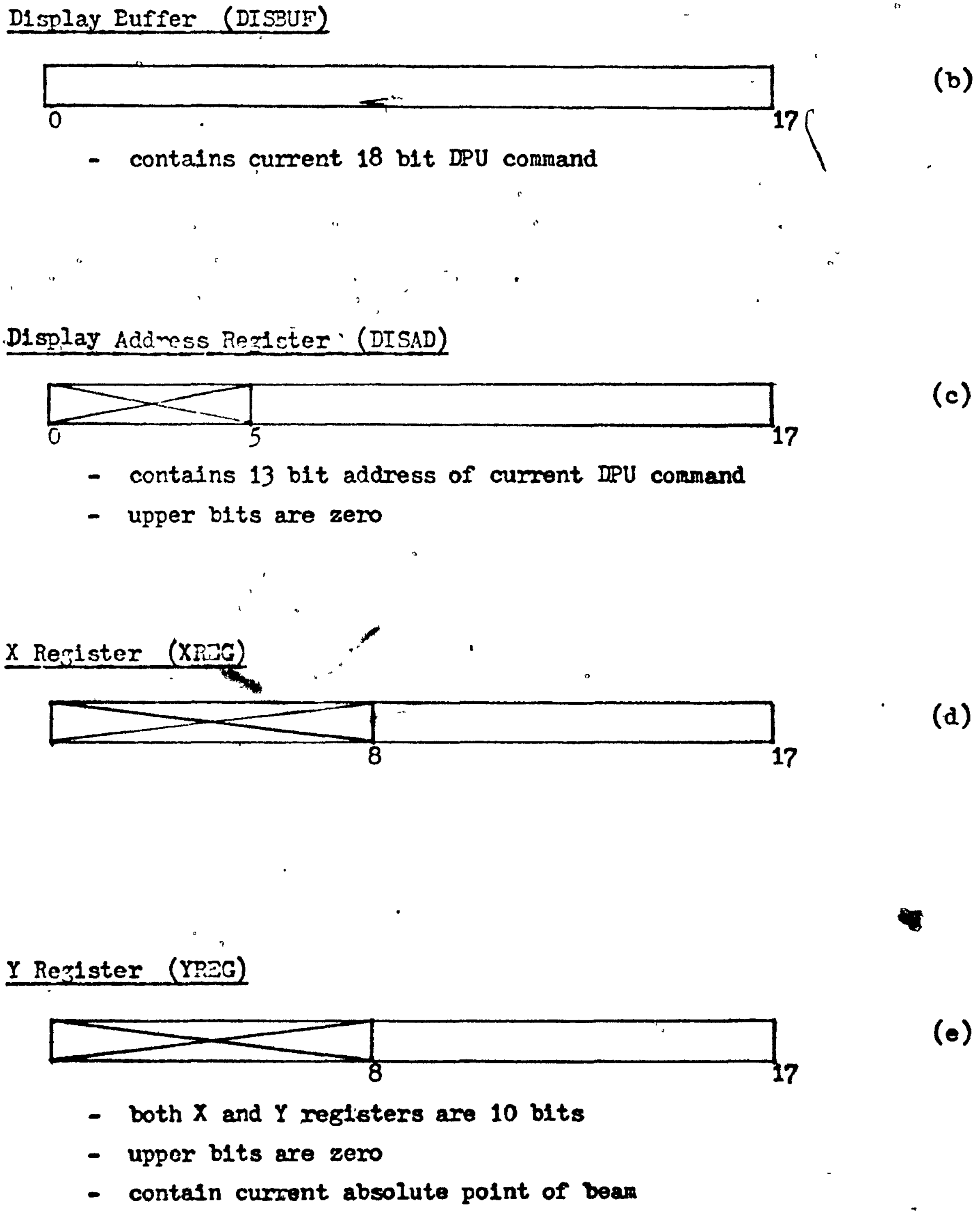
Display Status Register (DSTAT)*



*Symbols in parenthesis refer to labels in the source listings of the simulator program.

FIGURE 4.2. SOFTWARE REGISTERS IN SIMULATED DPU.

(continued)



Display Buffer (DISBUF)

- contains current 18 bit DPU command

Display Address Register (DISAD)

- contains 13 bit address of current DPU command
- upper bits are zero

X Register (XREG)

Y Register (YREG)

- both X and Y registers are 10 bits
- upper bits are zero
- contain current absolute point of beam

FIGURE 4.2. SOFTWARE REGISTERS IN SIMULATED DPU.

(continued)

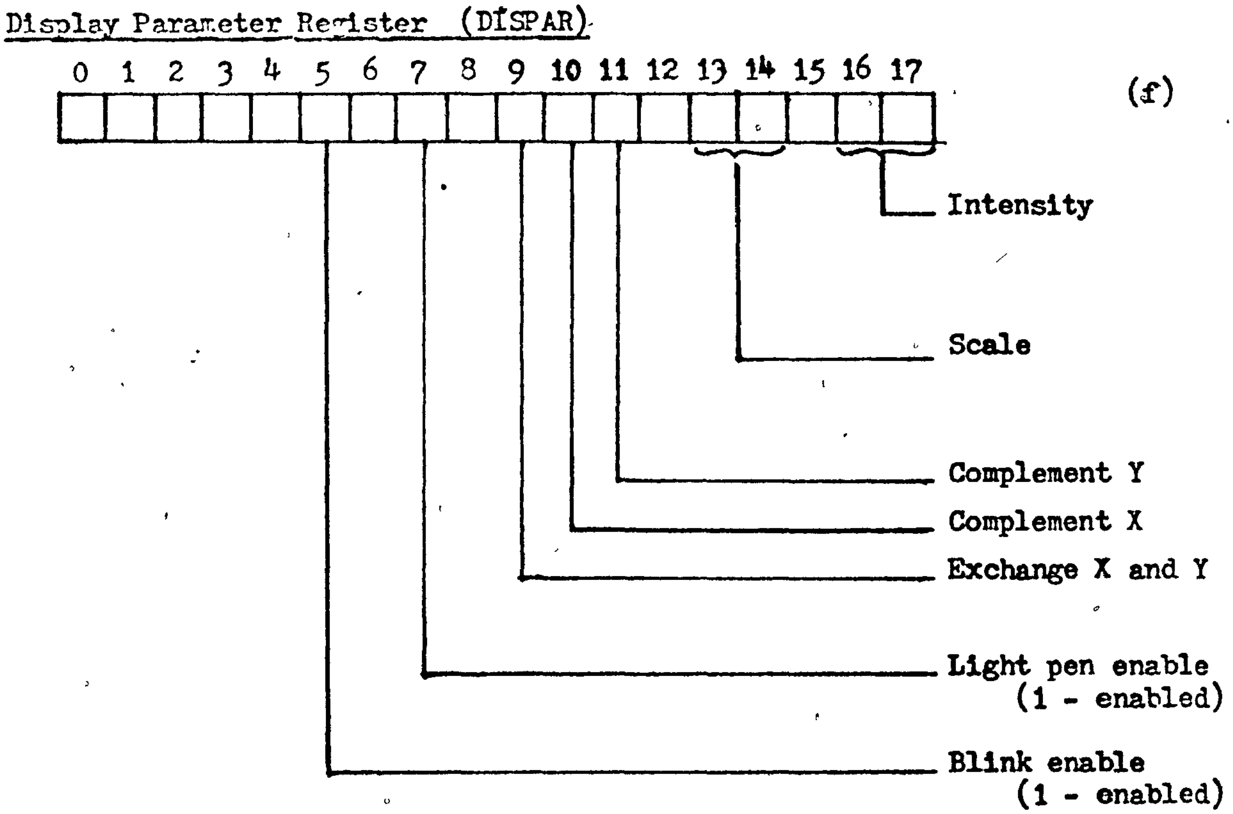


FIGURE 4.2. SOFTWARE REGISTERS IN SIMULATED DPU.

4.4.1 Display Disc Control

A picture is put up on screen by the display subroutine, in the executive, traversing the data structure, and feeding control and DPU commands to the interface programs which translates them and sends them to the display disc via the PDP-8. Each time the graphics monitor receives a block of data corresponding to an instance (one copy of a leaf in the data structure) it adds it to the display list of files on the disc and waits for the next. Some means is required to signal the PDP-8 program that an entirely new picture is required so that it can erase the current one in preparation for the new display. A new simulated IOT command "CCRT" (clear CRT screen) has been defined to perform this function. It is inserted in the existing BELLGRAPH executive subroutine "DCLEAN" used to initialize the display at the beginning of each picture. There is another new simulated IOT "DCRT" used to display leaves. It will be discussed in the section on light pen handling.

4.4.2 Tracking Pattern

BELLGRAPH displays a tracking pattern during execution of the WHERE function for inputing positional information via the light pen. On a light pen interrupt, the handler determines which arm (if any) of the tracking cross was hit and updates its position accordingly. In McGRAPH, light pen tracking is handled solely by the PDP-8. This improves response time, frees the PDP-15 for background computation during tracking, and reduces the size of the "WHERE" subroutine by 70%. A simulated IOT called "TCRT" (display tracking pattern) is issued in "WHERE" for displaying the pattern. CCRT removes it from the screen. The algorithm used for tracking is presented in the section describing the PDP-8 monitor program.

4.4.3 Edge Detection

In the BELLGRAPH system, the DEC 339 interrupts the processor whenever one of the beam position registers (the X or Y register) overflows or underflows. An edge handler routine in BELLGRAPH then takes the appropriate action to prevent the picture from wrapping around on the screen.

In McGRAPH, the translator program G2TRAN updates the software equivalent of these registers after decoding each move and then checks for a ten bit overflow or underflow. On an edge violation, G2TRAN remembers the last X, Y position (saved in the X and Y registers) stops transmission to the PDP-8 but continues to decode commands and maintain the X, Y registers. When the beam returns to the screen an invisible point move from the saved off-screen position to the new on-screen point is sent to the PDP-8 and normal transmission resumes.

4.4.4 Margin Trap Commands

The DPU "TRAP" command of the DEC 339 processor stops the command transfer to the DPU and interrupts the CPU. In BELLGRAPH, this command has many uses. One is to signal margin set and reset commands. That is, on a margin set, a specified X co-ordinate is designated as the starting column for all text. A margin reset trap command resets the X co-ordinate to this value and decrements Y to the next line. In McGRAPH these functions are carried out internally by G2TRAN and do not create interrupts.

4.4.5 The Console Keyboard

The console keyboard is simulated by the teletype on PDP-8. No changes to the BELLGRAPH software were required to support this teletype.

Striking a key on the PDP-8's teletype interrupts (through software in the PDP-8) the PDP-15 causing the interrupt handler to poll the active devices. While polling the graphics devices, entries into G2SIM will be made allowing it to accept the device number and ASCII character from the PDP-8. This data can now be assessed by the main program by issuing the proper IOT's. For example, a "skip on console keyboard flag" IOT transfers to a G2SIM subroutine which checks the software console keyboard flag and increments the return address if set. A "load console keyboard buffer" IOT simulation loads the accumulator with the six bit ASCII character received from the PDP-8's teletype.

4.4.6 The Pushbutton Lights

Up to eight numeric characters, 0 to 7 displayed along the bottom of the screen simulate lighted pushbuttons allowing the user to select them with the light pen. Those remaining blank represent pushbuttons which do not have their back light illuminated. The purpose of these pushbutton lights is to direct the user to those pushbuttons considered active, the other dark ones being ignored by the program. In our simulation of the pushbuttons, we remove from the display those turned off by the program, preventing the user from selecting them. The G2SIM program in the PDP-15, upon receipt of a "turn on" or "turn off" button light IOT, commands the PDP-8 to make the required display adjustment.

4.4.7 The Pushbuttons

Depressing a pushbutton is simulated by a light pen hit on one of the numerics at the lower edge of the screen. The PDP-8 program decodes this hit as a pushbutton selection, interrupts the PDP-15 and passes the pushbutton number in an identical manner as the console keyboard.

4.4.8 The Display Processor

The DEC 339 display processor accepts an IOT command "BEG" which initiates a data break transfer (direct memory access) of sequential display commands to the display processor. A "TRAP" display command stops this transfer and creates a PDP-15 interrupt. This action is simulated by G2SIM. Upon receipt of a "BEG" command, G2SIM translates the DEC 339 display instructions into McGRAPH DPU commands and passes them to the PDP-8. When a trap is decoded, a command is sent to the PDP-8 to transfer the DPU instruction block onto disc, and signal a successful transfer by interrupting the PDP-15. The G2SIM routine then exits and returns to the main program. By returning to the main program immediately, and not waiting for the completion flag from the PDP-8, we can more closely approximate the action of the core refreshed system and significantly improve response time.

The display program in the executive has been rewritten to permit it to retrieve the next leaf from the PDP-15's disc while the PDP-8 is writing its file onto its display disc. This halves the time taken to put up a picture.

4.5 DEC 339 to McGRAPH DPU Command Translation

Due to differences between the command sets of the DEC 339 and McGRAPH processors, the translation of DEC 339 commands to those of McGRAPH is not one-to-one or in fact linear. For example, a pair of DEC 339 commands can specify up to a 10 bit vector move. The translation program (G2TRAN) interpreting this command must first remember if the last move was a vector. If not, a series of commands must be issued setting the McGRAPH processor to vector mode. Then, the vector move is approximated by a series of head to tail bit vector moves. The translator must also segment the McGraph display command list into a set of display files corresponding to each selectable picture primitive displayed on the screen. Each display file is assigned a filename I.D. number for identification on a light pen strike. The following paragraphs describe the translation procedures taken for each of the seven DEC 339 commands used in BELLGRAPH. The format of these commands may be found in Appendix A or in the BELLGRAPH Programmer's Manual.

4.5.1 The Character Command (CHAR)

The DEC 339 CHAR instruction packs two 7 bit ASCII characters into one 18 bit instruction word. The graphic command translator (G2TRAN) in the simulation system extracts these two characters, does a table look up to retrieve a block of relative vector moves for each character and transmits these two blocks to the PDP-8. Note the characters are not affected by the symmetry adjustments since the vector decode step is bypassed. This feature could be included at the cost of slower transmission.

The 64 character set given in Appendix A is currently supported. Lower case alphabetics are currently displayed as upper case characters. All other characters not in this set are ignored. New characters are easily installed by inserting a block of vector move commands into the character look up table.

The size of the alphabetics is 6 x 7 addressable points which include one move right to provide intra-letter spacing. The numbers and other special characters are 8 x 10 which includes a right hand side spacing of two. The line feed character decrements the Y position by 15_8. The carriage return character resets the X position to the margin value set by the last margin trap (see TRAP command).

4.5.2 The Parameter Command (PARAM)

The parameter command sets display parameters for subsequent vector moves. These parameters are:

1. Blink
2. Light Pen Enable/Disable
3. Symmetry (complement X, complement Y, exchange X and Y)
4. Scale
5. Intensity

If the blink is set all subsequent pictures displayed on the screen blink at approximately 2 Hz. This is performed by sending the PDP-8 monitor a "blink-on" instruction. Similarly this parameter can be reset resulting in a "blink-off" PDP-8 monitor instruction. The blink bit in the software parameter word is updated to reflect the current status.

The Light Pen enable sets the appropriate flag in the display status word, and affects the filename I.D. numbers in the HEADER commands of the output instruction stream.

The symmetry bits determine whether succeeding X and Y vector move commands will be complemented and/or exchanged before being decoded and displayed. When both complementing and exchanging are to occur complementing is done first. These transformations are simulated by software. The symmetry bits set three software switches used by the vector command decoding subroutine.

Both the DEC 339 and McGRAPH display processors permit four hardware scale settings, normal (1024 x 1024 addressible points), twice normal size (512 x 512 addressible points), four times (256 x 256), or eight times normal size (128 x 128). That is, given a vector command to draw a line ten units in the X direction, in scale "0" (normal size) a horizontal line ten units long would appear on the screen. If the hardware scaling was set to "3" (eight times) however, the same command would generate a line eighty units long. The two scale bits are extracted from the parameter word by the command decoding subroutine and inserted directly in a scaling sequence of DPU commands for McGRAPH. The scale then remains at this value until the next scale adjusting parameter command, with one exception. The DEC 339 DPU absolute point moves (x - Y command) are not affected by scaling. To simulate this, the current scale setting is saved, and a scale "0" command sequence is inserted before each absolute point move. The scale is reset after the absolute point move sequence has been generated.

The beam intensity is adjusted in a similar manner to the scale. The two intensity bits in the parameter word are used to modulate the high order bits of McGRAPH's four bit intensity register. These remain unaltered until the next intensity setting parameter word.

4.5.3 The Absolute Point Move Command (X - Y)

The X-Y command gives a ten bit co-ordinate position of the beam in either X or Y. Using two of these commands, the beam can be placed at any point in the 1024 x 1024 addressible point array regardless of current scaling. Since a pair of X-Y commands generates an absolute move independent of all preceding commands, they usually precede a picture instance drawing list. By making all remaining moves in the list relative vector moves (invisible or visible), the instance is easily moved on the screen by changing only the first two X-Y commands. For example, the display routine in the BELLGRAPH software package issues one and only one X-Y command pair for each leaf. These commands are stuffed with the accumulated co-ordinates of each branch block in the path to the leaf.

To simulate the action of a X-Y command G2TRAN generates the following DPU instructions:

SET SCALE TO "0"

HEADER WITH FILENAME I.D. (if X move is visible)

CLEAR AND ADD 10 BIT (visible/invisible) X POINT MOVE

HEADER WITH FILENAME I.D. (if Y move is visible)

CLEAR AND ADD 10 BIT (visible/invisible) Y POINT MOVE

RESET SCALE TO "X"

For the sake of clarity we have omitted the mode changing commands required by the McGRAPH DPU. See Fabi[30] for complete details concerning the display processor. The intensity of each move (invisible or visible) is copied directly from the original X-Y command. The example sequence above is generated from a pair of adjacent X-Y commands. If there is only one X-Y command followed by another type of command there is only one absolute point move in the generated sequence. Scale "X" in the above example is the scale setting previous to the X-Y commands. In an attempt to optimize the generated code, G2TRAN looks ahead for X-Y command pairs before resetting the scale to "X". In practice, a PARAM command which sets the scale to a new value normally follows an X-Y pair. Therefore, further optimization could be done by looking ahead for a scale setting PARAM command. This is not implemented at present.

The HEADER command preceding each visible move is used to identify objects on a light pen strike. The 12 bit filename I.D. in each HEADER instruction is the only real-time data available on a light pen hit. (See Chapter III for a discussion of McGRAPH's hardware). All other data pertaining to each instance is generated from this twelve bit number. Therefore, each separate display entity must have a unique filename I.D.. The coding of this I.D. number is discussed in Section 4.6.

4.5.4 The Long Vector Move Command (VECT)

The long vector command contains a ten bit relative move, a bit specifying X or Y and two bits giving the following control instructions:

1. Load holding register only.
2. Load holding register, draw invisible, clear registers.
3. Load holding register, draw visible, clear registers.
4. Load holding register, draw invisible (except for end point), clear registers.

There are two holding registers (X and Y) in the DEC 339 display used to contain the relative moves of vectors. These registers are simulated in the command decoder routine. The list of DPU commands generated from a long vector command depends upon the control specified. A load holding register instruction does just that, and generates no commands at all. The draw invisible and draw visible, except for end point commands produce two ten bit relative point moves. A typical output sequence is given below:

(ADD/SUBTRACT) 10 BIT X POINT MOVE (invisible)
HEADER WITH FILENAME I.D. (if Y visible)
(ADD/SUBTRACT) 10 BIT Y POINT MOVE (visible/invisible)

The contents of the holding registers are inserted into the relative point moves. If the contents of the Y holding register is zero, the X move takes on the intensity attribute of the VECT command and the Y move is omitted. If the contents of the X holding register is zero the X move is omitted.

Visible vector moves are approximated by a list of vector commands.[30] McGRAPH vector commands provide 3 bit (7 units) moves in X, Y or X and Y. In other words, there are twenty discrete slopes possible in any octant (Table 4.2). Lines having slopes other than these, are approximated using the algorithm presented in Figures 4.3 and 4.4. This algorithm is quite successful especially when the vector segments used in the approximation are small relative to the vectors drawn.

TABLE 4.2

POSSIBLE SLOPES OF McGRAPH DPU VECTORS IN FIRST OCTANT

X, Y > 0

X > Y

X Y	0	1	2	3	4	5	6	7
0		HORIZONTAL LINES						
1	VERTICAL LINES	1/1	2/1	3/1	4/1	5/1	6/1	7/1
2				3/2		5/2		7/2
3					4/3	5/3		7/3
4						5/4		7/4
5							6/5	7/5
6								6/7
7								

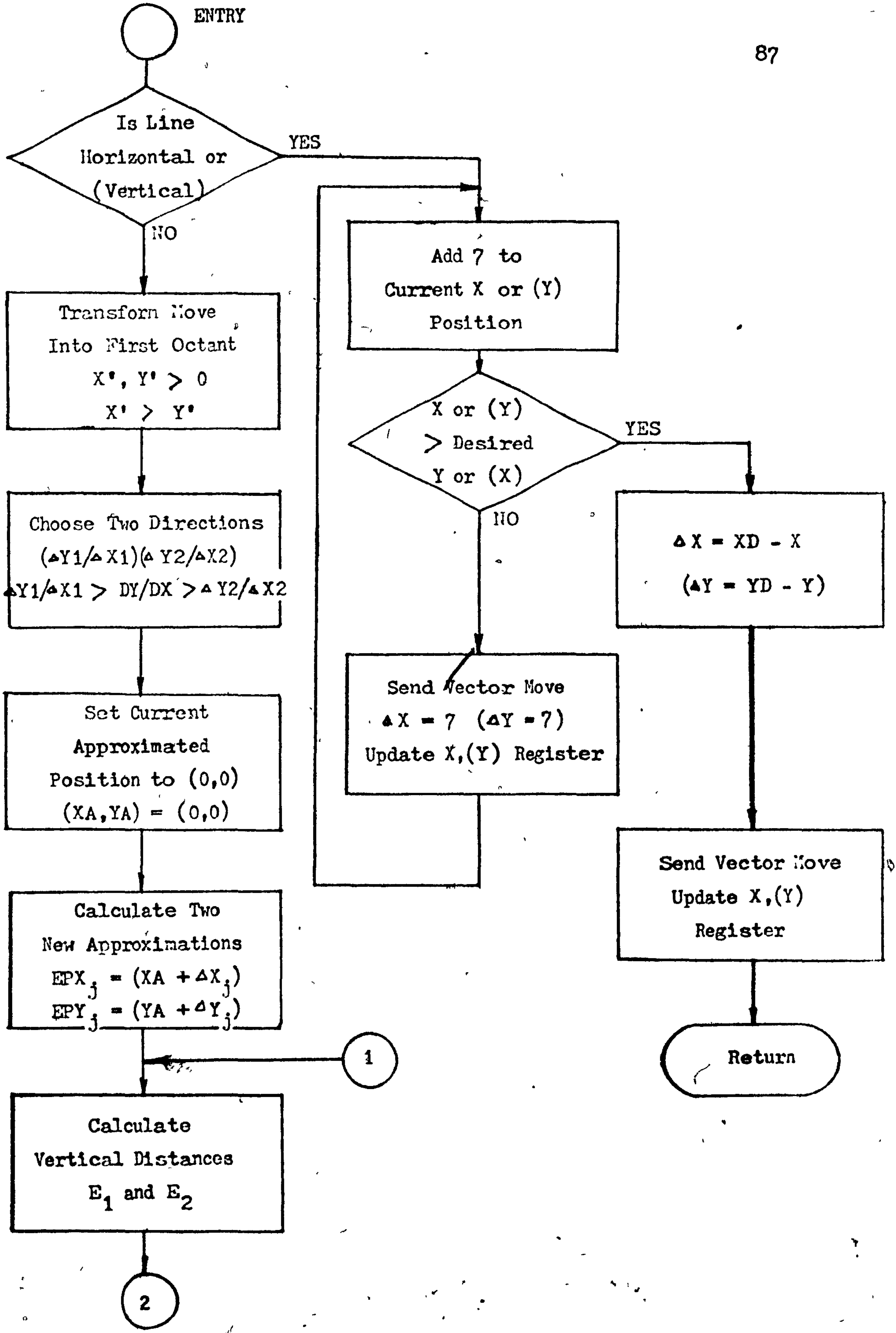

ENTRY
Is Line Horizontal or (Vertical)
YES
NO
Transform Move Into First Octant
X', Y' > 0
X' > Y'
Choose Two Directions
(ΔY1/ΔX1)(ΔY2/ΔX2)
ΔY1/ΔX1 > DY/DX > ΔY2/ΔX2
Set Current Approximated Position to (0,0)
(XA,YA) = (0,0)
Calculate Two New Approximations
EPX_j = (XA + ΔX_j)
EPY_j = (YA + ΔY_j)
1
Calculate Vertical Distances E1 and E2
2
Add 7 to Current X or (Y) Position
X or (Y) > Desired Y or (X)
YES
NO
Send Vector Move
ΔX = 7 (ΔY = 7)
Update X,(Y) Register
ΔX = XD - X
(ΔY = YD - Y)
Send Vector Move
Update X,(Y) Register
Return

FIGURE 4.3. VECTOR APPROXIMATION ALGORITHM.

(continued)

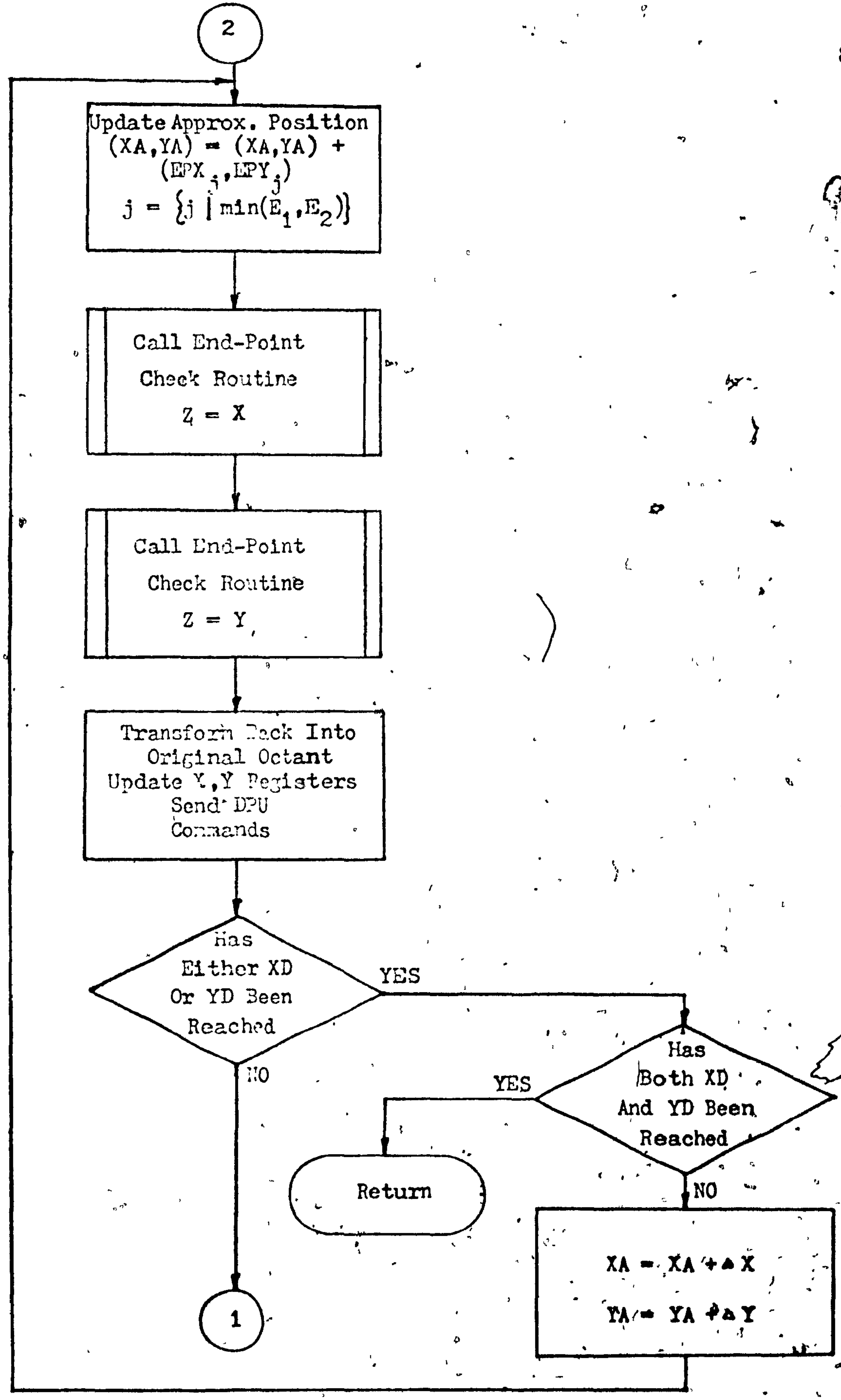
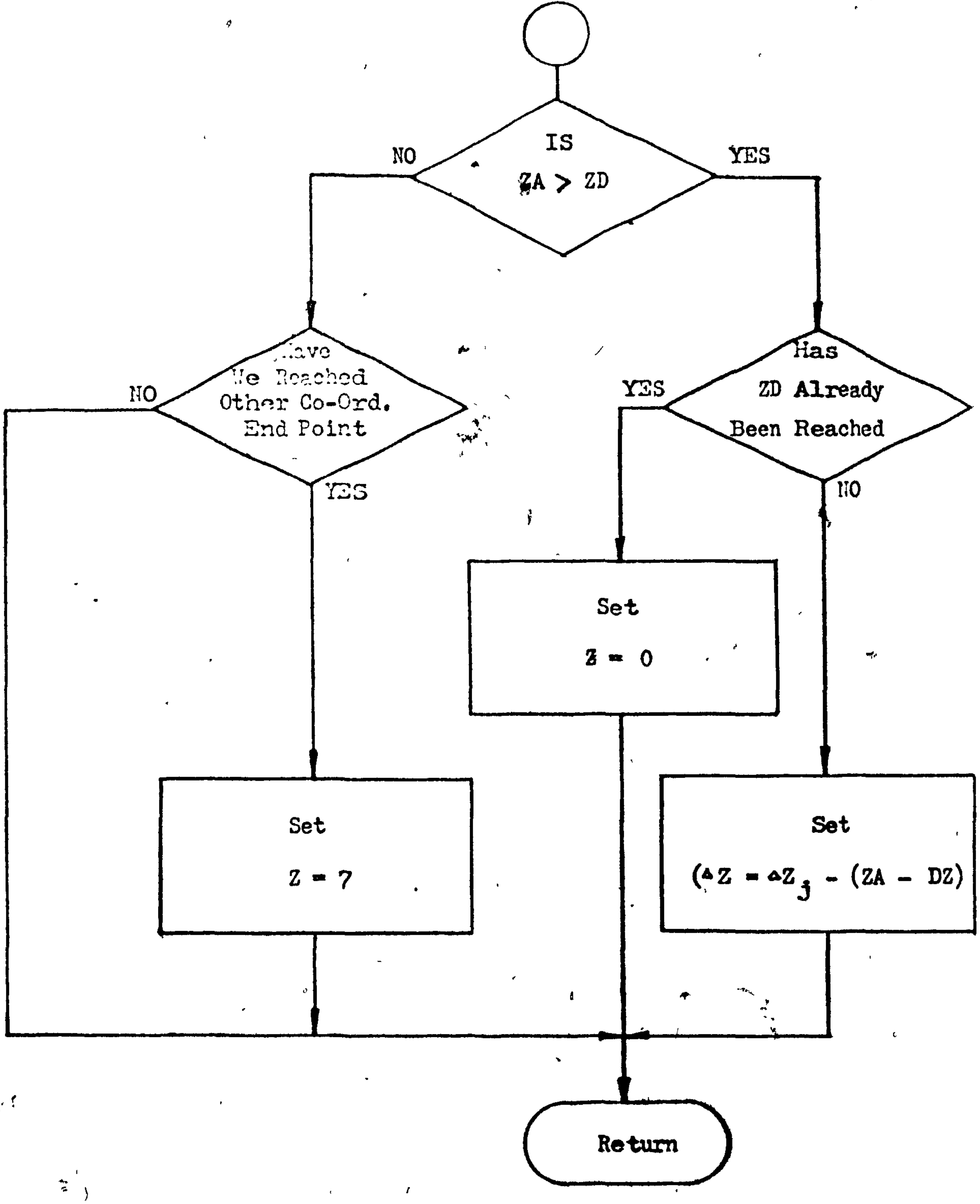


FIGURE 4.3. VECTOR APPROXIMATION ALGORITHM (continued)



End-Point Check Routine

FIGURE 4.3. VECTOR APPROXIMATION ALGORITHM.

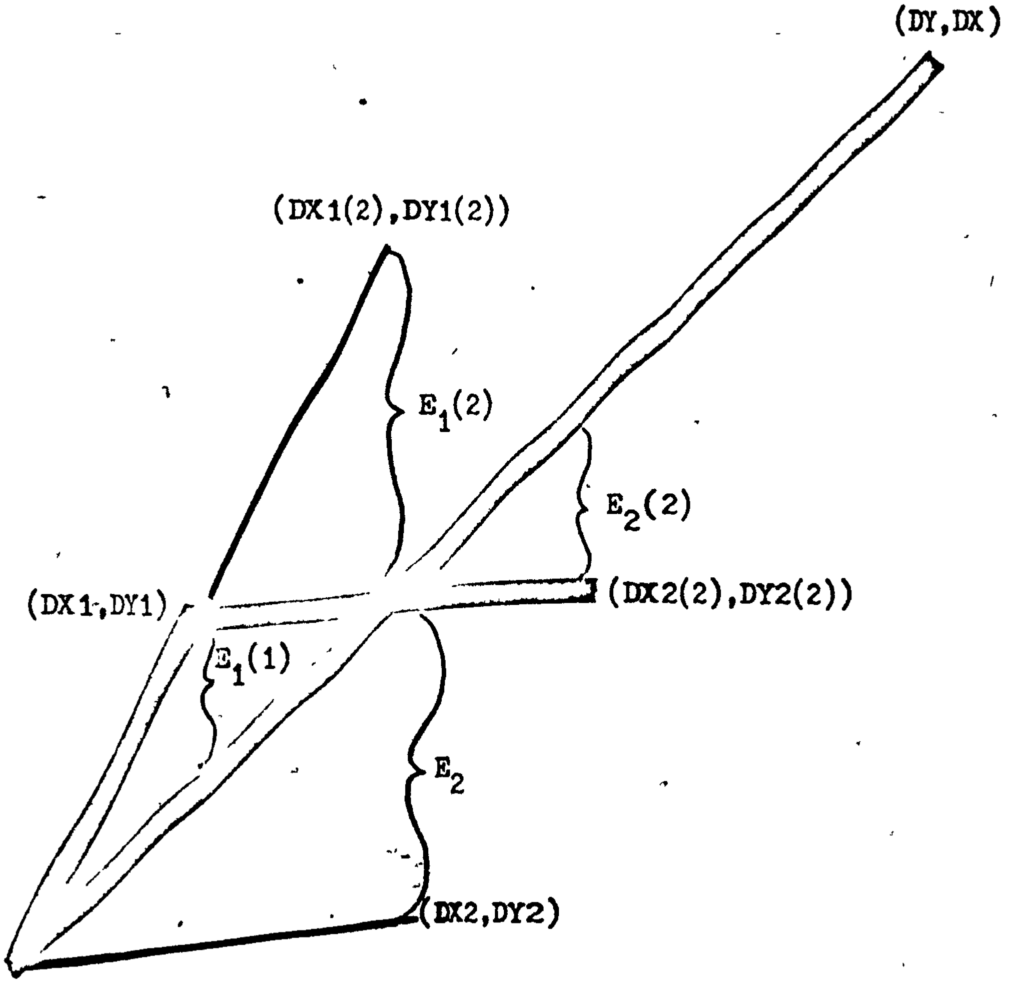


FIGURE 4.4. APPROXIMATION OF A LINE USING McGRAPH VECTORS.

A HEADER command precedes the output vector command list if:

1. The preceding move command was an invisible vector move with invisible end point.
2. The preceding move command was an invisible X-Y.
3. There has been a change in the light pen status (enable/disable) since the last move.

4.5.5 The Short Vector Move Command (SVEC)

This command is similar to the long vector command except that five bit moves for both X and Y are packed in one command. Both the X and Y holding registers are loaded with this command and the execution of th9 move is identical to that of the long vector command.

4.5.6 Edge Detection Program

In BELLGRAPH, the DEC 339 DPU provides an interrupt and raises one of four flags in the case when a vector command moves the beam off screen. The program can then take corrective action such as turning off the intensity to prevent the picture from wrapping around on the opposite edge. This feature must be simulated on McGRAPH. The simulation program does this by keeping the 10 bit co-ordinate position of the beam in two software registers (XREG, YREG). Before sending a vector or point move to the PDP-8 these registers are updated and checked for overflow or underflow. If one of these conditions occur, subsequent commands are not sent until the X and Y registers return within range. Then an invisible relative point move is made from the last point displayed on the screen to the current re-entry point. By not sending the off-screen vectors, we prevent wrap around, and save space on the display disc. By performing all the edge violation procedures within the simulator, we gain speed, since we no longer require interrupts for edge detection, and we gain space in managed memory, since we do not need edge handlers in the BELLGRAPH executive.

4.5.7 The Control Command (CNTRL)

The control command is used to:

1. Stop the transfer of DPU commands.
2. Override the beam intensity (turn it off).
3. Set the vector mode to broken (dashed) lines.

The stop command raises the stop flag in the software display status register, and causes an exit from the interface programs. The beam override is effected by turning off the subroutine (CSEND) used to send commands to the PDP-8. Dashed line control sets a switch in the vector generator subroutine causing it to generate alternate invisible vector segments.

4.5.8 The Trap Command (TRAP)

The trap command signals the end of a DMA transfer cycle by stopping it, raising a trap flag and initiating an interrupt. In McGRAPH the trap is used as a:

1. End of a leaf flag.
2. Margin set flag.
3. Margin reset flag.

In the original BELLGRAPH software, the trap was used for a repeat leaf flag and real-time subroutine entry as well. These functions are not supported in the current McGRAPH version. The interested reader is referred to the BELLGRAPH Programmer's Manual for details of these two features.

The command decoder (G2TRAN) on an end of leaf command returns control to the main program after instructing the PDP-8 to transfer the previous block of DPU commands to the display disc. The PDP-15 is later interrupted by the PDP-8 upon successful completion of this transfer.

The margin set trap command is handled internally by the simulation program and thereby does not require a handler in the main program. This command sets a value for the text margin. In other words, on a receipt of a carriage return character or margin reset trap command, the X co-ordinate is set to the value set by this command.

4.6 The Light Pen Handling Routine

The DEC 339 display processor provides the following real-time data on a light pen strike:

1. Address of current DPU command.
2. X and Y co-ordinates of the beam.
3. Current display parameters (intensity, blink, etc.)

On a light pen hit the display subroutine in the original BELLGRAPH system is interrupted from its traverse of the data graph, so that the status of this program (e.g. the pushdown stacks) provides additional information concerning the picture instance (e.g. the path taken through the graph).

In McGRAPH, the display subroutine executes only once, and then waits for input interrupts. All real-time information on a light pen hit must be derived from the 12 bit filename I.D. supplied by the DPU, requiring each separate instance to have a unique filename I.D. so that it can be identified on a light pen hit. We define an instance as a section of text or a contiguous string of visible vectors in which there areno light pen status changes, invisible vector moves, or absolute point positions. Every such instance is assigned a unique filename I.D. number and a file on the display disc. In other words, an instance is a connected figure separated from all others by space or light pen status.

Sufficient resolution of light pen strikes is expected using this definition to identify separate instances, for normally, the light pen is used to select only picture parts. If finer positional resolution is required, the tracking cross is used.

The real-time data associated with a light pen strike can be grouped into categories:

1. That needed by all users of the simulation program.
2. That specific to the BELLGRAPH software package.

To make the graphic device handler package have universal application for all users, it is as independent as possible from the BELLGRAPH software. In an effort to accomplish this objective, the display and light pen handling is performed in two modes. One is used by BELLGRAPH, the other is reserved for potentially different applications. The mode is selected by choosing one set of IOT's to display non-structured pictures or messages, and another for the display of leaves in the data graph (See the list of IOT's in Appendix A). The "BEG" instruction is used to display non-structured pictures or messages. In this mode, unique filename I.D. numbers, called message numbers, are generated for each HEADER in the output data stream while the light pen enable flag is "ENABLED". If the light pen is disabled a special filename I.D. number is used.. (See filename I.D. number allotments in Appendix A). The PDP-8 does not interrupt the PDP-15 on hits of these disabled files. Sixteen separate light pen sensitive messages (or pictures) are permitted. For each of these the following data is saved:

1. Address of the command generating the message number.
2. The current X and Y co-ordinates.
3. The current display parameters.

On a light pen hit, the data associated with the selected message is loaded into the display address register, the X and Y registers, and the parameter register respectively. Subsequent IOT commands can then interrogate these registers and extract the required data. If the display exceeds sixteen light pen sensitive messages or pictures the following error message is printed:

```
TOO MANY FILES
MMMMMM
SSSSSS
LLLLLL
```

where,

MMMMMM is the message number,

SSSSSS is the subleaf number,

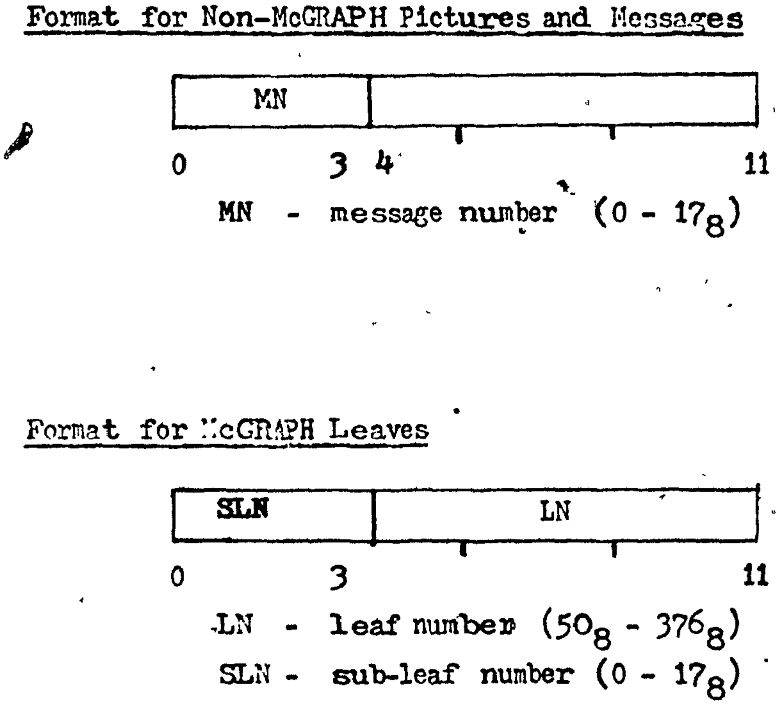
LLLLLL is the leaf number.

The display subroutine in the modified BELLGRAPH executive, and only this subroutine, uses the special IOT instruction "DCRT" to display leaves in the data structure. In this mode, a leaf counter is incremented on each occurrence of an X-Y command pair. Remember a maximum of two of these commands occur in the first two words of each leaf. If the light pen is disabled at this time, the non-sensitive filename-I.D. number is placed in the DPU HEADER instructions. If the light pen is enabled, the leaf number is inserted in bits 4 to 11 of the filename I.D. (See Figure 4.5). If the leaf number exceeds 367_8 the above error message is printed and the program halts.

On each subsequent occurrence of a HEADER command in the output list, while the light pen is enabled, a four bit sub-leaf number counter is incremented and inserted into bits of 0-3 of filename I.D. along with the leaf number. If the sub-leaf counter overflows, the above error message is printed and the program halts.

This filename I.D. coding technique enables one to display up to 320_8 instances of leaves (numbers 0-40 are reserved for the system) each having up to 20_8 individual parts. (Separated by invisible moves). Although this technique lowers the upper limit on the number of displayed leaves, and imposes a potential maximum on the number picture parts per leaf, it does provide the simulation program more information on light pen hits, significantly reducing the time taken to extract the real-time data needed. The coding procedure appears justified on the grounds that overflow of the leaf or subleaf counters is not expected during normal use.

If the filename I.D. received from the PDP-8 on a light pen interrupt indicates a leaf, the simulation program searches the data structure for this instance. The search is performed by turning off all



Format for Non-McGRAPH Pictures and Messages

MN - message number $(0 - 17_8)$

Format for McGRAPH Leaves

LN - leaf number $(50_8 - 376_8)$
SLN - sub-leaf number $(0 - 17_8)$

FIGURE 4.5. FORMAT OF FILENAME I.D. WORD.

command decoding, save that of the X-Y instruction, and then entering the display subroutine. While in this search mode, the simulator receives leaf instances from the display subroutine, as during a display cycle, and counts them. When a match occurs between this count and the leaf number of the instance struck, a second level search is initiated. The simulator is now allowed to interpret the commands although transmission to the PDP-8 is inhibited, and a match is sought on the sub-leaf numbers. When a match is found, both the display subroutine and simulated display processor are in the desired state and control is returned to the main program. All real-time data used by the main programs on a light pen hit are now available. The search on light pen hits does degrade response time although it is not expected to be objectionable (less than 1 second).

4.7 The PDP-8 Graphics Monitor

4.7.1 Introduction

All communication between graphics programs residing in the PDP-15 and the graphics hardware devices connected to, or simulated on, the PDP-8 is handled by the set of programs called the graphics monitor. The graphics monitor is completely independent of the BELLGRAPH programs in the PDP-15. It is equally useful to the user wanting to draw graphics directly with FORTRAN, or it can be used in a stand-alone mode as well. Typing a CNTRL T character switches the data input/output device from the PDP-15 to the PDP-8's teletypewriter. In this mode all commands normally originating from the PDP-15 can be typed in as four digit octal numbers, and all responses to the PDP-15 are diverted to the PDP-8 teletypewriter and are printed as four digit octal numbers. This PDP-8 stand alone mode enables one to investigate hardware faults locally without tying up the PDP-15. It also helps when adding and debugging new features to the PDP-8 monitor program itself.

The PDP-8 graphics monitor performs the following functions on demand from the PDP-15 (or teletypewriter):

1. Receives display data and writes them out onto the display disc.

2. Displays up to eight numbers on command along the bottom of the screen as a simulation of back lighted pushbuttons.

3. Outputs a pushbutton flag and number when one of these simulated pushbuttons have been selected by the light pen.

4. Displays a tracking cross which can be dragged around using the light pen. The current co-ordinates of this tracking cross are continually output after every update.

5. Accepts a character from the typewriter, translates it into six bit ASCII code, and outputs the result.

6. Accepts light pen hits and outputs the light pen flag and current filename I.D. (if they are not on the tracking cross or pushbuttons).

7. Blinks specified files at approximately 2 Hz.

A complete user's guide for the Graphics Monitor is given in the back of this section.

4.7.2 The Tracking Pattern Algorithm

Moving the tracking cross is a simple function best done by the graphics monitor. By removing the tracking cross procedures from the BELLGRAPH software, we have significantly decreased the overhead load on the PDP-15 during tracking and increased available core space. This has required a complete rewrite of the GRIN subroutine "WHERE" but the result is a simpler smaller routine. A similar revision to the subroutine "DRAW" is required. This also will decrease its size and complexity.

The tracking pattern along with its algorithm is given in Figure 4.6. It consists of eight display files. The four corners of the

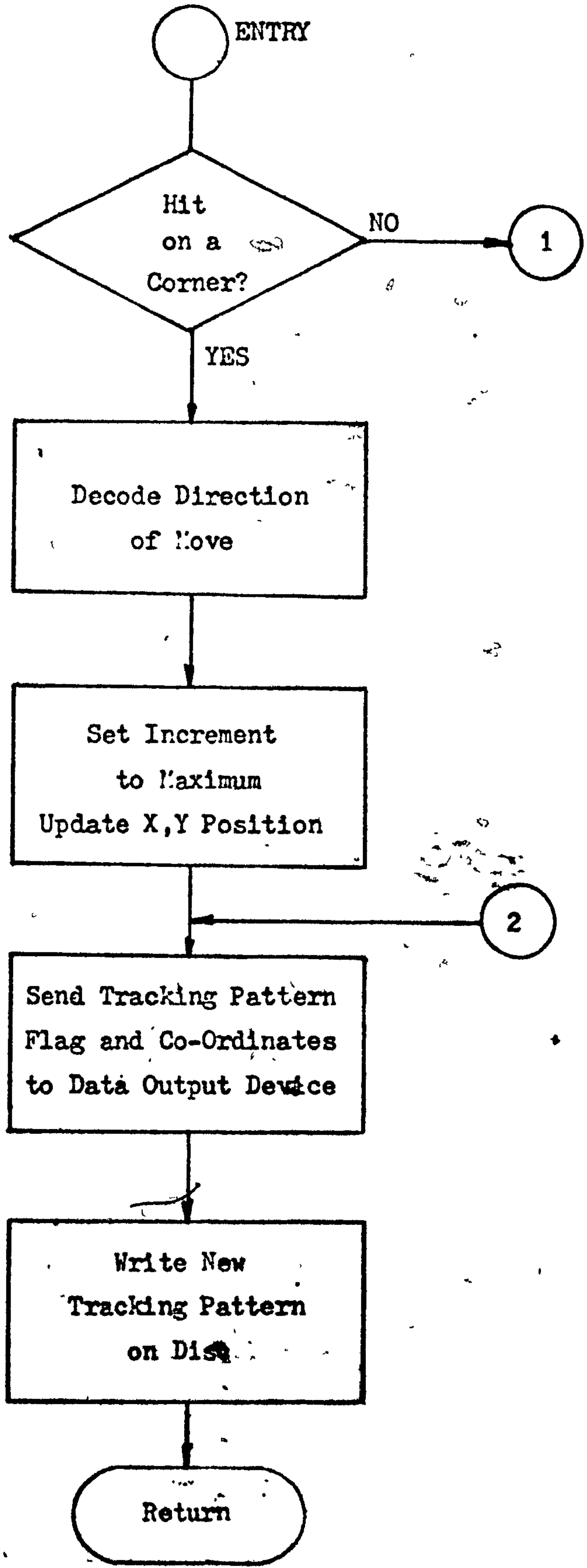

ENTRY
Hit on a Corner?
NO
1
YES
Decode Direction of Move
Set Increment to Maximum Update X,Y Position
2
Send Tracking Pattern Flag and Co-Ordinates to Data Output Device
Write New Tracking Pattern on Disk
Return

FIGURE 4.6. TRACKING PATTERN ALGORITHM

(continued)

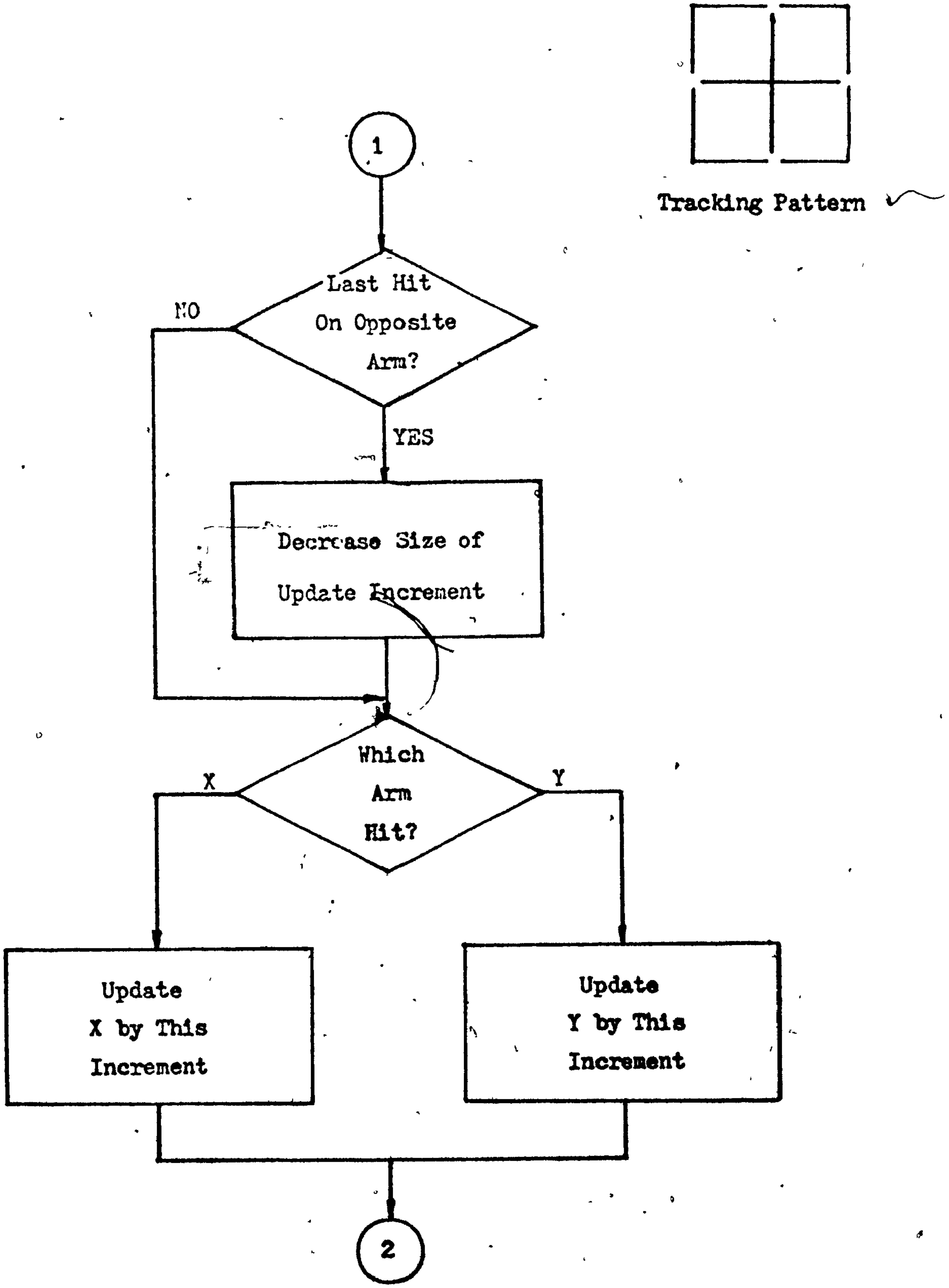

Tracking Pattern
1
Last Hit On Opposite Arm?
NO
YES
Decrease Size of Update Increment
Which Arm Hit?
X
Y
Update X by This Increment
Update Y by This Increment
2

FIGURE 4.6. - TRACKING PATTERN ALGORITHM.

box are used for gross moves so that the pattern can be quickly dragged across the screen, and the inner cross hairs enable fine adjustment of the pattern's position. Any point in the 1024 x 1024 addressible point array can be selected using the tracking cross.

The tracking cross is used by first moving it such that the desired point is within its box. Holding the light pen on a horizontal cross hair at the desired X co-ordinate will move the tracking cross to this location. The vertical cross hair now intersects the desired point. By holding the light pen at the desired location, the tracking cross will zero in on this point.

4.7.3 PDP-8 Graphics Monitor Users Manual

This section describes the modes of operation of the Graphics Monitor and its instruction set. The following discussion is suggested for the reader wishing to use this software package in controlling the graphics devices, both real and simulated.

The graphics monitor alters its mode of operation upon receipt of the following mode control commands.

↑T* CONTROL T substitutes the teletype for the PDP-15 as a data input/output device. All data normally received from the PDP-15 is now expected from the teletype's keyboard.

↑P CONTROL P restores the PDP-15 as the data input/output device.

↑K CONTROL K indicates that the following text from the teletype keyboard is to be regarded as coming from the simulated console keyboard. Another ↑K command resets this mode.

↑D CONTROL D followed by a two digit octal number allows the user to change the display disc's track. NOTE: The selector switch on the graphics unit must also be switched to the new track.

Besides accepting mode control commands the graphics monitor can also perform certain functions on demand from the data input device. The function commands are twelve bit negative number. The input data (DPU commands) are eight bit numbers right justified in the PDP's word. Therefore, negative function commands are easily separated from the data in the input stream.

These function commands are used to route the data originating from the PDP-15 (or teletype) to one of the following output devices:

*Note: ↑ indicates the CNTRL key is depressed at the same time as the T, P, K, and D keys are struck.

1. Display.
2. Tracking cross (output of initial position).
3. Blink control.
4. Pushbutton lights (selects the displayed bushbuttons).

Table 4.3 gives a list of the function commands and their action.

Devices sending data from the PDP-8 are identified by preceding the data with a status word. This status word contains one bit set corresponding to one of the following input devices:

1. Display trap (end of disc write).
2. Tracking pattern.
3. Light pen.
4. Pushbuttons.
5. Console keyboard.

Figure 4.7 gives the format of the status and the data following each one.

These function commands and input status words provide a convenient method to any user of accessing the graphics devices. The graphics monitor program allows further additions to the function command repertoire and supported input devices. For example, McGRAPH joystick is not currently supported, but could be added.

TABLE 4.3

A LIST OF THE FUNCTIONAL COMMANDS

1. RESET THE DISPLAY (-1):

   - Writes the next display at the beginning of the track thereby erasing the previous picture.
   - Resets the core buffer.

2. END OF DISPLAY (-2):

   - Accepts one more word from the input device (trap address when used with BELLGRAPH) then appends the current block of display data to the display list on the disc.
   - Transmits the last word accepted back to the data input device as acknowledgment of successful completion.

3. TRACKING PATTERN ERASES (-3):

   - Remove tracking pattern from the display.

4. START DISPLAY DATA BUFFER (-4):

   - Resets the core buffer in preparation for graphics data from the data input device.

5. ENABLE BLINKING (-5):

   - Inserts a "start blink" header into the data list.
   - The monitor continuously scans the filename register for the start blink I.D. and modulates the intensity accordingly.

6. DISABLE BLINKING (-6):

   - Inserts a "stop blink" header into the data list.
   - The monitor scanning the filename register restores normal intensity on this I.D.

7. DISPLAY TRACKING PATTERN (-7):

   - Receives two words from the PDP-15 to define the starting position of the tracking pattern.
   - The tracking pattern is then displayed.
   - Each time its position is updated two words are sent indicating its current position.

8. RECEIVE PUSHBUTTON LIGHT WORD (-10_8):

   - Receives one word giving the pushbuttons to be displayed.
   - The position of set bits in this word corresponds to the pushbutton numbers displayed. e.g., Bit #2 set indicates pushbutton #2 is to be displayed.

Status Word	Device	Data
Bit #		
0	Display Trap	Word following end of display command
1	Tracking Cross	Current X, Y position
2	Light Pen	Filename of instance hit
3		
4		
5	Pushbuttons	Bit # of bit set = pushbutton #
6	Console Keyboard	6 bit ASCII character
7		
8		
9		
10		
11		

FIGURE 4.7. FORMAT OF DATA INPUT STATUS WORD.

CHAPTER V

THE GRAPHICS LANGUAGE ASSEMBLY SYSTEM

5.1 Introduction

A high level graphical programming language called "GRIN" is available to the applications programmer. Using this language, he can prepare programs which define the procedures and algorithms available to the user of the graphics terminal. For example, a typical program might display a menu of functions available to the user. Upon picking one with the light pen, the program transfers control to the corresponding function subroutine within it. It is important to keep clear the distinction between the applications programmer and the user. The applications programmer prepares a program off-line which can be used to solve a class of problems. The individual users then run this program, and supply their unique data using the graphics devices, in an effort to solve their particular problem.

GRIN programs are prepared off-line on cards and are submitted to an IBM 360/75 for assembly into machine language. The resulting object module is then read into the PIP-15 under control of the graphics executive program.

This chapter discusses the assembly system for the McGRIN language, a modified version of BELLGRAPH's GRIN2. Assembled McGRIN programs are read into the PIP-15 and executed under the executives program's supervision. Each McGRIN source file (starting with a "G2ID" statement and finishing with an "END") is translated into one relocatable program block. Program blocks are treated by the memory management system as any other data block (e.g. nodes, leaves, branches) and are relocated in core whenever necessary, with all relocatable references in the block adjusted accordingly. A table at the bottom of each program block (called the "trailer") contains the relocation attributes of each word in the program. Each word is flagged as

(a) absolute (not dependent on address in core)

(b) relocatable

(c) block pointer (reference outside the program block)

Programs can refer to other programs by use of a block pointer. This pointer is a two word list giving the I.D. number (id) of the referenced program and the relative address (ra) of the word referenced.

Program blocks can be loaded on to the PDP-15 disc off-line using the G2LIBE program or can be input directly at run time under the executive. The McGRAPH executive expects the first program block from the high speed paper tape reader. Subsequent blocks are searched for on the disc. If not there, the executive returns to the paper tape reader for them.

As mentioned earlier, GRIN2 has been modified slightly in its implementation at McGill. The modifications are due to a change of computers used to assemble GRIN2 programs. The original GRIN2 assembler resided on a GE635 computer using the GMAP assembly language but it was later supported on an IBM 360 using the BAL assembly language. Incompatibilities between these two versions arise mainly in statements dealing with the control of the assembler itself and with macros.

At McGill, we are using the IBM system installed on a model 360/75 to assemble GRIN programs. In order to process the source code from Bell Laboratories (written in GMAP syntax), a preprocessing step is required to resolve the incompatabilities between the two systems, and translate the GMAP syntax into BAL. A PL/1 program, supplied by Bell Laboratories and later modified extensively here, is used to perform this preprocessing.

5.2 Assembler Implementation

The syntax of McGRIN (McGill's version of GRIN2) is that of IBM's 360/OS BAL assembly language (See IBM Manual #GC-28-6514-8 Assembler Language Reference Manual). All BAL assembler pseudo-operations are available plus a few additional ones specific to McGRIN. These extra pseudo-operations will be discussed later. A McGRIN programmer may define macros within his program using the BAL macro processing facility.

The McGRIN assembler also supports PDP-9 instructions so that a McGRIN program may consist of GRIN2 function statements intermixed with PDP-9 commands.*

The McGRIN assembly system consists of three segments, a preprocessor, a macro library and a postprocessor. The preprocessor modifies certain statements in GRIN source programs, which would cause assembly errors in the next job step, and expands certain pseudo-operations not available in BAL. The modified source, received from the preprocessor, is assembled by BAL using a macro library to define the GRIN function statements and PDP-9 instructions. A second smaller macro library is available for assembly of PDP-9 programs without GRIN statements. The executive and the off-line support programs are assembled using this library. The output of the assembly step is a deck of 32 bit data words, which is translated by a postprocessor program into a list of PDP-9 words. Currently the postprocessor outputs the list of PDP-9 instructions onto magnetic tape and a later step transcribes them to paper tape in the proper format. The postprocessor also optionally prints a listing of the program.

The remainder of this chapter is devoted to detailed descriptions of each assembly segment. Appendix B gives the control cards (job

*The PDP-9 instruction repetoire is a subset of the PDP-15's. Therefore, programs written for a PDP-9 will run on a PDP-15 without any modifications.

control procedure) used to assemble a GRIN program or PDP-9 program on McGill's IBM 360 HASP system.

5.3 The Preprocessor

The preprocessor is a PL/1 program used to alter certain statements in both GRIN and PDP-9 programs, making them acceptable to the BAL assembler.

Other illegal statements which are context sensitive and cannot be transformed by the preprocessor are flagged by the preprocessor and an accompanying error message is printed. The applications programmer must make these corrections by hand.

The preprocessor prints the resulting source file and outputs it on file ASMIN. (See Appendix B). A punch card deck can also be produced by specifying the "DECK" option. Another option "GEN" inserts a "PRINT GEN" statement in the output source file. This statement, upon assembly, causes all MACRO generated statements to be printed in the assembly steps output listings. The default condition is "NOGEN" or suppress MACRO expansions.

Besides adjusting the syntax, the preprocessor also executes a set of pseudo-operations to define and construct microprogrammed PDP-9 instructions. The microprogrammable PDP-9 instructions are grouped into three classes, the operate instructions, the extended arithmetic element (EAE) instructions, and the input/output transfers (IOT). The following paragraphs describe the preprocessor pseudo-operations used to define and generate each of these three classes of instructions.

5.3.1 The Operate Instruction Set

The operate instructions perform various functions on the accumulator and/or LINK. By setting the appropriate bits in the instruction word, a number of these functions can be executed by one instruction. For example,

```
(symbolic)   SZA!CLA   (internal)   750200
```

performs "skip if accumulator is zero" followed by a "clear accumulator" action. The resulting machine code is the inclusive OR of the machine codes for SZA and CLA instructions.

```
(symbolic)   SZA   (internal)   740200
             CLA                750000
                                ------
                                750200
```

To generate a microprogrammed operate instruction the AUGOPR (augment operate instruction) pseudo-operation is used. Its format is:

```
cols    1       8       16
                AUGOPR  (operation list)[mnemonic]
```

where the operation list is a list of operate instructions, and the mnemoric field is optional. If the mnemonic is present, a macro will be inserted into the output source defining the new microprogrammed instruction to the assembly step. Subsequent use of this mnemonic will generate the new microprogrammed instruction. For example,

```
AUGOPR     (SZA,CLA),SZC
```

will generate

```
           MACRO
&LOC       SZC
&LOC       QOPER      750200
           MEND
             .
             .
             .
             .
*          AUGOPR     (SZA,CLA),SZC
           SZC
```

The instruction QOPER is used to translate a 6 character string consisting of octal digits into an 18 bit binary word. Note, the original AUGOPR statement is retained as a comment to identify the following operate instruction.

An alternate technique is to first define the new instruction using DFNOPR pseudo-operation. For example, our original source could be

```
          DFNOPR    750200,SZC
            .
            .
            .
          SZC       ,
```

with the result being

```
          MACRO
&LOC      SZC
          QOPER     750200
          MEND
            .
            .
            .
          SZC       ,
```

The DFNOPR enables one to assign anemonics to any 6 digit octal word, not necessarily an operate instruction. The anemonics defined by DFNOPR or AUGOPR can be referenced in the operation lists of all other preprocessor pseudo-operations.

5.3.2 The IOT Instructions

The IOT instructions are used to control input or output peripherals. New IOT instructions can be defined using the DFNIOT pseudo-operation. For example,

```
DFNIOT          DSCF,707041
DFNIOT          DSFX,707042
```

generates macro definitions for two IOT's used to drive the RF09 disc. These two instructions subsequently can be microprogrammed by

```
AUGOPR          (DSCF,DSFX)
```

There is one restriction concerning the use of DFNIOT and AUGOPR to generate IOT instructions. One must be careful when constructing the graphics device IOT instructions, for a "jump to subroutine" (JMS G2SIM,*) instruction must follow each graphics IOT allowing it to be executed by the simulator program. This problem arises only when attempting to define new IOT's using AUGOPR or DFNIOT. All graphics device IOT's listed in Appendix A are already defined in the permanent macro library and need no special processing.

5.3.3 The EAE Instructions

The EAE class of instructions can be sub-divided into shift instructions, setup instructions, and step count instructions.

The shift instructions shift the contents of the AC and MQ registers left or right. For example,

```
LRS     9
```

shifts the AC-MQ registers 9 bits to the right. Note, the BAL assembler uses a decimal, not octal, number to specify shift count. A pseudo-operator called DFNSFT has been provided to define new shift instructions. For example,

```
                    DFNSFT      ACLS,640700
```

generates the following macro definition

```
                    MACRO
          &LOC      ACLS        &CNT
                    QSHFT       640700,&CNT
                    MEND
```

The QSHFT (supplied in the macro library) adds the shift count field to the least significant six bits of the PDP-9 word (e.g. 640700).

The EAE setup instructions perform operations on the registers, (i.e. step count, accumulator, and MQ) and are defined using the DFNEAE pseudo-operation. An example of the use of DFNEAE and the macro inserted by it is given below:

```
                    DFNEAE      LMQ,652000
```

generates

```
                MACRO
&LOC            LMQ
&LOC            STUP      652000
                MEND
```

The EAE step count instructions provide for such operations as normalization, multiplication and division. The assembly system allows one to override the default step count of these commands. For example, the divide instruction "IDIV" has a default step count of $23_8 = 19_{10}$. In other words, the divide operation performs 19 iterations producing an 18 bit quotient plus remainder from the division of two 18 bit integers. By explicitly specifying a step count, the length of the dividend, divisor and quotient can be varied. For Example,

```
        INDIV 11
```

performs an integer division on two ten bit integers. Unlike the shift instructions which adds the shift count to the default value, the step count field replaces the default count. New step count instructions can be defined by the DFNSTP pseudo-operation. It's format is identical to DFNIOT, DFNEAE, DFNSFT.

The previously defined EAE instructions can be microprogrammed using the AUGEAE instruction. It's format is as follows:

```
        Cols.  1      8        16
                      AUGEAE   (Operation list) [illegible]
```

The optional count is a decimal number or arithmetic expression giving the shift or step count, and the optional mnemonic defines the new instruction by a macro.

5.4 Preprocessor Error Messages

In a few instances, the preprocessor cannot resolve GMAP/IBM differences due either to a context sensitive translation or insufficient data. These statements it cannot process, are followed by an error message in the output source file and are printed in an error summary. A list of the possible error messages along with a brief discussion of each is given below.

5.4.1 ***ERROR*** UNDEFINED OPERATION

This message may follow the AUGOPR, AUGEAE operations or a GMAP conditional assembly statement. Referring to a AUGOPR or AUGEAE instruction, the error results when one of the mnemonics in the operations list of AUGOPR or AUGEAE is undefined to the preprocessor. This can be corrected by using one of the define mnemonic pseudo-operations discussed earlier (e.g. DFNOPR, DFNSTP, etc.). For example, the statement

```
                    AUGOPR          (SZC,IAC)
```

would cause this message since SZC is not defined in the standard PDP-9 instruction set. This could be corrected by the following:

```
                    DFNOPR          750200,SZC
                    AUGOPR          (SZC,IAC)
```

GMAP conditional assembly statements must be transformed manually. Table 5.1 gives the syntax of two GE conditional assembly operations and their IBM equivalent. The reader is referred to IBM manual #GC28-6514-8 "OS Assembler Language" for a more complete explanation of the syntax for IBM conditional assembly instructions. For example, the source

```
          A         SET             4
                    IFE             A,3,2
                    TAD             N100
                    CMA             ,
                    DAC             TEMP
```

must be translated into

```
          &A        SETA            4
                    AIF             (&A NE 3).OMIT
                    TAD             N100
                    CMA             ,
          .OMIT     ANOP
```

English	GE MACRO			IBM MACRO*		
Standard Macro syntax.	Name	MACRO			MACRO	
		.		&LOC	Name	&A,&B,&C
		.			.	
		.			.	
		ENDM	Name		MEND	
Symbolic Parameters						
First symbolic parameter is operand		#1			&A	
Second symbolic parameter is operand		#2			&B	
Third symbolic parameter is operand		#3 etc.			&C	
Conditional Assembly Statements						
If "a = b", then assemble next "n" statements		IFE	a,b,n		AIF	(&A NE &B).x
					-	
					-	n statements
					-	
				.x	ANOP	
If a ≠ b, then assemble the next "n" statements		INE	a,b,n		AIF	(&A EQ &B).x
					-	
					-	n statements
					-	
				.x	ANOP	
Concatenation**						
Symbolic parameter ‖ constant string "XYZ"		#2XYZ			&B.XYZ	
Constant string ‖ symbolic parameter		XYZ#2			XYZ&B	
Symbolic parameter ‖ symbolic parameter		#2#1			&B&A	

, &B, &C are symbolic parameters in IBM Macros

‖ denotes concatenation.

TABLE 5.1. GE GAMP TO IBM BAL MACRO SYNTAX CONVERSION.

5.4.2 ***ERROR*** INVALID SYNTAX

This error occurs whenever the preprocessor encounters a conversion error when converting a character string to an integer. Scrutiny of the statement before this error message will uncover a non-numeric character in an exclusively numeric field. For example the statement

```
BUFFER      BOOL      45-36
```

will cause a syntax error since the BOOL statement permits only octal numbers, not expressions, in its operation field.

8

5.4.3 TOO MANY BAD SYMBOLS

If the number of illegal symbols found in the input source exceeds 70, a table in the preprocessor overflows and the process aborts. All preprocessed statements up to the time of table overflow are output.

5.4.4 ***ERROR*** ARG LIST TOO LONG - USE ETC. STATEMENT

The preprocessor inserts commas between items in the argument lists of GRIN2 function statements. If the resulting list overflows the card boundaries processing of the statement is stopped and the above error message is printed. The suggested solution to this problem is to continue the argument list on a following "ETC card. The following example illustrates the use of ETC. The statement

```
   TREE    OP1,(,LW,UF)WI)(,RW,UF)WI)(,(LW,0)DOOR)(,(RW,LF)WI)(,(LW,0)DOOR)
```

is too long to be preprocessed. By changing the input source to

```
   TREE    OP1,,(,(LW,UF)WI)(,(RW,UF)WI)
   ETC     (,(RW,LF)WI)(,(LW,0)DOOR)
```

the statement is correctly translated into

```
   TREE    OP1,,(,(LW,UF),WI),(,(RW,UF),WI)                        +
           (,(RW,LF),WI),(,(LW,0),DOOR)
```

The character "+" in column 80 indicates to the BAL assembler that the statement is continued on the next card.

5.4.5 ***GE MACRO***

All GMAP macros in the input source deck are printed in the error message list and deleted from the program. BAL macros are left unaltered. The programmer must replace the discarded GMAP macros by their BAL equivalent. Table 5.1 gives the conversion of GMAP macro syntax to that of BAL.

5.4.6 A LIST OF ALL SYMBOLS OF ILLEGAL SYNTAX

This printout is not an error message but is included in this discussion to explain its occurrence in the preprocessor's printed output. Certain characters within symbols are acceptable to GMAP but are illegal in IBM assembler language. The preprocessor replaces these offending characters by others accepted by BAL, and prints a list of symbols so altered. It is left to the programmer to insure the preprocessor's replacement symbol is not already defined in the input source (a highly unlikely occurrence).

The preprocessor identifies two classes of illegal symbols and makes the appropriate character replacements. Symbols whose first character is a numeric (0 through 9) have their first character replaced by "N". The character "." in symbols in the label field or in the operand field of a SYMREF statement is replaced by the character "@". References to this symbol in the operand field of all other statements are adjusted accordingly. There is one exception to this rule. BAL sequence symbols appear in the label field and are identified by a "." as the first character. To permit sequence symbols in the input source, they are further restricted to the form

.Zxxxxxx

where x is any letter or digit.

All label symbols of this form are passed unaltered by the preprocessor. In order to clarify the symbol adjusting procedures let us look at the following example. Consider the input source

```
             SYMREF       .DR2
&N           SETA         3
             LAC          .DR1
             DAC          .X1
.DR1         LAW          -2
             AIF          (&N EQ 3).ZSKP
35DF         LAW          -3
.ZSKP        ANOP
             DAC          35DF
```

The resulting output from the preprocessor:

```
A LIST OF SYMBOLS OF ILLEGAL SYNTAX
            @DR2
            @DR1
            N5DF
```

(Program continued on following page)

```
                ICTL      1,79,80
                PRINT     NOGEN
                BEGIN     ,
                SYMREF    @DR2
      &N        SETA      3
                LAC      @DR1
                DAC       .X1
      @DR1      LAW       -2
                AIF       (&N EQ 3).ZSKP
      N5DF      LAW       -3
      .ZSKP     ANOP
                DAC       N5DF
                QEND      ,
                END
```

The additional statements ICTL, PRINT, BEGIN and QEND, inserted in every source program, are used to control the assembler and do not concern us here. In our example, symbols .DR2, DR1, and 35DF are found to be illegal and all instances of them are modified. Symbol .X1 is left unaltered since it is not defined in any label field or in the operand field of a SYMREF statement. Such symbols refer to McGRIN system variables and must not be changed. The use of these system variables are explained in Section 5.5.

5.4.7 ***ERROR*** IF OPERAND IS ABSOLUTE PRECEDE ALL REFERENCES TO LABEL BY '+'

This warning message is printed after each EQU statement and flags potential trouble in the input source program. Due to an implementation restriction of the assembler, absolute expressions in the operand field of memory reference instructions must be preceded by a "+" if their value is greater than 4095. For each EQU statement the programmer must determine:

1. If its operand is relative or absolute.
2. If absolute, is its value greater than 4095.
3. If its value is not easily determined then it should be assumed to be greater than 4095.

Next, the programmer must search the source code for references made to symbols equated to absolute values and precede them with a "+". The following EQU statements illustrate this procedure.

```
   Original Source                  Source Modified by Programmer

LABL1   LAC    A                  LABL1    LAC     A
        SNL                                SNL
LABL2   DAC    B                  LABL2    DAC     +B
               .                                   .
               .                                   .
               .                                   .
A       EQU    LABL2              A        EQU     LABL2
B       EQU    4096               B        EQU     4096
```

References to symbol "A" require no changes since A is equated to label "LABL2", a relative quantity. References to symbol "B" require a preceding "+" since its value is absolute and greater than 4095.

5.5 The Assembly Step

This section describes the actual assembly process in the McGRIN assembly system. Details concerning the syntax and format of the BAL assembly language are given in the IBM "OS Assembler Language" manual #GC28-6514-8. The potential applications programmer may want to refer to this manual while reading this section to achieve a detailed understanding of the assembly system. To the casual reader, the semantics of the BAL statements should be sufficiently described by their context.

There are two assembly steps available to the McGRIN programmer. One is used to generate relocatable program blocks from programs written in preprocessed GRIN2. The other is a PDP-9 assembly language package to generate absolute PDP-9 programs. The GRIN2 program blocks generated by the GRIN assembly systems are prepared on paper tape and input to McGRAPH's PDP-15 under the control of an executive program. The PDP-9 assembly system is used to prepare absolute core loads of the PDP-9 monitor system (i.e. the executive programs) and accompanying off-line support programs (e.g. G2LIBE). The paper tapes containing absolute core loads have a bootstrap loader placed at the beginning and are read in via the hardware READ-IN facility of the PDP-15.

As mentioned earlier the GRIN2 language contains the PDP-9's instruction repetoire along with its own function statements. Since the PDP-9 instructions in GRIN2 have the same syntax as those supported by the IBM/PDP-9 assembler, they can be processed by either assembly step. However, the resulting output format of the two systems is different. It is important to remember this fundamental distinction between these two assemblers to avoid confusion in deciding which one to use. In this section, we will present the syntax of the PDP-9 assembler language input to either of these assembly steps. The preprocessing is assumed to be done. The preprocessor is a large (300K) and expensive program to run, so it is suggested all new programs written at McGill be directly input to the assembly step. Instructions written in the syntax presented here need no preprocessing before the assembly step. The user is referred to Appendix C for a list of modifications to GRIN2 programs syntax which, if done by hand, avoid preprocessing.

After the discussion of the PDP-9 assembly language syntax we will present two problems encountered in implementing BELLGRAPH programs on McGRAPH and explain the modifications made to the assembler to overcome them.

5.5.1 PDP-9 Assembly Language Implementation

Two libraries of BAL macros residing on the IBM 360's on-line disc storage are used to translate the PDP-9 source instructions into a list of 32 bit words which are later processed by a postprocessor into 18 bit machine instructions. One library is reserved for GRIN2 programs, the other for absolute PDP-9 code. The mnemonics of the PDP-9 instructions are those used in DEC's own MACRO-9 assembler language but the instruction syntax has been altered slightly to accomodate BAL's macro calling format. A number of additional pseudo-operations and storage defining statements are also available increasing the power of the assembly language.

The applications programmer or potential applications programmer, of McGRAPH should become familiar with the statement and macro format, terms and expressions, and types of symbols in BAL. These are adequately described in the "OS Assembler Language Manual #GC28-6514-8. Only those features unique to McGRAPH's PDP-9 instruction format are presented here.

5.5.2 Memory Reference Instructions

The PDP-9 memory reference instructions format is as follows:

```
Cols   1          8           16
       Location   Operation   Operand [,indirect] [comments]
```

Note, square brackets always indicate optional fields and the LOCATION field can contain an ordinary BAL symbol (See page 11 of OS Assembler Language Manual). The OPERATION field must contain a mnemonic representing one of the thirteen memory reference instructions or one of the special instructions ZERO or ADRS (See DEC PDP-15 Reference Manual DEC-15-BRZC-D, page 6-1). The OPERAND may be one of the following:

(1) A signed decimal literal. Decimal literals are indicated by a preceding = sign. For example,

```
LABEL    LAC    =19ØØ
```

(2) A signed octal literal. Octal literals are indicated by a preceding =0. For example,

```
LABEL    LAC    =01ØØ
```

(3) A relocatable symbolic address or symbolic address + or - an absolute symbol or constant. The symbol "*" is a legal relocatable symbolic address denoting the current value of the location counter. For example,

```
JMP    *-1
```

Note: Caution must be taken when using the location counter. See indirect addressing.

(4) An absolute decimal address or absolute symbol. Absolute symbols whose value is greater than 4095 must be preceded by a +. For example,

```
        ADDR    EQU     4096
        LABEL   LAC     +ADDR
```

The symbol ** assembles an address of 0.

(5) An external symbolic address or an external symbolic + or - an absolute symbol or constant.

Two modes of indirect addressing are available by placing an "I" or a "*" in the INDIRECT FIELD.

(1) An "*" in this field denotes the normal indirect addressing mode. In a PDP-15 the referenced word contains a 16 bit effective address. For example,

```
                LAC     TABLE,*
        TABLE   OCTAL   57777
```

loads the AC with the contents of location 57777.

(2) An "I" in this field denotes the referenced word contains a 13 bit effective address. The upper bits are ignored. For example,

```
                LAC     TABLE,I
        TABLE   OCTAL   57777
```

loads the AC with the contents of 17777.

Indirect memory reference instructions using 13 bit address generate three or four PDP-15 words. The reason for these two modes of indirect references is due to differences in operation between our PDP-15 and Bell Laboratories modified PDP-9. Section 5.6 gives a complete description of the differences and explains how the modified indirect memory references are simulated by software on McGRAPH.

5.5.3 The ZERO and ADRS Memory Reference Operations

ZERO and ADRS operations produce a data word containing a 13 bit address. For example,

```
LABEL        CLA       ,
              .
              .
              .
DATA         ZERO      LABEL
```

places the value of LABEL in word DATA. The operation ADRS is used if the operand is an external reference.

5.5.4 The DECML and OCTAL Operation

The DECML and OCTAL operation are used to place a list of constants in the program. Their format is

```
Cols  1           8          16
      Location    BSS        N           Comments
```

where N is an absolute decimal quantity specifying the number of locations to be allocated. The symbol in the LOCATION field is given the value of the location counter before allocation takes place. When allocating storage it cannot be assumed that the contents of the entire block is zero.

5.5.5 The BCD Operation

The BCD operation packs alphanumeric characters three per 18 bit word for printing on the teletype. Its format is:

Location	BCD	Number, (Text)	Comments

The NUMBER argument gives the number of characters in the TEXT argument. The TEXT argument is the desired information. This argument must be enclosed by parentheses and cannot contain spaces or parentheses. A space is represented by a ¬ sign and a carriage return and line feed by < or ; .

5.5.6 The EQU and EQUR Pseudo-Operations

These operations are used to establish equivalence between plus or minus a constant in the EQUR case.

```
Cols.   1           8
        Symbol*     EQU         Absolute Expression
                    EQUR        Symbol ± constant
```

The EQU pseudo-operation is not defined by a macro, but is implemented directly in BAL. It is presented here to illustrate the difference between it and EQUR, a macro implemented pseudo-operation. Since EQU is not implemented by a macro, system symbols (See 5.5.14) in its operand field must have their illegal characters replaced manually by the programmer. For example,

```
        TEMP        EQU         .16
```

is illegal. This must be written

```
        TEMP        EQU         #16
```

5.5.7 The BOOL Pseudo-Operation

The BOOL operation establishes an equivalence between a symbol and a positive octal number

```
Cols.   1           8           16
        Symbol*     BOOL        Octal Number
```

5.5.8 The NULL Pseudo-Operation

```
Cols.   1           8          16
        Location    NULL        ,
```

The symbol in the LOCATION field is given the current value of the location counter. If the location field is empty nothing is done.

5.5.9 The ENTRY, SYMDEF Statements

The ENTRY or SYMDEF statements declare a set of symbols defined in this program to be external entries which can be used by other programs. Their format is

```
Cols.   1     8        16
              ENTRY    A list of up to 10 symbols     Comments
              SYMDEF
```

These statements are not supported in the current version of McGRIN but are available in the PDP-9 assembler.

5.5.10 The EXTERN, SYMREF Statements

The EXTERN or SYMREF statements list the symbols used in the program which are defined in another. Their format is identical to ENTRY or SYMDEF. These statements are not supported in the current version of McGRIN but are available in the PDP-9 assembler.

5.5.11 The CALL Statement

The CALL statement is used to call an external subroutine and supply its arguments. It has the following format,

```
Cols. 1          8          16
      Location*  CALL       Name, (Arguments)        Comments
```

The NAME field contains the name of the external ENTRY point of the subroutine. This name must not appear in the operand field of a SYMREF statement since the CALL generates its own external reference statement. The parenthesized list of arguments generates a word for the address of each symbolic parameter or absolute argument within the list.

5.5.12 The DEBUG Pseudo-Operation

The DEBUG pseudo-operation has the following format,

```
          DEBUG     OFF
or
          DEBUG     ON
```

With the DEBUG switch ON, the octal address of each assembled statement is printed so that symbolic statements can be traced to the memory dump given by the postprocessor. For example,

```
          JMS       DCLEAN
                    *,*          000127
```

The JMS instruction is at octal location 000127 relative to the start of the program. The default for all assemblies is DEBUG ON, but this can be reversed by DEBUG OFF. In addition, by issuing just DEBUG, the switch will restore the mode prior to the last DEBUG ON or OFF statement.

5.5.13 BELLGRAPH System Symbols

There is a set of symbols defined internally (in the BEGIN and QBEGIN macros) for each GRIN2 or PDP-9 program. These symbols refer to either locations in the executive's transfer vector table, or to standard system values. They are denoted by a "." character followed by up to five alphanumerics, or by ".." followed by up to four alphanumerics. For example, .X1 is the label assigned to location 11, the second auto-increment register in a PDP-15; .LPOF is a parameter statement argument indicating the light pen is to be disabled.

An anomaly seems to exist concerning the system symbols. As mentioned earlier, BAL rejects symbols containing "." but these system symbols are perfectly legal. To explain this, one must recall how the BAL assembler operates. All macros calls are processed and expanded first. Then the resulting code is assembled. The macro package will accept symbols of any format. Those preceded by a "." are processed as system symbols by the macro package and the "." is changed to a "#" (a legal symbolic character) before assembly.

5.5.14 Blank OPERAND Field

Blank OPERAND fields are denoted by a comma in column 16. Statements which do not have an OPERAND field or whose OPERAND field is blank must have this comma. For example,

```
Col.  1          8            16
                 SKP          ,
```

5.5.15 Statement Continuation

Statements are continued onto the next card by punching any character (normally a + sign) in column 80. The rest of the statement starts in column 16 of the next card.

Source decks which are preprocessed before assembly use the preprocessor pseudo-operation ETC to indicate a statement continuation. ETC pseudo-operation causes the preprocessor to insert a + sign in column 80 of the previous card and erase the "ETC" characters from the current card.

5.5.16 Program Control Statements

Each program requires a set of assembler control statements to define the card statement boundaries, the control sections to be used, and the BELLGRAPH system variables. PDP-9 programs are bound by the following control cards,

```
                        ICTL      1,79,16
                        BEGIN     ,
                        PRINT     NOGEN
                          .
     (Body of Program)    .
                          .
                        QEND      ,
                        END
```

All the control statements, except END are generated by the preprocessor. Of course, if the preprocessor step is omitted, they must be inserted manually.

GRIN2 program blocks are identified by the following control statements;

```
1CTL      1,79,16
G2ID      XXX
PRINT     NOGEN
  .
  .
  .
QEND      ,
END
```

where XXX in the G2ID statement gives the program block's I.D. number. The G2ID statement is not inserted by the preprocessor.

5.6 - Software Execution of Indirect Memory References

In Section 5.5.2 we briefly discussed two modes of indirect memory references in the McGRAPH version of the BELLGRAPH software. The first mode is the conventional procedure used by the PDP-15. That is, the effective address used by indirect memory referenced instructions is found in the low order sixteen bits of the word pointed to by the instruction.

The second mode is used by Bell Laboratories' modified PDP-9 processor. In this machine, the effective address of indirect memory reference instructions is contained in the thirteen low order bits of the address word. Since the upper bits of these words are not considered part of the address, they may be used to store other data. The BELLGRAPH programs in fact do assign other uses to them. On a conventional machine, the non-zero high order bits in a word become part of the address, preventing proper execution of BELLGRAPH programs.

To run BELLGRAPH at McGill we require some technique to circumvent this problem. The system programs make far too extensive use of words containing both 13 bit address and data to make changing them practical. Modifying the PDP-15's processor to operate as Bell Laboratories' is not possible since McGill's machine must support other existing software systems as well. (Note this mode of indirect addressing limits the addressing range to 8K, a severe restriction). The last possible approach (used in McGRAPH) is to simulate by software the truncation of the address word on each indirect memory reference instruction. Doing this, one pays a penalty in both core used and execution time. Each indirect memory reference instruction requires two or three extra words and takes approximately twenty times longer to execute. Since indirect memory reference instructions comprise less than 10% of all executable code the increased execution time is not unreasonable.

Software calculation of 13 bit effective addresses is invoked by inserting subroutine calls before each indirect memory reference statement.

The assembler expands indirect references into the following sequence of commands,

```
JMS   FIX,*      Jump to indirect processing subroutine
OPR   LABEL,*    Instruction to be executed by FIX
OPR   LABEL,*    2nd word of calling sequency used for work space.
```

where OPR LABEL,* is an indirect memory reference statement (e.g. LAC TABLE,*). The subroutine FIX computes the 13 bit effective address of the indirect memory reference instruction immediately following it, and places the newly constructed direct memory reference instruction in the second word of its calling sequence. Control is then passed to this fabricated instruction. Initially the second word contains a copy of the indirect memory reference instruction to insure it is considered a relocatable word by the monitor's memory management system. If an interrupt occurs during the execution of FIX, it's internal temporary storage is stacked by the interrupt handler making FIX re-entrant.

The above instruction sequence for simulating 8K wrap around of indirect memory referenced instructions fails under certain conditions. For example, consider the following piece of code.

```
SNL                   SKIP ON NON-ZERO LINK
DZM       X1T,I
```

When expanded by the assembler this would become

```
SNL
JMS     FIX,*
DZM     X1T,*
DZM     X1T,*
```

To prevent the subroutine call to FIX being by-passed by the skip command, an additional jump instruction is inserted in the calling sequence. Our example expands to

```
SNL
JMS     FIX,*
JMP     *+3
DZM     X1T,*
DZM     X1T,*
```

Now on a skip due to non-zero link, the DZM command is not executed. Note this expansion consumes four words of memory for each indirect memory reference. Assembled BELLGRAPH programs using this expansion created object modules too large to load into the PDP-15. Therefore, the indirect memory reference expansion macro inserts the "JMP *+3" only if the indirect memory reference command is preceded by one of the following commands.

1. Jump to subroutine (JMS)
2. Execute memory location (XCT)
3. All skip instructions (SKP, SNL, SZL etc.)

**4. All IOT instructions whose last digit is odd.

5. Skip if memory not equal to ACC (SAD).

6. Increment memory and skip on zero (ISZ).

7. Jump (JMP).

There is one other situation which is potentially troublesome when using indirect memory reference expansions. Transfer commands of the form

```
JMP    *+n
```

where

*   represents the current location counter, and

n   is any decimal number,

may no longer transfer to the desired location if there is an expanded instruction nearby. For example, consider:

```
JMP    *+3
LAC    XIT,I
DAC    TMP
CLA
```

Because of the expansion of "LAC XIT,I" the code must be altered to

**It is a convention in PDP-15 IOT commands that bit number 17 of the instruction signals a skip on flag condition.

```
                JMP     LABL
                LAC     XIT,I
                DAC     TMP
        LABL    CLA
```

The use of the location counter (*) in JMP instructions was rare in BELLGRAPH programs so they have been removed manually.

5.7 Software Execution of Graphics IOT Commands

In Chapters III and IV, we discussed the simulation of the original BELLGRAPH graphics devices by executing the graphics device IOT by software. This is accomplished by writing a BAL macro defining the mnemonics of the IOT commands as subroutine calls to the simulation routine G2SIM. Each PDP-9 load module assembled using this system which contains graphic IOT commands must contain the following transfer vector location

```
G2SIM     OCTAL     20000     Starting Addr. of G2SIM
```

Graphics IOT expansions generate an indirect transfer through the location G2SIM.

GRIN2 or McGRIN programs use a system variable ".G2SIM" to refer to the simulation transfer vector location in the executive system.

The IOT's in the McGRIN assembly package refer to this system variable. For example the command BEG (begin display) expands to

```
700547
JMS          .G2SIM,*
```

The programmer need not concern himself with the generated call to G2SIM except to remember IOT commands generate two words.

5.8 The Postprocessor Step

The postprocessor is a PL/1 program which transforms the object modules from the assembler into files acceptable to the PDP-15.

PDP-9 object modules from the assembler are linked together by the postprocessor forming a PDP-9 executable load module. A bootstrap loader is automatically inserted at the beginning of the load module so that it can be read into the PDP-15 using the hardware READ-IN facility. The postprocessor prints an external reference table giving the definition of each external label and a list of all references to it. Unresolved references generate error messages. A dump of the entire load module is also printed giving the contents and octal address of each word.

GRIN object modules are transformed into program blocks which are read into the PDP-15 using the executive program. The post processor builds a relocation table called the "trailer" at the end of each program block to provide the memory management system information for relocating the block in core. A dump of the program block excluding the "trailer" is printed for each GRIN object module.

The postprocessor will produce a listing of both program blocks and load modules if the option "LIST" is included in the parameter field (See Appendix B). The listing merely gives the mnemonics of each instruction word followed by its octal operand. Due to the lack of comments this option is not very useful. A punched deck of the input object modules is produced when the option "DECK" is given. This option is useful when the input modules reside on disc, and one would like to save them on cards and remove the disc files upon postprocessing.

The output modules from the postprocessor are currently placed on 9 track 800 BPI magnetic tape, and later transcribed to paper tape using a SABR program "TAPNCH" which runs on a PDP-12. This extra step is necessary since there is no data medium common to McGRAPH's PDP-15 and the IBM 360. See Appendix B for a listing and description of TAPNCH.

CHAPTER VI

CONCLUSIONS

6.1 Background

This project was undertaken to provide a graphics language for the users of the McGRAPH disc oriented display system. The language is required to be sufficiently general to be applicable to a wide class of problems, since users of McGRAPH are researching in many different areas of Electrical Engineering and Computer Science. With such a general purpose graphics language, the individual applications programmers can quickly apply interactive graphics techniques to their problem solving.

At the beginning, numerous existing graphics systems were surveyed to determine those features common and/or desirable in a graphics language. It was found that during a normal session using an interactive graphics terminal, the displayed pictures, hence their data representation, continually grow and diminish. Demands to the system for storage fluctuate widely in real-time and cannot be anticipated at program generation time. For this reason, interactive graphics software must provide some sort of dynamic memory management scheme which allocates space on bulk storage and in main memory.

Another observation made during a session of computer-aided design, using an interactive graphics terminal, concerns the manner in which pictures are generated. The designer first defines the basic picture parts or primitives (e.g. resistors, capacitors, molecules, shapes) he intends to use. He then combines these into sub-assemblies which can be reproduced and inserted into a larger assembly and so on. Complex scenes are quickly produced by progressively adding new components made of previously defined objects. Usually, information concerning the hierarchy of picture parts in a total display is significant, and should be saved in the computer's data structure.

Based on these observations, one can state that dynamic memory management and a hierarchical data structure are fundamental to computer graphics, and should be included in McGRAPH. Although these features exist on large data processing machines, minicomputer manufacturers have yet to write operating systems of this complexity.

Therefore, we had a choice of developing our own, or attempting to obtain an operating system from another source. The technique of obtaining an already existing system appeared promising, since it afforded the opportunity to attain a graphics system of more sophistication and power than could be produced here given the limited available time.

In our evaluation of different systems, the Bell Telephone Laboratories' BELLGRAPH was particularly attractive since besides providing the attributes of a good general purpose graphics system, it was written for a PDP-15. The accompanying "GRIN2" language suited our requirements perfectly. It is a general purpose picture synthesis language which fully exploits the dynamic memory allocation provided by the operating system and is open-ended so that additional features, such as picture analysis statements, are easily added to it.

6.2 Implementation

Two major obstacles prevented direct implementation of BELLGRAPH on McGRAPH. First, the display hardware is very different especially the display refresh policy. This has great impact on the operating philosophy of the software system. We shall discuss this later in Section 6.3. Second, the BELLGRAPH programs were written for a 8K PDP-15 whose processor was modified so that all memory references used only the 13 lower bits of the address bus. The full significance of this problem was found only after an intensive journey deep into the listings. A software fix up was found, as described in Chapter IV, which performs adequately.

6.3 McGRAPH's Refresh Policy

The original BELLGRAPH system supplied to us operates in a single task environment. That is, only one operation is performed at a time. For example, when the display is running, the display routine cycles through the data structure and continually passes DPU commands to the display. Interrupts occuring during this process stop the display and initiate other routines. The display is resumed or restarted by either the interrupting routine or by subsequent language statement subroutines in the running GRIN2 program. The time interval, during which the screen is dark, ranges from a few microseconds to seconds.

In McGRAPH, the display monitor program traverses the data structure once and then idly awaits an interrupt. Future additions to McGRAPH could make extensive use of this idle time. For example, a background FORTRAN program could be swapped into core and be executed while the graphics user is deciding what action to take next.

BELLGRAPH requires the entire data structure be in core during its display since its continual refresh does not permit time to bring sections in from disc. McGRAPH imposes no such restriction. The display disc on the PDP-8 poses a lower limit on the time taken to update the screen. This is a characteristic of the hardware (the disc) which cannot be improved by software. However, we can effectively multiplex the time taken by the PDP-8 in transfering a picture to the display disc, by allowing the PDP-15 to get the next data segment from its disc during this time. Therefore, we can reduce the amount of PDP-15 core required by display data without significantly degrading the display update time.

6.4 Future Extensions

Although the major implementation problems have been overcome, some effort is still required to make the McGRAPH version of BELLGRAPH a convenient workable system. The bottleneck in transcribing the IBM assembler's output from magnetic tape to paper must be removed. The data channel between McGRAPH's PDP-8 and the IBM 360 at the McGill Computer Centre could greatly improve access of the assembler to future GRIN and McGRAPH system programmers, although this is not working at present. Establishment of the inter-computer link is of prime importance.

Bell Telephone Laboratories at Holmdel also provided us with a GRIN assembler which resides on the PDP-15 and runs under the BELLGRAPH operating system. By installing this assembler McGRAPH would be free from its IBM 360 dependency.

On the other Hand, the availability of a link to the IBM 360 opens many avenues of research. Further modifications to the operating system and the PDP-15 could permit GRIN programs access to the IBM machine's large computational and storage facilities.

Extension to the GRIN language for picture analysis as well as synthesis is a possibility. Provision for an on-line algorithmic language as discussed in Chapter II is another desirable feature.

6.5 Conclusions

BELLGRAPH has a number of drawbacks. It was designed for an 8K computer and cannot use a larger amount of core without completely rewriting it. The GRIN assembly system on the IBM 360 is cumbersome and expensive. It is incompatible with Digital Equipment Corporation's PDP-15 operating system and therefore cannot interface with FORTRAN programs.

However, we must weigh these against the graphics power achieved via the GRIN language. Such a language could not have been developed in the time available. Full appreciation of the power of GRIN will have to await actual applications and further experience.

APPENDIX A

SIMULATED DEC 339 PROCESSOR

This appendix includes detailed information needed to interface the DEC 339 display processor simulation programs with the user's own. The simulator program package enables one to run PDP-9 or PDP-15 programs, which drive a DEC 339 display, on McGRAPH. It consists of four separate absolute programs which are loaded in the middle 8K memory bank of the PDP-15.

The source code of these four programs are saved in the following files on DEC tape #122.

1. G2SIM - IOT simulator.
2. G2TRAN - DEC 339 DPU command translator.
3. SYSGGN - Character generator.
4. SYSVGN - Vector generator.

A.1 Simulated Graphic Device IOT's

The graphics devices are driven by user programs via calls to the simulation program package. These calls are of the form

```
IOT
JMS   G2SIM,*
```

where IOT is an PDP-15 IOT command addressing a non-existent device. Since the device is non-existent, the IOT executes as a no-operation (NOP). The list of IOT's below describe those recognized by the simulation program.

The assembly system on the IBM 360 generates the appropriate call to G2SIM when it encounters the graphic device IOT mnemonics.

For a complete description of the action of these IOT's, see the memo "Graphic-2 Hardware Organization" in the accompanying BELLGRAPH documentation.

Mnemonic	Instruction	Description
		Console Keyboard (device #43)
CCK	704304	Clear flag.
LCK	704312	Load console keyboard into AC.
SCK	704301	Skip on flag.
OCK	704302	OR console keyboard with AC.

Mnemonic	Instruction	Description
		Pushbuttons (device #44)
SPB	704401	Skip on flag.
LPB	704412	Load pushbutton buffer into AC.
OPB	704402	OR pushbutton buffer with AC.
CPB	704404	Clear pushbutton flag.
		Pushbutton Lights (device #442)
WBL	704424	Write contents of AC into light buffer.
LBL	704432	Load light buffer into AC.
		Light Pen (device #07)
ELP	700701	Enable light pen.
DLP	700721	Disable light pen.
		Display (device #05)
CDF	700501	Clear all display flags.
WDA	700502	Write display address from AC.
BEG	700547	Start the display at location specified by the contents on the 13 low order bits of the AC.
		McGRAPH Display (device #46)
(The following IOT's are unique to McGRAPH)		
CCRT	704601	Clear display screen.
TCRT	704621	Initialize tracking pattern (tracking pattern is display at the position given by .WHEREX and .WHEREY).
DCRT	704602	Display leaf. (A leaf number and sub-number are inserted into the file header).

Mnemonic	Instruction	Description
		DPU Registers
LX (or QLX)	701412	Load the AC with the contents of the X register.
LY	703412	Load the AC with the contents of the Y register.
LDS	701052	Load the AC with the display status flags.

A.2 Filename Allotments

Display files on the PDP-8 disc are separated by HEADER commands containing 12 bit filename I.D.'s. These 12 bit words are divided into two fields, an 8 bit leaf number field, and a 4 bit sub-leaf number field. (See Figure 4.5). Certain filename I.D.'s are reserved for system use. The list below gives the octal contents of the leaf and sub-leaf fields of the reserved filenames plus those used for leaves and messages.

00 000	-	00 007	Simulated lighted pushbuttons.
00 010	-	00 027	Tracking pattern.
00 030	-	00 037	Spare.
00 040	-	17 040	Message numbers.
00 050	-	17 367	Leaf and sub-leaf number.
17 370	-	17 373	System spare.

17	374	Light pen insensitive file.
17	375	Turn off blink file.
17	376	Turn on blink file.
17	377	Track original file.

A.2.1 Leaf and Sub-Leaf Filename I.D.'s

Each instance of a leaf in a BELLGRAPH data structure is given a 8 bit identification number. Each seperable picture part within the leaf is given a 4 bit number. These two numbers concatenated together give a unique filename I.D. for each seperable picture part displayed. By subdividing the filename I.D. into leaf and sub-leaf numbers, the search through the data structure, which the executive must make on a light pen strike, is speeded up.

A.2.2 Message Filename I.D.

At times one may wish to display a picture not represented by a BELLGRAPH data structure. In this case we cannot use a traverse through the data structure to find the display status information associated with the instance struck by the light pen. Non-structured instances are generated by a "BEG" IOT which assigns a "message number" instead of "leaf/subleaf number" to each instance. In this mode, the contents of the display parameters, display status, and X and Y registers are saved for each instance. On a light pen strike, these registers are loaded with the saved values so that they can be interrogated by the program.

A.3 GRAPHIC-2 SCOPE COMMANDS

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
CHARACTER	0	0	0	0														
PARAMETER	0	0	0	1	SET	BLINK	SET	LP	S/A	E	C_X	C_Y	SET	S_0	S_1	SET	I_0	I_1
LONG VECTOR	0	0	1	0	CONTROL		X=0 Y=1	+=0 −=1	Δ_0	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
X-Y	0	0	1	1	DELAY	INT	X=0 Y=1		0	1	2	3	4	5	6	7	8	9
SHORT VECTOR	0	1	0	0	CONTROL		+=0 −=1	ΔX_0	ΔX_1	ΔX_2	ΔX_3	ΔX_4	+=0 −=1	ΔY_0	ΔY_1	ΔY_2	ΔY_3	ΔY_4
INCREMENT	0	1	0	1	REPEATS			INT	DIRECTION			REPEATS			INT	DIRECTION		
CONTROL	0	1	1	0	STOP	COND STOP					SET	OVRD	SET	BRKN				
TRAP	1																	

CHARACTER: bits 4–10 CHARACTER 1; bits 11–17 CHARACTER 2

PARAMETER: bits 4–5 BLINK; bits 6–7 LIGHT PEN; bits 8–11 SYMMETRY; S/A: 0=ACCUMULATE, 1 = SET; bits 12–14 SCALE; bits 15–17 INTENSITY

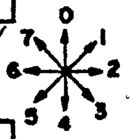
LONG VECTOR, CONTROL (bits 4–5):

00 → LOAD HOLDING REGISTERS ONLY
01 → LOAD REGISTERS, DRAW INVISIBLE, CLEAR REGISTERS
10 → LOAD REGISTERS, DRAW VISIBLE (EXCEPT STARTING POINT), CLEAR REGISTERS
11 → LOAD REGISTERS, DRAW INVISIBLE EXCEPT END POINT (WHICH IS VISIBLE), CLEAR REGISTERS

X-Y: DELAY: 0 = NO DELAY, 1 = 35μs DELAY; INT: 0 = INVISIBLE, 1 = VISIBLE; bits 8–17 COORDINATE

SHORT VECTOR: CONTROL: 00 — NO OPERATION, 01, 10, 11 — SAME AS IN LONG VECTOR; bits 6–11 ΔX COMPONENT; bits 12–17 ΔY COMPONENT

INCREMENT: INT: 0 = INVISIBLE, 1 = VISIBLE; bits 4–10 INCR 1; bits 11–17 INCR 2



CONTROL: bits 10–11 OVERRIDE 1=ON 0=OFF; bits 12–13: 0 = DRAW SOLID LINES, 1 = DRAW BROKEN LINES

TRAP: bits 1–3 SPECIFIES API CHANNEL LOCATION TO TRAP TO

A.4 CHARACTER CODES

ASCII* Character Set

Character	8-Bit Octal	6-Bit Octal	Character	8-Bit Octal	6-Bit Octal
A	301	01	!	241	41
B	302	02	"	242	42
C	303	03	#	243	43
D	304	04	$	244	44
E	305	05	%	245	45
F	306	06	&	246	46
G	307	07	'	247	47
H	310	10	(	250	50
I	311	11	)	251	51
J	312	12	*	252	52
K	313	13	+	253	53
L	314	14	,	254	54
M	315	15	-	255	55
N	316	16	.	256	56
O	317	17	/	257	57
P	320	20	:	272	72
Q	321	21	;	273	73
R	322	22	<	274	74
S	323	23	=	275	75
T	324	24	>	276	76
U	325	25	?	277	77
V	326	26	@	300	
W	327	27	[	333	33
X	330	30	\	334	34
Y	331	31	]	335	35
Z	332	32	↑	336	36
0	260	60	←	337	37
1	261	61	Leader/Trailer	200	
2	262	62	LINE FEED	212	
3	263	63	Carriage RETURN	215	
4	264	64	SPACE	240	40
5	265	65			
6	266	66			
7	267	67			
8	270	70			
9	271	71			

*An abbreviation for USA Standard Code for Information Interchange.

APPENDIX B

IBM ASSEMBLY SYSTEM

B.1 Compile and Load Preprocessor

The following control cards are needed to compile the preprocessor and load it on the disc.

This job requires 200K of core.

```
//                EXEC     PL1LFCL,PARM.PL1L='SORMGIN=(1,72,80)',
//                         COND.LKED=(9,LE,PL1L)
//PL1L.SYSIN      DD       *
          (source of PL/1 preprocessor)
/*
//LKED.SYSLMOD    DD       DSNAME=A.EE39.PREPRO(PRE)
//                         UNIT=ONLN,DISP=(,CATLG),SPACE=(TRK,(20,10,1))
/*
```

B.2 Loading the PDP-9 Assembler Macro Library

The following control cards are required to load the PDP-9 macro library onto the disc. This macro library is used to assemble PDP-9 load modules.

This job requires 100K core and is I/O bound.

```
//              EXEC    PGM=IEBUPDTE,PARM=NEW
//SYSUT2        DD      DSNAME=A.EE39.P9MACROS,UNIT=ONLN,
//                      DISP=(,CATLG),DCB=(RECFM=FB,LRECL=80,BLKSIZE=8360),
//                      SPACE=(TRK,(65,5,50),RLSE)
//SYSPRINT      DD      SYSOUT=A
//SYSIN         DD      *

          (PDP-9 macros)

/*
```

B.3 Loading the GRIN Assembler Macro Library

The following control cards are required to load the GRIN macro library on disc. This library is used in assembly of GRIN programs.

This job requires 100K of core and is I/O bound.

```
//              EXEC    PGM=IEBUPDTE,PARM=NEW
//SYSUT2        DD      DSNAME=A.EE39.GRIN2,UNIT=ONLN,DISP=(,CATLG),
//                      DCB=(RECFM=FB,LRECL=80,BLKSIZE=8360),
//                      SPACE=(TRK,(65,5,50))
//SYSPRINT      DD      SYSOUT=A
//SYSIN         DD      *

          (GRIN2 macro library)

          (3 boxes of cards)

/*
```

B.4 Compile and Load of Postprocessor

The following control cards are needed to compile and load the PL/1 postprocessor.

This job requires 200K of core and is CPU bound.

```
//                EXEC    PL1LFCL,PARM.PL1L='SORGMIN=(1,72,80)',
//                        COND.LKED=(9,LE,P11L)
//PL1L.SYSIN      DD      *

          (PL/1 source of postprocessor)

/*
//LKED.SYSLMOD    DD      DSNAME=A.EE39.G2POST(POST),
//                        UNIT=ONLN,DISP=(,CATLG),SPACE=(CYL,(2,1,1),RLSE)
/*
```

B.5 Preprocessing, Assembling and Postprocessing a GRIN2 Program

The following control cards execute the entire assembly job. The GRIN2 source deck is included as shown below. The resulting program blocks are output on an unlabelled, 9 track, 800 BPI magnetic tape for later processing by "TAPNCH" (See B.7).

Due to the large preprocessor the following job must be run in 300K of core.

The input source deck may contain any number of individual GRIN programs or PDP-9 programs, as they are assembled in a batch mode. The postprocessor generates one program block for each GRIN program and a load module containing all the PDP-9 programs. The postprocessor requires the GRIN programs be first, in a mixed input source deck containing both PDP-9 and GRIN programs.

(Job step 'PRE')

```
//PRE            EXEC     PGM=PRE,PARM='options'
//STEPLIB        DD       DSNAME=A.EE39.PREPRO,DISP=(OLD,KEEP)
//SYSPRINT       DD       SYSOUT=A
//PUNCH          DD       SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//PREIN          DD       UNIT=DISK,SPACE=(7280,(35,4)),DISP=(NEW,DELETE),
//                        DCB=(RECFM=FB,LRECL=80,BLKSIZE=7280)
//ASMIN          DD       UNIT=DISK,SPACE=(7280,(35,4)),DISP=(NEW,PASS),
//                        DCB=(RECFM=FB,LRECL=80,BLKSIZE=7280)
//SYSIN          DD       *
```

(GRIN2 source deck)

```
/*
```

(Job step 'ASM')

```
//ASM            EXEC     PGM=ASMGASM,PARM='NOLOAD,DECK,BATCH'
//SYSLIB         DD       DSNAME=A.EE39.GRIN2,DISP=OLD
//SYSPRINT       DD       SYSOUT=A
//SYSPUNCH       DD       UNIT=ONLN,DISP=(NEW,PASS),
//                      DCB=(RECFM=FBS,BLKSIZE=1600,LRECL=80,BUFNO=1)
//SYSUT1        DD      UNIT=ONLN,SPACE=(CYL,(10,20))
//SYSUT2        DD      UNIT=ONLN,SEP=SYSUT1,SPACE=(CYL,(10,20))
//SYSUT3        DD      SEP=SYSUT1,SPACE=(CYL,(10,20)),
//                      UNIT=(2314,SEP=SYSUT2)
//SYSIN         DD      DSNAME=*.PRE.ASMIN,DISP=(OLD,DELETE)

        (Job step  'POST')

//              EXEC    SETUP,PARM='T8=PDP9(RING IN,NL,SLOT=43)'
//POST          EXEC    PGM=POST,PARM='option list'
//STEPLIB       DD      DSNAME=A.EE39.G2POST,DISP=(OLD,KEEP)
//SYSPRINT      DD      SYSOUT=A,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=1680)
//MTAPE         DD      VOL=SER=PDP9,LABEL=(,NL),UNIT=TAPE8,
//                      DCB=(RECFM=F,BLKSIZE=300,LRECL=300,DEN=2)
//PUNCH         DD      SYSOUT=B,
                        DCB=(RECFM=F,BLKSIZE=1600,LRECL=80)
//SYSIN         DD      DSNAME=*.ASM.SYSPUNCH,DISP=(OLD,DELETE)
/*
```

It is usually more economical to run the preprocessor as a seperate job and create a preprocessed source card deck. The programmer can then do any necessary corrections indicated by the preprocessor and input the preprocessed deck to the assembler in another job.

Job step 'PRE' given above can be executed in 300K of core seperately. To direct the preprocessed output to the card punch the option 'DECK' is given on the EXEC statement. That is, the first control card becomes

```
//PRE          EXEC     PGM=PRE,PARM='DECK'
```

Inputing source to the assembler from cards instead of directly from the preprocessor requires a change to the //SYSIN card in the 'ASM' job step. It now becomes

```
//SYSIN        DD       *
```

followed by the source deck.

The assembler and postprocessor require only 200K of core and are CPU bound.

B.6 Preprocessing, Assembling and Postprocessing PDP-9 Programs

To prepare PDP-9 load modules the above job control sequence is used with the GRIN2 macro library replaced by P9MACRO. This is done by changing the library reference in the //SYSLIB card of the ASM step to the following

```
//SYSLIB       DD       DSNAME=A.EE39.P9MACROS,DISP=(OLD,KEEP)
```

B.7 Preprocessor Options

The following options may be given in the parameter field of //PRE EXEC statement of the above job control sequence.

DECK - a deck of cards containing the preprocessed source is produced.

GEN - inserts a statement in each preprocessed program which causes the macro expansions to be printed out in the assembler listing.

B.8 Postprocessor Options

The following options may be given in the parameter field of the //POST EXEC statement of the above job control sequence.

ORG=nnnnn - specifies the octal starting address of absolute load modules. It is ignored on input of GRIN programs.

- nnnnn is an octal number.

- default origin value is 0.

LIST - produces a listing of PDP-9 mnemonics and contents of each word.

DECK	-	produces a deck of cards containing the object modules input to the preprocessor.
NODUMP	-	suppress the printing of the contents of each word in the program.
	-	the dump printing cannot be suppressed on PDP-9 programs.

B.9 TAPNCH: - Magnetic Tape to Paper Tape Conversion

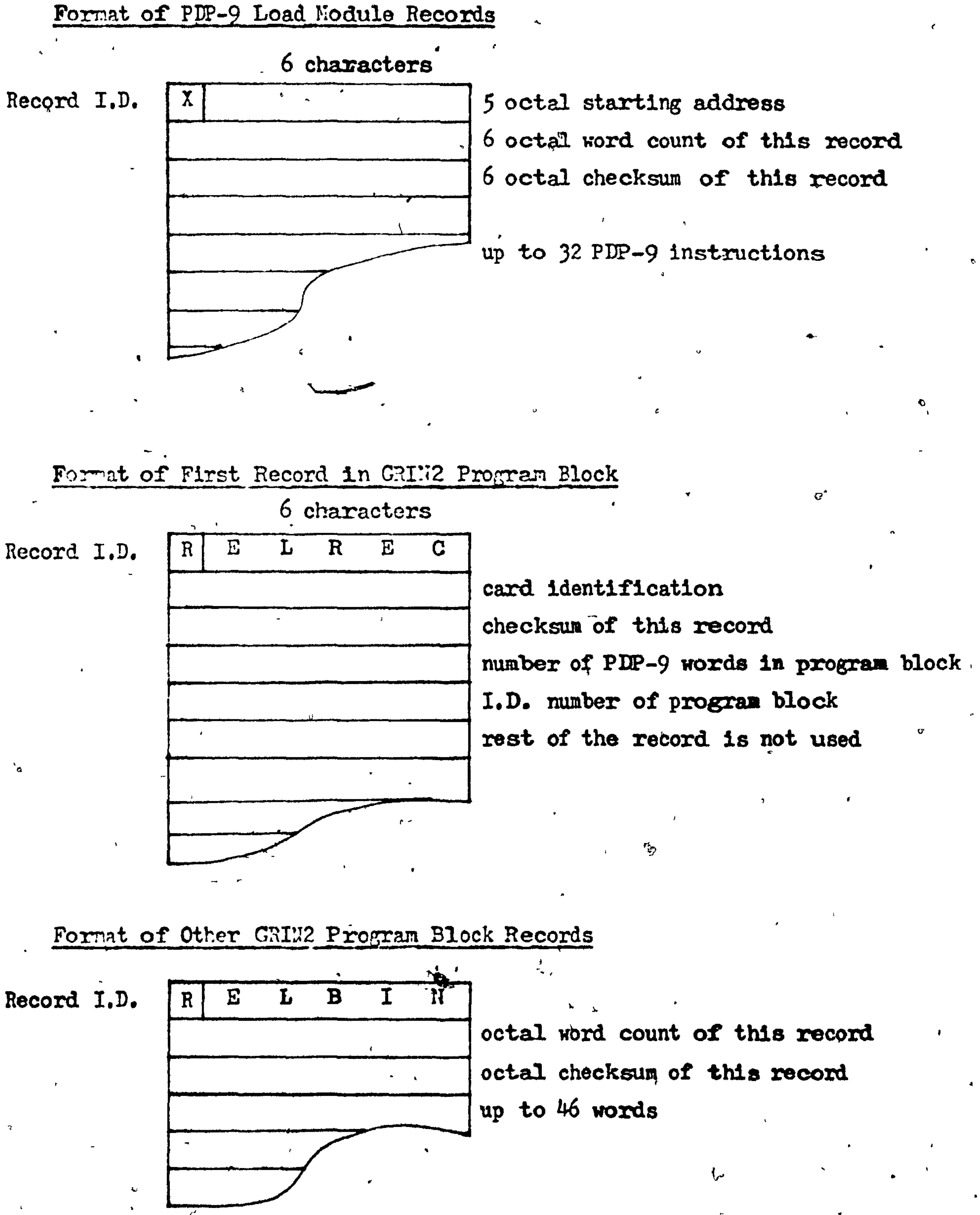
The load modules and program blocks output by the postprocessor are contained in 300 character records on an unlabelled 9 track 800 BPI magnetic tape. Each record is considered by TAPNCH as containing 50 PDP-9 words, each word being represented by 6 characters. The format of these records differ depending on whether they contain a load module segment or program block segment. The first character is used to identify the type of record as shown in Table B.1. The format of the rest of the record is given in Figure B.1..

Paper tapes in the correct format are produced using a PDP-12 with a 9 track 800 BPI magnetic tape drive, and the PS/8 operating system. Two programs are required, TAPNCH and TAPE. TAPNCH is a SABR program whose source is given in this appendix, and TAPE is a subroutine sup-

TABLE B.1

TYPES OF RECORDS ON MAGNETIC TAPE POSTPROCESSOR OUTPUT

First Character in Record	Type of Record
L	Record contains bootstrap loader.
B	Record is part of a load module.
E	Last reoord of load module.
R	One record of a GRIN program block.
T	Last record of GRIN program block.
G	End of a batch of GRIN program blocks.



Format of PDP-9 Load Module Records

6 characters
Record I.D.
X
5 octal starting address
6 octal word count of this record
6 octal checksum of this record
up to 32 PDP-9 instructions

Format of First Record in GRIN2 Program Block

6 characters
Record I.D.
R E L R E C
card identification
checksum of this record
number of PDP-9 words in program block
I.D. number of program block
rest of the record is not used

Format of Other GRIN2 Program Block Records

Record I.D.
R E L B I N
octal word count of this record
octal checksum of this record
up to 46 words

FIGURE B.1. FORMAT OF THE POSTPROCESSOR'S OUTPUT.

plied by the Montreal Neurological Institute for driving the magnetic tape transport. Compiling, loading and executing SABR programs is explained in DEC's PS/8 Operating Manual.

The program TAPNCH starts by printing

TYPE A 1 OR 0

A '1' response causes the entire contents of the magnetic tape to be output on the teletype. A '0' response causes only the record number and its word count to be printed. A paper tape is punched regardless of the option. Successful completion of the job is signalled by the message

DONE.

B.10 Error Messages

The following errors may occur when using TAPNCH.

(1) BAD 1st CHAR

The first character of every record must be a L, G, T, B, or E. Any other character will stop 'TAPNCH' and print this message. Most probable cause is incorrect operation of the postprocessor.

(2) READ ERROR

An error in attempting to read the magnetic tape occured if this message is printed. This is caused by a malfunction in the magnetic tape transport, or an error in its handler TAPE.

(3) EOF

An EOF mark was encountered before the logical end of the program on the tape, when attempting to read the tape. This is usually caused by forgetting to rewind the tape after being read once.

SOURCE OF TAPNCH

```
C
C   THIS PROGRAM TRANSFERS PDP-9 OBJECT CODE FROM MAGTAPE
C   TO PAPER TAPE
C
          COMMON LREC
          DIMENSION LREC(3ØØ),LOUT(3),IN(6),ITTY(6)
          READ(1,15)ISUP
15        FORMAT('IYPE 1 OR Ø',I1)
          IREC=Ø
S         CLA      /START PUNCH
S         PLS
C   PUNCH LEADER TRAILER
5Ø        DO 1Ø1  I=1,1Ø
S         JMS PNCH
1Ø1       CONTINUE
C
C   READ MAGNETIC TAPE
C
          NERR=2
          I=Ø
          IREC=IREC+1
          WRITE(1,12)IREC
12        FORMAT(1ØX,'RECORD NO.',I6)
          CALL TAPE(6,3ØØ,LREC(1),NERR)
          IF(NERR) 2ØØ,2Ø1,2Ø2
C   ERROR CODES (NERR)  ARE:
C    1  ILLEGAL 1ST ARG OF TAPE SUBROUTINE
C   -1  AN EOF WAS FOUND ON THE TAPE
C 512 TRANSMIT ERROR
2Ø1       LSTRT=LREC(1)
          I=1
S         JMS EBCASC        /CONVERT EBSDIC TO ASCII & TYPE
C   1ST CHARACTER = 'R' (217) FLAGS A RELOCATABLE OBJECT TAPE
C   1ST CHARACTER = 'T' (227) FLAGS END OF RELOCATABLE PROGRAM
C   1ST CHARACTER = 'G' (199) FLAGS END OF RELOCATABLE ASSEMBLY
C   1ST CHARACTER = 'L' (211) FLAGS READ-IN LOADER
C   1ST CHARACTER = 'B' (194) FLAGS ABSOLUTE OBJECT TAPE
C   1ST CHARACTER ='E' (197) FLAGS END OF ABSOLUTE TAPE
          IF(LSTRT-199)47,416,47
47        IF(LSTRT-217)48,7ØØ,48
48        IF(LSTRT-227)49,7ØØ,49
C         IGNORE RECORDS WITH WRD CNT>32
49        IF(LREC(7)-247)5Ø,51,5Ø
51        IF(LSTRT-194) 316,32Ø,316
316       IF(LSTRT-197) 317,32Ø,317
317       IF(LSTRT-211) 322,321,322
322       GO TO 32Ø
32Ø       IN(I)=Ø
S         JMS ASCBIN        /CONVERT TO BINARY AND PUNCH
7ØØ       I=7
S         JMS EBCASC        /CONVERT -WORD COUNT & TYPE
S         JMS ASCBIN        /CONVERT TO BINARY AND PUNCH
S         CLA
S         TAD \INNI         /GET WORD COUNT
S         AND (ØØ77
S         SNA      /CHECK FOR ZERO WRD COUNT
```

```
S       JMP ZRO
S       DCA \KWC
        IF(LSTRT-217)701,702,702
701     CONTINUE
S       CLA
S       TAD \KWC
S       TAD (7700
S       CIA     /WORD COUNT IS ANEGATIVE NUMBER
SZRO,   DCA \KWC        /AND PUT IN 'DO' LOOP LIMIT
702     WRITE(1,11)KWC
11      FORMAT(10X,'WRD CNT',I5)
        KWC=KWC*6+18
        I=13
S       JMS EBCASC      /CONVERT -CHECKSUM
S       JMS ASCBIN      /PUNCH -CHECKSUM
C   PUNCH BODY OF TAPE BLOCK
401     DO 420 I=19,KWC,6
S       JMS EBCASC
S       JMS ASCBIN
420     CONTINUE
        IF(LSTRT-211) 410,411,410
410     IF(LSTRT-194) 412,50,412
412     IF(LSTRT-217)413,50,413
413     IF(LSTRT-227)417,750,417
417     IF(LSTRT-197) 322,414,322
C   END OF TAPE
414     CONTINUE
S       CLA
S       TAD (277
S       JMS PNCH        /PUNCH BINARY TRANSFER BLOCK
S       JMS PNCH
S       JMS PNCH
S       JMS PNCH
S       JMS PNCH
S       JMS PNCH        /PUNCH END OF TAPE CODE
S       CLA
S       TAD (200
S       JMS PNCH
S       JMS PNCH
S       JMS PNCH
416     WRITE(1,2)
2       FORMAT('DONE')
        STOP
319     WRITE (1,3)
3       FORMAT('BAD 1ST CHAR')
        STOP
750     DO 751 I=1,30
S       JMS  PNCH
751     CONTINUE
        GO TO 50
C   IF 1ST CHARACTER IS AN 'L' (211) THIS RECORD CONTAINS
C   THE READ IN LOADER
C   PUNCH END CODE FOR THE READ IN LOADER
411     CONTINUE
```

```
S       CLA
S       TAD (0261
S       JMS PNCH
S       CLA
S       TAD (0277
S       JMS PNCH
S       CLA
S       TAD (0375
S       JMS PNCH
        GO TO 50
321     KWC=31
        WRITE(1,11)KWC
        KWC=KWC*6+18
        GO TO 401
200     WRITE(1,4)
4       FORMAT('EOF')
        GO TO 414
202     WRITE(1,5)
5       FORMAT('READ ERROR')
        STOP
SPNCH,  0       /PNCH ONE FRAME
SPCK,   PSF
S       JMP PCK
S       PLS
S       JMP I PNCH      /YES
C
SASCBIN,        0       /ASCII TO BINARY & PUNCH ROUTINE
S       7621    /CLR AC AND MQ
        DO 500 K=1,6,2
        INN=IN(K)
        INN1=IN(K+1)
S       TAD \INN
S       AND (0007
S       7413    /SHIFT LEFT 3
S       2
S       DCA \INN
S       TAD \INN1
S       AND (0007
S       TAD \INN
S       TAD (200
S       JMS PNCH
S       DCA \INN1
500     CONTINUE
S       JMP I ASCBIN
C   EBSDIC TO ASCII SUBROUTINE
SEBCASC,        0
        JJ=I+5
        JK=0
        DO 600 J=I,JJ
        JK=JK+1
        INN1=LREC(J)
S       CLA CLL
S       TAD \INN1
S       AND (0077
S       TAD TABAD
S       DCA TEMP
S       TAD I TEMP
S       DCA \INN1
        INC(JK)=INN1
S       7621    /CLEAR AC AND MQ
S       TAD \INN1
S       7413    /SHIFT LEFT 6
S       5
S       DCA \INN2
600     ITTY(JK)=INN2
        IF(ISUP)602,603,602
602     CONTINUE
        WRITE(1,601)ITTY
601     FORMAT(6A1)
603     CONTINUE
S       JMP I EBCASC
STEMP,  0
STABAD, LOOKUP  /LOOK UP TABLE ADDRESS
S       PAGE
C       CHARACTER CONV TABLE FROM EBCDIC TO ASCII
SLOOKUP,        240     /SPACE
S       301  /A
S       302  /B
S       303  /C
S       304  /D
S       305  /E
S       306  /F
S       307  /G
S       310  /H
S       311  /I
S       240  /NO EQUIV
S       256  /.
S       274  /<
S       250  /(
S       253     /+
S       240     /NO EQUIV
S       246  /&
S       312  /J
S       313  /K
S       314  /L
S       315  /M
S       316  /N
S       317  /O
S       320  /P
S       321  /Q
S       322  /R
S       241  /!
S       244  /$
S       252  /+
S       251  /)
S       273  /;
S       236  /↑
S       255  /-
```

```
S          257  /SLASH
S          323  /S
S          324  /T
S          325  /U
S          326  /V
S          327  /W
S          330  /X
S          331  /Y
S          332  /Z
S          240  /NO EQUIV.
S          254  /,
S          245  /%
S          255  /-
S          276  />
S          277  /?
S          260  /1
S          261  /1
S          262  /2
S          263  /3
S          264  /4
S          265  /5
S          266  /6
S          267  /7
S          270  /8
S          271  /9
S          272  /:
S          243  /
S          240  /NO EQUIV
S          274  /
S          275  /=
S          242  /"
S          240
           END
```

APPENDIX C

SYNTAX DIFFERENCES BETWEEN GRIN2 AND McGRIN

C.1 Introduction

GRIN2 programs require some preprocessing before they can be assembled by the IBM 360 assembly system. It is suggested that new GRIN programs written at McGill be coded in the preprocessed syntax thereby avoiding this expensive and time consuming job step. The GRIN2 dialect defined by the preprocessed syntax is called McGRIN.

A complete description of GRIN2 and its syntax is given in the BELLGRAPH Programmer's Manual. The rules given below list the alterations and restrictions of the GRIN2 syntax imposed by the McGRIN dialect. By following these rules when writting GRIN programs, the programer does not need to preprocess his source deck.

C.2 McGRIN Macros

Macros are defined in McGRIN programs using the BAL macro processing facility. A description of BAL macros is given in Sections 8 through 10 of the OS Assembler Language Manual (IBM #GC28-6514-8).

C.3 Blank Operand Field

It is standard practice to define the statement field boundaries as follows:

Columns 1 - 7	Label field
Columns 8 - 15	Operation field
Columns 16 - 30	Operand field
Columns 31 - 79	Comments

Column 16 must not be blank. If there is no operand field a comma should be placed in column 16.

C.4 GRIN Function Statement Argument Lists

Unlike GRIN2, McGRIN requires a comma delimitating all arguments and subarguments. The example below illustrates this.

GRIN2	McGRIN
`TEXT  8, .UC(AN ARROW)`	`TEXT  8, .UC, (AN ARROW)`
`CURVE ((0, W)(W,0)(0,-W)(-W,0))`	`CURVE ((0,W),(W,0),(0,-W),(-W,0))`

C.5 ASCII Character Set

The McGRAPH software system supports the standard 64 character ASCII set. All lower case text (indicated by .LC in text arguments) is assembled into its correct ASCII representation by the McGRIN assembler, but is displayed in upper case by the McGRAPH character generator.

C.6 ASCII Text Arguments

The text arguments of TEXTIN, ASCII, TEXT, TYPOUT and BCD must conform to the following rules. (See page 2.1-70 in the BELLGRAPH Programmer's Manual). Text strings arguments of the form

k, .UC, (string) or k, .LC, (string)

must not contain .

(a) parentheses

(b) spaces (the character "¬" is translated as a space

(c) commas .

Also, the text string must be delimited by parentheses.

C.7 Modified Operation Mnemonics

The following GRIN2 operation mnemonics must be preceded by 'Q'.

MR OR ENTRY START END*

ADD NOP CLC LX

for example, NOP becomes QNOP.

*NOTE: END becomes QEND
END

BIBLIOGRAPHY

1. Sutherland, I.E., "Sketchpad: A Man-Machine Graphical Communication System", Proceedings AFIPS, Spring 1963.

2. Sutherland, I.E., "Computer Graphics", Datamation, Vol. 12, pp. 22-27, May 1966.

3. Sutherland, I.E., "Computer Displays", Scientific American, Vol. 222, pp. 57-81, June 1970.

4. Stack, T.R. and Walker, S.T., "AIDS - Advanced Interactive Display System", Proceedings of AFIPS, Vol. 38, pp. 113-121, Spring 1971.

5. Sutherland, W.R., "On-Line Graphical Specifications of Computer Procedures", Tech. Rep. 405, Lincoln Lab MIT, May 1966.

6. Ninke, W.H., "GRAPHIC-1: A Remote Graphical Display Console System", Proceedings of AFIPS, Fall Joint Computer Conference, Vol. 27, pp. 834-846, 1965.

7. Devere, G.S., Hargreaves, B., Walker, D.M., "The DAC-1 System", Datamation, Vol. 12, pp. 37-47, June 1966.

8. Lidinsky, W.P., "MIRAGE: A Microprogrammable Interactive Raster Graphics Equipment", Proceedings of the IEEE International Computer Society Conference, pp. 15-16, 1971.

9. Gray, J.C., "Compound Data Structure for Computer-Aided Design; A Survey", Proceedings of A.C.M. National Meeting, pp. 355-364, 1967.

10. Williams, R., "A Survey of Data Structures for Computer Graphics Systems", Computing Surveys of the A.C.M., Vol. 3, No. 1, pp. 1-21, March 1971.

11. Christensen, C. and Pinson, E.N., "Multi-Function Graphics for a Large Computer System", Proc. AFIPS, Fall Joint Computer Conference, Vol. 31, pp. 697-771, 1967.

12. Cotton, I., and Greatorex, F.S., Jr., "Data Structures and Techniques for Remote Computer Graphics", Proc. AFIPS, Fall Joint Computer Conference, Vol. 33, Pt. 1, pp. 533-544, 1968.

13. Hurwitz, A., "GRAF: Graphic Additions to FORTRAN", Proc. AFIPS, Spring Joint Computer Conference, Vol. 30, pp. 553-557, 1967.

14. Bracchi, G. and Ferrari, D., "A Language for Treating Geometric Patterns in a Two-Dimensional Space", Comm. of A.C.M., Vol. 14, pp. 26-32, Jan. 1971.

15. Miller, W.F. and Shaw, A.C., "Linguistic Methods in Picture Processing - A Survey", Conf. Proc. AFIPS, Fall Joint Computer Conference, Vol. 33, Pt. 1, pp. 279-290, 1968.

16. Kulsrud, H.E., "A General Purpose Graphic Language", Comm. of A.C.M., Vol. 11, pp. 247-254, April 1968.

17. Rully, A.D., "A Subroutine Package for FORTRAN Interactive Graphics in Data Processing", IBM Systems Journal, Vol. 7, pp. 248-256, 1968.

18. Ledley, R.S., "BUGSYS: A Programming System for Picture Processing - Not for Debugging", Comm. of the A.C.M., Vol. 9, pp. 79-84, February 1966.

19. Shaw, A.C., "The Formal Description and Parsing of Pictures", Ph.D. Thesis, SLAC Report No. 84, Stanford Linear Accelerator Center, Stanford, California, 1968.

20. Roberts, L.G., "A Graphical Service System with Variable Syntax", Comm. of the A.C.M., Vol. 9, pp. 173-175, March 1966.

21. Wagner, F.V. and Laltood, J., Computer Graphics: Software Design, Computer Graphics, Graphics Symposium University of California, Academic Press, pp. 99-135, 1966.

22. Ninke, W.H., "Interactive Computer Graphics: Some Interpolations and Extrapolations", Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Graphics, University of Illinois Press, pp. 429-439, 1969.

23. Baskin, H.B., "A Comprehensive Applications Methodology for Symbolic Graphics", Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Computer Graphics, University of Illinois Press, pp. 414-428, April 1969.

24. Sibley, E.H., "The Use of a Graphic Language to Generate Graphics Procedures", Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Computer Graphics, University of Illinois Press, pp. 390-413, April 1969.

25. Chen, F.C., and Dougherty, R.L., "A System for Implementing Interactive Applications", Interactive Graphics in Data Processing, IBM Systems Journal, Vol. 7, pp. 257-270, 1968.

26. Yarbrough , L.D., "CAFE: A Non-Procedural Language for Computer Animation", Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Computer Graphics, University of Illinois Press, April 1969.

27. Brown, S.A., "A Description of the APT Language", Comm. of the A.C.M., Vol. 6, pp. 649-658, November 1963.

28. Ledley, R.S., and Ruddle, F.H., "Chnomosome Analysis by Computer", Scientific American, pp. 40-46, April 1966.

29. Streit, Edward, "VIP: A Conversational System for Computer-Aided Graphics", Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Computer Graphics, University of Illinois Press, pp. 224-260, April 1969.

30. Fabi, R., "The Design and Construction of a Disc Oriented Graphics System", Thesis Department of Electrical Engineering, McGill University, February 1971.