1

OF/DE

3

24228

National Library
of CanadaBibliothèque nationale
du CanadaCANADIAN THESES
ON MICROFICHETHÈSES CANADIENNES
SUR MICROFICHENAME OF AUTHOR/NOM DE L'AUTEUR C. Douglas O'BrienTITLE OF THESIS/TITRE DE LA THÈSE IMAGE: A language for the Interactive Manipulation of
A Graphics Environment.UNIVERSITY/UNIVERSITÉ Carleton UniversityDEGREE FOR WHICH THESIS WAS PRESENTED/
GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE M.Eng.YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE DEGRÉ 1975NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE Professor R.J.A. Buhr, Department of Systems Engineering

Permission is hereby granted to the NATIONAL LIBRARY OF
CANADA to microfilm this thesis and to lend or sell copies
of the film.

*L'autorisation est, par la présente, accordée à la BIBLIOTHÈ-
QUE NATIONALE DU CANADA de microfilmer cette thèse et
de prêter ou de vendre des exemplaires du film.*

The author reserves other publication rights, and neither the
thesis nor extensive extracts from it may be printed or other-
wise reproduced without the author's written permission.

*L'auteur se réserve les autres droits de publication; ni la
thèse ni de longs extraits de celle-ci ne doivent être imprimés
ou autrement reproduits sans l'autorisation écrite de l'auteur.*

DATED/DATE May 1 / 75 SIGNED/SIGNÉ C Douglas O'BrienPERMANENT ADDRESS/RÉSIDENCE FIXE C. Douglas O'Brien115 - 1435 Prince of Wales DriveOttawa, Ontario.

I M A G E:
a language for the
Interactive Manipulation of
A Graphics Environment

by

C. Douglas O'Brien

A thesis submitted to the Faculty of
Graduate Studies in partial fulfilment
of the requirements for the degree of
Master of Engineering

Faculty of Engineering

Carleton University

Ottawa, Ontario

May 1, 1975

C. Douglas O'Brien

1975

The undersigned recommend to the Faculty
of Graduate Studies acceptance of the thesis
" IMAGE, a language for the Interactive
Manipulation of A Graphics Environment "
submitted by C. Douglas O'Brien, B.Eng.,
in partial fulfillment of the requirements
for the degree of Master of Engineering.

[Signature]
.....
Thesis Supervisor (Carleton U)

[Signature]
.....
Thesis Supervisor (external)

[Signature]
.....
Chairman,
Department of Systems Engineering

Carleton University

May 1, 1975

ABSTRACT

This thesis addresses itself to the problems involved in programming an interactive graphics display. A survey is presented of several representative languages commonly used for writing interactive graphics programs, and a list of programming facilities considered necessary for an interactive graphics programming language is presented. An examination of several application programs, written in a variety of existing languages, revealed that many of these facilities are usually lacking. A new language 'IMAGE' has been designed to provide an application programmer convenient access to these facilities.

The 'IMAGE' language places particular emphasis on providing a graphics application programmer with the ability to program graphical interaction. It utilizes the better features of several current graphics languages and combines these features with a unique interaction control structure. This OBJECT / ACTION control structure, the display picture description syntax and the hardware independent handling of input devices are the main features of the language, providing excellent graphical input response and drawing facilities. The device independent input / output structure permits the implementation of a portable language syntax, since there are no references to particular display hardware devices. All display references are performed through a virtual terminal.

The available techniques with which to implement this language, and the effects they would have on the language syntax and structure, are considered. Compiler portability and the concept of a virtual graphics terminal are discussed.

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to Mr. H.G.Bown and to Dr. R.J.A.Buhr for their guidance and constructive criticism which continued throughout this study and to Mr. G.Thorgeirson and Mr. R.E.Warburton for many stimulating discussions. The author is indebted to the Communications Research Centre of the Canadian Federal Department of Communications for the use of their facilities.

1. INTRODUCTION	1
2. SURVEY OF SEVERAL REPRESENTATIVE GRAPHIC LANGUAGES	3
2.1 TEST PROBLEM 1 : A SIMPLE LINE EDITOR	4
2.2 TEST PROBLEM 2 : A CALCULATION PROBLEM	5
2.3 A COMPARISON OF GRAPHICS LANGUAGES	5
2.3.1 ASSEMBLER (D.E.C. PDP 9/15 -MACRO)	7
2.3.2 FORTRAN	9
2.3.3 ICPL (Interactive Control Program Language)	12
2.3.4 GRAPPLE (Graphical Application Programming Lang)	14
2.3.5 GRAPHICS APL	18
2.3.6 BASIC	20
2.3.7 EULER/G	22
2.3.8 DIAL	24
3. REQUIREMENTS FOR A GRAPHICS PROGRAMMING SYSTEM	27
4. THE DESIGN OF A NEW GRAPHICS LANGUAGE, 'IMAGE'	29
4.1 GENERAL PROGRAM FORMAT and SYNTAX DEFINITION	31
4.2 DEFINITIONAL STATEMENTS	35
4.2.1 BLOCK DEFINITION STATEMENTS	35
4.2.2 DECLARATION STATEMENTS	36
4.3 EXECUTABLE STATEMENTS	40
4.3.1 DISPLAY EDITING COMMANDS	42
4.3.2 GRAPHICAL DRAWING INSTRUCTIONS	47
4.3.3 GRAPHICAL DRAWING MODIFIERS	51
4.3.4 PROCEDURES AND EXTERNAL SUBPROGRAMS	61
4.3.5 INTERACTION CONTROL	66
4.3.5.1 IDENTIFYING	70
4.3.5.2 SKETCHING	72
4.3.5.3 POSITIONING	75
4.3.5.4 INPUT OF TEXTUAL STRINGS	79
4.3.5.5 PUSHBUTTONS	82
4.3.5.6 VALUATORS	83
4.4 AN EXAMPLE PROGRAM	86
5. IMPLEMENTATION CONSIDERATIONS	88
5.1 PORTABILITY	89
5.2 RUN-TIME SYSTEM CONFIGURATION	99
6. CONCLUSIONS	104
REFERENCES	106
APPENDIX A -- SUMMARY OF COMMANDS	109
APPENDIX B -- ASSIGNMENT STATEMENTS	119
APPENDIX C -- EXECUTION FLOW CONTROL	126
APPENDIX D -- GRAPHICS SUBROUTINE LIBRARY	133
APPENDIX E -- GTI COMMUNICATIONS PROTOCOL	138
APPENDIX F -- THE 'STAGE2' MACRO PROCESSOR	143
APPENDIX G -- TWO EXAMPLE PROGRAMS IN A VARIETY OF LANG.	149

1. INTRODUCTION

An interactive computer graphics system is a system which permits a 'conversation' to ensue between the user and the computer, where the communications medium is a computer-driven display. Potentially a larger amount of information may be displayed in a single picture on the display screen than can be presented in several pages of line printer listings. Because this information is presented in picture format, it can make use of the eye's ability to detect two-dimensional spatial relationships, and therefore provide a great improvement over the amount of information which can be communicated in a one-dimensional stream of printed text. Rapid input to the computer can be achieved with the graphics display by using this two-dimensional capability to allow a user to address information. The concept of a 'light-button' is based on this spatial addressability. By placing a light-pen or equivalent device on the display screen at a particular location, a user can indicate a particular symbol or string of text and can cause an action to occur conditional on the state of the program. It is the complexities of the interactive dialogue, especially control of the 'light-button' construct, which create the major difficulties in programming interactive graphics programs.

Conventional programming languages are executed serially, instruction by instruction, with the execution flow controlled by conditional statements. Although it is possible to write a graphics application program in this serial fashion it can be quite awkward to do so. The highly interactive nature of a graphics program lends itself to more specialized forms of

programming. In the following chapters, a representative group of existing languages are analyzed for their potential for programming graphics applications. The languages analyzed are:

ASSEMBLER (DEC PDP 9/15 MACRO) - an assembler language directly controlling graphics hardware.

FORTTRAN (DEC PDP 15 - VT 15-) and (GINO / F) - FORTRAN implementations augmented by a sets of graphic subroutine calls.

ICPL (Interactive Control Program Language) - a specialized graphics language

GRAPPLE (Graphics Application Programming Language) - a specialized graphics language

GRAPHICS APL - an implementation of APL with specialized graphics functions

BASIC incorporating graphics (DEC PDP 11 BASIC/GT) - an implementation of BASIC with specialized graphics commands.

EULER/G (Univ. of Utah, PDP 410) - an extensive addition to EULER to incorporate graphics. EULER is itself a generalization of ALGOL 60.

DIAL - a specialized graphics language

Two relatively different example programs were written in those languages for which a translator was locally available in order to determine the difficulties in programming interactive graphics programs. The programming facilities which aided in the writing of graphics programs were identified. This thesis presents the design of a new interactive graphics language specifically designed in order to provide these facilities to the application programmer. The best features of the surveyed languages along with a unique interaction control structure, a display procedure based picture description grammar and device independent input facilities were combined to form the IMAGE language.

2. SURVEY OF SEVERAL REPRESENTATIVE GRAPHIC LANGUAGES

This chapter compares several representative languages used for interactive computer graphics. Two example problems have been programmed in those languages which were locally available and the language-dependent difficulties are described. The two test problems endeavour to cover the range of graphics interaction problems, from interactive manipulation of a data base to a calculation and presentation problem.

In one test problem a simple data base consisting of the starting and end co-ordinates of displayed lines is maintained, and this data base is edited through interaction with a light-pen or similar device. Interaction was difficult to program in some languages, while others handled it with ease. Some languages required the use of external routines written in another language to maintain the data base. This is not necessarily bad, since a complicated data base may require the use of a specialized data base language. What is important is the ability to access external programs written in other languages so that facilities such as specialized data types may be accessed. Most languages possess this feature, but in some the manner of passing parameters to a foreign subroutine is complicated.

In the other problem a curve is calculated and displayed from data entered via the teletype. The only interaction via a light-pen or similar device is to initiate the starting and

clearing of the screen and the selection of which data is to be entered. Not all of the languages could handle the calculation but in some, external routines written in another language were called to perform it.

The two test problems were selected to reflect, as much as possible, the requirements that any application may impose on a graphical programming system. Only short, test problems were chosen due to the requirement of writing multiple copies of each. Also, one of the main considerations in selecting the test problems was the desire to evaluate the performance of each programming system in handling an interactive situation such as could be imposed by a typical interactive graphics application. Refer to Figures G1 and G2 for the general layout of the screen during execution of the test programs and to Appendix G for the test program coding.

2.1 TEST PROBLEM 1 : A SIMPLE LINE EDITOR

The simple graphics line editor program permits a user to input, delete or move lines visible on the display screen. A line is created by positioning the marker to a chosen screen location, selecting the light-button 'START CO-ORD', re-positioning to the desired end of the line and selecting 'END CO-ORD'. Similar operations allow one to move and delete lines. A line will be moved to the position of the marker when the 'MOVE CO-ORD' and then the 'MOVE' light-buttons are selected. In order to delete a line, the line is indicated and then the

'DELETE' light-button is touched. This test problem is very simple, but it does illustrate the essential difficulties encountered in programming an interactive problem involving a data base. The essence of the task is to create and identify lines on the display device and to manipulate their representation in a data base.

2.2 TEST PROBLEM 2 : A CALCULATION PROBLEM

This harmonic synthesis demonstration problem illustrates on the display screen the superposition of harmonically related sine waves of varying amplitudes and phases. A menu of light-buttons is provided to initiate the display of the graph. This test problem involves the manipulation of character strings, scientific calculations and the presentation of a graph on the display screen. This is basically a graphical output problem with very simple input interaction. Most of the effort required is in the scientific calculations which can be handled by external routines more appropriately fitted for the task.

2.3 A COMPARISON OF GRAPHICS LANGUAGES

A general description of some languages used for interactive computer applications is presented in the following section. With each description is an analysis of the capability of that language to perform the test problem based on an analysis of the code presented in Appendix G. The languages which were chosen

range from specialized high-level graphics languages to a typical assembler language. The use of assembler language cannot be ignored because assembler language has the power to perform any function efficiently, but it is low-level and tedious to write. Many large graphics applications packages have been programmed entirely in assembler language. Where a translator, assembler or compiler was available, the test programs were written in that language; otherwise, only example sections of code were written. Executable versions of the two test programs exist for ICPL [1,2,3], GRAPPLE [4,5], the DEC PDP15 FORTRAN graphics system [6], and for assembler language on the PDP9 [7] computer.

Having only programmed two test programs per language, there is insufficient data to produce statistics on the relative merits of each language, so only qualitative impressions have been obtained. Some of the factors which affect the relative merit are the length of the program to perform a given task, the programming time, the compile time, and the execution time. Two other important but more subjective factors are the readability of the code and its error proneness. One can be more confident that all the bugs have been removed from a simple, clear, readable program than from a large, awkward piece of code. Since the four languages mentioned above were executed on several different machines, it is impossible to compare runtime behaviour.

It took only a few pages of code to write the test programs in the two graphics-oriented languages GRAPPLE and ICPL, while it took in the order of ten pages to write them using the D.E.C. FORTRAN package. The assembler language programs were only 30% longer than the FORTRAN programs, and it took longer to debug the FORTRAN programs. The rest of this chapter will describe several languages and attempt to evaluate them for programming graphics.

2.3.1 ASSEMBLER (D.E.C. PDP 9/15 -MACRO)

This assembler language [7] is a relatively straight-forward language providing access to the machine instructions of a single-accumulator single-address machine with 8K memory pages, indirect addressing, but no indexing. This language is similar to one described by Newman and Sproule in Appendix III of their book [8]. A macro capability exists to ease programming and the graphical primitives are described in the following macros:

UTEXT, N, CHARTAB	- for putting N upper-case characters on the screen
SETXY, X, Y	- for setting the beam to absolute screen location (X,Y)
VECTOR, DELX, DELY	- for drawing a relative vector
INTY, N	- set the beam intensity
SEEK	- wait for light-pen interrupts

This language provides a graphical input and output facility and a low-level, but highly general, capability of performing interactive and conditional operations. Because of the high

machine dependence of an assembler language there is almost no portability.

Graphical input response facilities are provided by utilizing a dispatch table to an appropriate action routine upon an interrupt. Except for the length of the code which had to be written, very few problems were experienced in programming graphical input on the assembler language level. This is due to the direct control of the machine's interrupt hardware which was available. On the other hand, the graphical output facilities required string manipulation which resulted in complicated character packing and unpacking. The data manipulation facilities provided by the assembler language are at a very low level, but are powerful enough to allow compact word packing of data. Assembler language provides very awkward access to external software because of the tedious task of parameter passing. The calling of a FORTRAN program from this assembler language involved multiple-level indirect addressing. The major difficulties with assembler language programming are the large amount of code required and the complexity in maintaining flags. Because of the lack of structure (branch statements and address pointers), assembler language is very difficult to debug. This assembler language is typical and one would expect similar programming problems would be encountered with others.

2.3.2 FORTRAN

A common method of providing a graphics capability is to supply a package of subroutines to be called from a high-level host language. There are many such packages available through FORTRAN on different computer installations. Most are highly device- and machine-dependent and provide non-standardized functions. The FORTRAN IV graphics package [6] operating on the PDF-15 VT-15 graphics hardware is a set of subroutine calls imbedded in the FORTRAN language. These calls are quite powerful but very awkward to use because of the long and variable argument strings. A capability to build structured dynamic display files is provided; in fact, one must program using them. Displayable code is generated in graphical display file subroutines (subpictures) stored as data in FORTRAN arrays and these are linked together by a main display file. There are different calling conventions required to generate the same graphical entities for a main display file or a subpicture. An example of the functions available are:

for a mainfile	for a subpicture
CALL PLOT (1, DELX, DELY, INT)	CALL LINE (DELX, DELY, 1)
- Plot a line (or re-position the beam) by amount DELX, DELY, with the beam intensified (INT = 1).	
CALL PLOT (3, STRING, N)	CALL TEXT (STRING, N)
- Plot a STRING of text N characters long.	
CALL PLOT (2, PRAM,)	CALL PRAMTR (PRAM,)
- set up one or many display hardware parameters such as beam intensity, where PRAM is a code number. For example, code number 256 means synchronize the display hardware to the line frequency.	

This system provides the good arithmetic capabilities of FORTRAN along with its adequate control structure and data types as well as a graphics drawing capability. It is highly machine-dependent because of a great reliance on special hardware features and is awkward to program.

The code for the two test cases was long and difficult to debug using the DEC FORTRAN system because of the inadequacy of the graphics functions supplied. String manipulations in FORTRAN were awkward and tedious, and the non-standard ENCODE-DECODE statements had to be used to convert numbers to character strings for display purposes. The subpicture capability provided by the system was of little use because the two test problems required the display of few duplicate symbols. However, the necessary use of these subpictures was a hindrance because of the different set of display function calls required for the main display file and for subpictures. The variable parameter list feature in function calls resulted in many errors.

The identification of user picture interrupts is provided by a "name register" available through a parameter call. This name register is used to associate an item or subpicture with a particular interrupt. The number associated with a particular item is returned upon an interrupt. The name register facility provides for up to 128 individually named items and programming for greater than this number of identifiable items is, very awkward. The line editor is a very simple, small program but it

almost surpassed this limit. This is a serious fault with this particular system, but is not a general fault with FORTRAN graphics packages. The programmer was not shielded from a hardware level complication. Because the base language was FORTRAN, all its computational, array manipulative and subroutine calling capabilities were available and were well utilized.

In comparison, another FORTRAN based subroutine package is GINO/F [9]. The GINO/F system contains routines for producing graphs, drawing 2D pictures, defining and manipulating 3D objects, means of interaction and program administration. It provides device independence by a call to specify the type of the device, and this invokes a set of device handlers. GINO/F provides a large number of functions of the form:

CALL MOVT0(x,y)	-	Move pen to point x,y
CALL LINBY(x,y)	-	Draw a line with vector increments x,y
CALL CHAASC(nchar)	-	Output the character whose code is 'nchar'.

GINO/F contains facilities which allow interaction through a keyboard or a cursor, but it doesn't provide control over interrupt driven devices such as a light-pen and therefore only polled menu selection techniques may be used. As with all FORTRAN based systems any interrupt interaction is difficult.

Both the DEC FORTRAN package and GINO/F share the common problem of being difficult to use to program graphical interaction, and both are deficient in being able to manipulate

character strings. Programs which consist of mostly a series of subroutine calls with long strings of parameters for each call, tend to be rather long and obscure because the meaning of the program resides in the numeric parameters. /

2.3.3 ICPL (Interactive Control Program Language)

The ICPL [1,2,3] graphics language is essentially a control program language of the action-reaction type, similar to that proposed by Newman [10]. The control program is written in the form of OBJECT / ACTION pairs, with each ACTION associated with an OBJECT. The program is parallel in nature and the OBJECT / ACTION pairs require no special ordering. A menu of light-buttons can easily be defined as a series of OBJECT statements and the associated ACTION statements identify the code to be executed when an interrupt-driven command such as a light-pen strike occurs. This technique makes the state transitions invisible to the programmer. It completely removes the graphics programmer from interrupt level programming and forces him to think of displayable objects and actions resulting from identified strikes upon these objects.

ICPL has a poor mechanism for organizing program structure. It does not have good constructs for iterative or conditional operations. The only data manipulation facilities available are integer numbers and text strings and no general data structure capabilities are provided.

The graphical input response facilities offered by ICPL proved to be very powerful and easy to use. The state transitions are invisible to the programmer, removing entirely the need for interrupt level programming. The inputs are not polled but are true interrupts. This means that executing tasks may be aborted or suspended and modified at any time. Control is not only available at specified points. The graphical output facilities of ICPL need improvement with the necessary addition of procedures and transformations. The easy access to other programming languages proved to be very useful, especially where FORTRAN was utilized to maintain the data base in the line editor test problem and to perform the necessary scientific computations in the harmonic synthesis test problem.

ICPL has poor mechanisms for program control, the only constructs available being GOTO and IF statements. This is clearly insufficient for larger application programs, but the language as it exists permits surprisingly readable programs to be written, probably due to the delimiting nature of the OBJECT / ACTION pair statements.

The only data manipulation facilities available are integer and character string variables. The character string variable proved to be very useful for the creation and manipulation of light-buttons. The following operations can be performed on a character string array:

APPEND TO - add characters to string
CLEAR - clear string
LIST - display string on screen

The lack of more powerful data manipulation facilities did not prove to be a problem in the test cases examined since the facilities of other languages were used when required. However, in larger applications a real number variable capability and more powerful computation facilities would prove useful, because it is annoying to have to call one-line external programs.

ICPL provides easy access to software available in external routines by a subroutine call capability, and interprogram variable passing. A library of runtime routines provides a set of function calls available to external languages. ICPL lacks a good method of transferring its special string character data type to external programs written in other languages.

2.3.4 GRAPPLE (Graphical Application Programming Language)

The basis of GRAPPLE [4,5] is the function call. Giving the name of a function is an instruction to GRAPPLE to perform that function. There are built-in functions called 'Primitives' to perform basic tasks, and the programmer may define his own functions in separate paragraphs within the program. These structures make the language highly procedure-oriented. Using the recursive procedures, it is easy to describe complex pictures. Graphical output is done via special output

primitives, for example:

S(X,Y)	- Position the beam to location X,Y
T ('string')	- Output the text string enclosed in quotation marks
D (DX,DY)	- Vector deltaDX by deltaDY

A good set of transformation capabilities is also provided as "Action Modifiers". The following GRAPPLE modifiers are available:

Scale	*
Rotation	?
Reflection	??
Translation	
Repetition	& and &+.
Conditional	& and &+, && and &&+

A typical statement or 'Action List' would be a combination of function calls and the above. This leads to very cryptic but powerful statements. The structure of a GRAPPLE program is that of a network of recursive procedure calls. Functions may invoke other functions to any level. Although it is easy for a programmer to mentally trace down this network of calls it is difficult for him to follow the stacked return addresses back up. This, and the fact that parameters are passed on stacks, make GRAPPLE programs difficult to read.

GRAPPLE is a highly procedure-oriented language. The procedures are referenced as functions. The following types of functions exist:

USER-DEFINED FUNCTIONS
BUILT-IN FUNCTIONS

The built-in functions are of three types:

PSEUDEO-VARIABLES
PRIMITIVES
SYSTEM FUNCTIONS (EUPHEMISMS & SYSTEM LIBRARY FUNCTIONS)

A GRAPPLE program consists of a number of procedures. For example, the harmonic synthesis test problem consists of the following seven procedures which were defined within the body of the program,

SHMDEMO
MENUSHM
REFRESH
LIST
PLOT
CHANGE

together with a definition of the constants and variables used. Each user-defined function consists of references to the various built-in functions and their action modifiers. A sample user-defined function, PLOT, is presented below to illustrate the use of the functional capability of GRAPPLE.

"" ROUTINE TO DRAW THE GRAPH OF THE SUPERPOSITION OF SINE
"" WAVES SPECIFIED IN THE ARRAYS 'AMPL' AND 'PHASE'

PLOT S(100,1800),-4->THETA,
&(91)(THETA+4->THETA,0->TOTAL
&(HWM)(\$COS(THETA*\$\$R+PHASE(RR\$))*AMPL(\$\$R)+TOTAL->TOTAL)
D(THETA*10+100,TOTAL+1800)),2;

Upon referencing the name 'PLOT', the entire 'user-defined function' from the ':' to the ':' will be executed. An example

of a 'pseudo-variable' is '\$\$R' and a 'primitive' is '\$S(100,1800)'. '\$COS' is an example of a 'euphemism' system function. The use of these function calls provides a powerful picture description and computational capability, especially since recursion is permitted.

The only graphical input facility provided to the GRAPPLE programmer is the X,Y screen co-ordinates of the strike position. In the two test problems this input identification was partially performed by pre-defined system library function, '\$RUNMENU'. This '\$RUNMENU' function presents light-buttons at pre-determined positions on the display screen. If this does not meet the programmer's requirements, then a new function must be written by the programmer, utilizing the '\$\$X', '\$\$Y' absolute screen co-ordinate pseudo-variables. This is a very restrictive form of graphical input since the x,y co-ordinates of the identifier strike must be compared against the position and shape of every object displayed on the screen. A prohibitive number of tests would be required if an object had a complicated shape.

In the line editor test problem, the real-number array capability of GRAPPLE was an adequate data structure. However, in more general and larger application, this certainly would not be the case.

GRAPPLE provides an adequate method of referencing externally compiled routines that have been written in FORTRAN

or PL/I. However, parameter passing has to be provided by FORTRAN COMMON or PL/I DECLARE statements. In the fixed COMMON block many variables are passed, most of which have meaning only to a systems programmer thoroughly familiar with the inner workings of GRAPPLE.

The control structure of GRAPPLE consists of the following functions:

- & (repetition)
- & (conditional if)
- && (conditional do while)

The latter two functions are basically extensions of the repetition, &, function. It is desirable that a language contain these structural constructs (with the noted beneficial absence of a GOTO statement) but one questions the cryptic syntactical implementation provided by GRAPPLE. A repetition of 0 or 1 times is an awkward way to implement an 'IF' function.

2.3.5 GRAPHICS APL

APL is a language designed for quickly writing scientific calculation programs, and its power lies in its capability of handling vectors and matrices of numbers as easily as scalars. It is a function oriented language, where many functions may be easily defined and invoked within a program. The major use of APL occurs in the timesharing environment, where small interactive programs are written to perform fast calculations. The extensions of APL to support graphics mostly consist of a group of pre-defined functions which allow simple drawing on the

display screen, primarily for the purpose of graphically plotting numeric results.

The work of Bork at the University of California [11,12], Harris at Dartmouth College [13] and Tektronix Inc. have led the development of APL graphics. Bork has implemented a set of function calls which allow 2D and 3D vectors, windowing and scaling and can be used as procedures in the same manner as other APL functions. These functions have been designed primarily for plotting mathematical functions, although they can be used to define pictures. Only very restrictive graphical input facilities are provided, and the output functions are designed for the use of direct view storage tubes and other non-dynamic devices. Although general transformation calculations can be handled via the matrix manipulation facilities in APL, the programmer must write this himself. Because APL is an interpretive language, it is usually difficult to access external programming languages and make use of external data structure capabilities, however, generalized facilities for accessing external software from APL have been written. [44].

Graphics APL does not contain sufficient facilities to allow it to be used as a general purpose graphics language, but it is an excellent tool for plotting mathematical functions, for which it was designed. Some implementations of APL graphics, notably Carleton University's 'APLGRAF', have been written to be relatively display device independent. [45], allowing

considerable freedom in the use of the language on different terminal hardware.

2.3.6 BASIC

BASIC is a popular language used primarily in an online timesharing environment which has been augmented by some manufacturers and universities to provide a graphics capability. Dartmouth College, the originators of the BASIC language, have indicated that a graphical output facility is a general requirement within a community of users. At Dartmouth 10% of all program executions result in graphical output [13], and a majority of these are written in BASIC by making calls to an external subroutine package. An example of this type of subroutine package system is BASIC/GT provided on the DEC PDP 11/40 GT/44 computer.

The BASIC [14] language provided on the Digital Equipment Corporation GT-44 computer system provides a graphics capability via a collection of primitive graphic functions, accessible by a call statement. An example of these calls are:

Call 'VECT' (X0,Y0)

- Draw a vector of X0,Y0 length.

Most calls, including the above vector call, have variable length parameter strings to optionally indicate intensification

of a line, etc. which makes their use very error prone. These call functions are machine-dependent and device-dependent in that they force the user to manipulate display files within arrays, and therefore utilize only a GT40/44 display. Only a low-level polled light-pen capability has been provided as a graphical input facility.

Luehrmann and Garland from Dartmouth College have proposed a syntax to include graphical statements as regular language statements within BASIC, without destroying the style of the language [13]. They have proposed the introduction of a PLOT X,Y statement to mark a point at location X,Y. A PLOT LIMITS statement defines the co-ordinate system and a PLOTTER statement associates a particular hardware output device. As an example of the following program will plot a sine curve on the screen:

```
190 PLOT LIMITS 0, 6.28, -1, 1
200 FOR X=0 TO 6.28 STEP .1
210 PLOT X, SIN(X);
220 NEXT X
225 PLOT
230 END
```

The bare PLOT statement in line 225 means "turn off the beam (lift the pen)". The semicolon in line 210 indicates that the points plotted should be connected by line segments. This is a meaning similar to the concatenation use of the semicolon in a PRINT statement, but it seems to be too small a differentiation for such an important effect.

This syntax provides a useful graphical output facility to

an online timesharing language, but it doesn't make BASIC a general purpose graphic language. The BASIC language provides adequate computational facilities, but poor data structure capability. Since the subroutine facility in BASIC is 'GOSUB line-number' (a local subroutine capability) and because BASIC is an interpretive language it is difficult to access external software written in other languages.

2.3.7 EULER/G

EULER [15,16] was designed by Wirth as a generalization of Algol 60. The language is extremely well structured and rich in data manipulation facilities, providing numbers, values, text strings, lists and procedures. The 'lists' facility provides a simple dynamic data structure capability, but does not provide hierarchical associative ring structure facilities often employed in graphical application programs. The graphical output facilities of EULER/G [17] are provided by display procedures. An example of an EULER/G display procedure would be:

```
BOX ← 'LINE(1,0);LINE(0,-1);
      LINE(-1,0);LINE(0,1)'
```

Some examples of the calling sequences are:

```
BOX AT [50,50] SCALE 1.5 ROT .5
MOVE TO [50,50]; BOX
```

In the first example the procedure BOX was called, but its

effect was modified by the translation, scale and rotation modifiers. In the second example, the drawing position was set before calling the procedure. This means that the definition of the procedure remains invariant, but its effect can be modified by the manner of the call. This is a very desirable way of defining a graphical procedure.

Input response in EULER/G is handled by a reserved procedure. The procedure HIT (X,Y) will return a value of 'true' or 'false' according to whether any lines or text lie within a small distance (X,Y) on the screen. To determine which item was struck, each procedure may be named. The name of the item pointed at is returned in the variable HITNAME. This method of input is not interrupt driven. The programmer must decide when he wants input and then he must poll the input device.

EULER/G is the most sophisticated graphics programming system surveyed, but it has several serious drawbacks. Firstly, EULER must run in an interpretive manner in order to support the highly flexible structure allowed for variables. This results in it being slow and inefficient. The syntax of EULER contains several direct references to the characteristics of the display and input devices used and to the hardware of the PDP-10 computer on which it is implemented. This makes it non-portable.

2.3.8 DIAL

The DIAL language [8], developed at the University of Utah, is a special purpose language for the control of graphical interaction dialogues, which incorporates both a facility for describing interaction dialogues via a state diagram description, and a facility for defining and branching to procedures. The base language is a subset of ALGOL 60. The four statements DURING, ENTERING, ON and ENTER allow a framework for the interaction dialogue to be defined.

An interaction dialogue may be described by a state diagram technique where at every node in the diagram the computer waits for input in the form of data or a decision upon which control path to follow. The commands required for a user to draw a line consist of requesting that a line be drawn, input of the starting co-ordinates, and then input of the end co-ordinates. This interaction dialogue would have the state diagram description as shown below in Figure 1.

Figure 1 : DIAL State Diagram

A state in a DIAL program is described by the DURING statement and a branch by an ON statement. The line drawing example above has been described by Newman as illustrated below [8].

```

DURING 1 DO
  ON LBUT "LINE" DO ENTER 2;
DURING 2 DO
  ON PENDOWN DO BEGIN
    X PENX; Y PENY;
  ENTER 3
  END;
DURING 3 DO
  ON PENDOWN DO BEGIN
    DRAWLINE(X,Y,PENX,PENY)
  ENTER 1
  END;

```

The prime difficulty with the state diagram technique is the large amount of code required to describe even simple interaction dialogues. Newman makes this point in his book [8,page-350]. "For [some] styles of [interaction dialogues] the technique creates problems. For example if we were to extend ... a four - command [dialogue] to 40 commands, we should require 41 states. If each of the 40 commands required two data elements

instead of one, the number of states would rise to 81. Thus the state diagram technique may produce very complex programs defining quite simple [interaction dialogues]. It may also be possible to produce unsatisfactory [interaction dialogues] in this way, because of the freedom with which program states may be added. Since the program responds differently in each state, the user is likely to become confused unless he is very clearly guided by means of directions from the program." The problem is basically that large state diagrams, although technically an exact representation of an interaction dialogue, are difficult to read and therefore their specification is error-prone..

Of the languages surveyed, only the ICPL language and DIAL provided convenient facilities for defining the interface between the man and the machine. In other languages, what control was possible was exercised by ordering the input statements.

3. REQUIREMENTS FOR A GRAPHICS PROGRAMMING SYSTEM

From an examination of the programming languages surveyed in the previous chapter, it was concluded that the writing of interactive graphical application programs is greatly aided when the following five facilities are available.

- a) Graphical input response facility
- b) Graphical drawing facility
- c) Structured programming constructs
- d) Data manipulation facilities
- e) Easy access to external software

Each of these facilities is discussed below, with particular emphasis on how they may be provided in a language.

a) Graphical Input Response Facilities: One of the main requirements of any interactive process is to define the response of the system to each input. One approach which has been suggested by Newman [10] is that a system be considered as a finite state automaton where an 'action' is simply an input to the system and the corresponding 'reaction' is determined by the state of the system at the time. This approach to defining a program control sequence can be implemented such that the programmer is required to be specifically aware of all the states of his system in order to specify the action-reaction sequences, or the system can be designed such that these state transitions are transparent to the programmer. DIAL [8] is an example of the former system, whereas ICPL [1,2,3] is an example of the latter. The number of explicit state transitions can become quite large using the former system. Because it puts less of a burden on the programmer to organize the states of his program, it is felt that the latter method is superior.

Most languages provide only highly device dependent information about particular inputs. This device dependence can be removed by providing suitable language constructs that reference a small number of idealized devices.

b) Graphical Drawing Facilities: There are two common ways of providing graphical drawing facilities. The first involves the notion of a function or subroutine call and does not require any modification to the base language from which it is called. Many graphics systems available today (especially those supplied by manufacturers) are of this type. Specific examples are GINO [9] and the PDP 15 VT15 graphics package [6]. Graphical drawing facilities may be provided by specific instructions within the language. This provides the capability of defining display procedures [18].

This method, however, requires modification of the syntax of the base language or the creation of a complete new language. Display procedures provide distinct items on which to apply transformations, and therefore provide a compact, high-level method of defining pictures. EULER/G [17] and GRAPPLE [4,5] are examples of language systems providing this display procedure capability.

c) Structured Programming Constructs: It is very important that the base language provide good facilities for defining the overall structure of the program. Examples of languages exhibiting excellent program structure facilities are ALGOL [19] and to a lesser extent, IFTRAN [20,21,22]. Both these languages provide good constructs for iterative and conditional operations such as IF, WHILE and ELSE. A local subroutine capability is desirable, and provides a convenient method of implementing a display procedure capability.

d) Data Manipulation Facilities: A computer graphics program describes pictures in numerical form and textual information in terms of character strings and it is therefore useful to have both numerical and character data types. In certain applications it is essential that a representation of a picture be stored in a dynamic data structure, thus providing the capability of manipulating and referencing it. It may overburden the syntax of a graphics language to provide a generalized data structure capability, therefore, as a minimum, access to external facilities written in other languages should be provided.

e) Easy Access to External Software: Any good programming system should provide an easy and convenient access to external software packages. Very often an application program will require access to some external routines that are already provided in the computer system library or that already exist written in some other language like FORTRAN or ALGOL. The communication of data to external software should be in a simple manner, such as direct parameter references.

These requirements are not adequately met in any of the surveyed languages. The 'IMAGE' language was designed to specifically satisfy the above criteria. A unique interaction control structure, along with a display procedure based picture description grammar and device independent interaction input facilities were combined to form 'IMAGE'. The next chapter will present the syntax of 'IMAGE'.

4. THE DESIGN OF A NEW GRAPHICS LANGUAGE, 'IMAGE'

This chapter describes the interactive graphic programming language 'IMAGE'. 'IMAGE' has been designed to provide a language in which a relatively untrained person can write interactive graphic application programs. The language is designed for 'application programmers', that is, persons who have some knowledge of programming but have most of their expertise in the field of their application. One of the primary goals in the design of this language is that it be simple to use and easy to learn. A system of defaults has been established so that a feature which a programmer does not know about does not affect him. Trivial programs are short to write. For instance, in the minimum case it takes only three commands to draw a single line or to put a string of text on the screen, two of which are the ENTRY and END statements. The language provides a basic set of drawing commands and augments them with a powerful group of modifiers. The language is procedure-oriented and procedures may be easily defined within a program.

An easy method has been provided to communicate with external programs written in other languages, so that an applications programmer can write sections of his program package in languages more suited to other tasks such as data manipulation or complex calculations. This feature has the virtue of easily allowing the programmer to add a graphics capability to an already existing applications package written

in some other language, and it allows the major emphasis of the syntax of this language to be for the writing of graphics programs.

The structure of the execution flow control statements borrows from the format of the 'IFTRAN' FORTRAN preprocessor (Bezanson [22], Miller [20,21]) to help produce error resilient programs. The use of statement labels in controlling the execution flow is eliminated by structures such as IF ... ELSE ... FIN and WHILE ... FIN. The other common use of statement labels in programming languages is to associate a format for an I/O or character string instruction. The adoption of a simple data formatting scheme and the use of character string variables has allowed for the design of a language structure which uses no labels at all.

This language has been designed so that it directly addresses the graphical problem in a hardware independent manner. A program written in this language should operate regardless of which I/O devices are available. Device independence has been designed into the language so that particular device technologies have a minimum effect on the programs written in the language. For example, a program which uses a light-pen [8] as an identifier on one system should be able to operate just as well using a track-ball [8] and a cursor [8] on another. The language treats all displays as identical devices and allows similar operations to occur on them. The programmer introduces device dependencies only when he depends

on the speed of a particular implementation. The myriad of input devices have been broken into six classes. Three of these classes are particularly concerned with interaction with the display. The other classes of interaction are general in nature and could be associated with a non-graphical program. Interaction control commands allow graphical devices to be assigned to particular input functions. Each device behaves as a virtual device so that it is possible to use software techniques to allow one device to perform the function of another. For example, a positioning device such as a track-ball can be used to identify light-buttons, even though the concept of the light-button was derived from light-pen usage. The use of virtual I/O devices leads to the device- and machine-independence of this graphics language.

4.1 GENERAL PROGRAM FORMAT and SYNTAX DEFINITION

The structure of an 'IMAGE' program consists of two types of instructions, executable statements and definitional statements. To distinguish between these two types two different margins are used. Executable statements must be indented one or more spaces and definitional statements may not be indented. Executable statements are executed serially in blocks. A comment line is indicated by a '*' in column one and a line continuation is indicated by a '&' in column one. Comments may also be placed on the right-hand side of a line of code by delimiting it with the double asterisk symbol '**'.

There are four types of blocks: OBJECT, ACTION, ENTRY and PROCEDURE definition blocks. OBJECT blocks define graphical material to be displayed and ACTION blocks delimit code to be executed upon an identifier strike on that displayed material. ACTION blocks must immediately follow their associated OBJECT blocks, thereby indicating that association. One ENTRY block may appear in a program to allow initialization and is usually the first block of executable code. If no ENTRY block is defined, a default entry sequence consisting of a DISPLAY followed by a SEEK and then a WAIT command is assumed. Procedure blocks provide a local subroutine capability, necessary to prevent the duplication of code. Transformation operators may be applied to a PROCEDURE block when a call is made to modify the displayed appearance of any graphical instructions described within.

Graphics routines may be either main programs or subprograms. A subprogram must have its name defined in its first statement along with any calling parameters. It is not necessary to start an applications package with a graphics program. All necessary display initialization is taken care of automatically, so a set of graphics subprograms may be added to an already existing applications program to provide a graphics capability.

Statements within an OBJECT block are executed by invoking the language statement DISPLAY. Any display code generated is marked (TAGged) so that an ACTION routine may be executed upon the stimulus of the displayed OBJECT. OBJECT blocks may

therefore define what is commonly referred to as a 'light-button'. An ACTION block defines the action to occur upon the stimulus of the preceeding OBJECT block. An ACTION block is basically a high-level interrupt handler routine. The identifier interrupt structure is disabled upon the execution of an ACTION block and is re-enabled by the use of a SEEK command. Execution is suspended at the end of an ACTION block by the use of a WAIT command. The program then reads as a series of OBJECT / ACTION pairs whose execution is interactively controlled. Figure 2 illustrates this program structure.

NOTATION

In describing the syntax of a particular language statement a template is used to show its form. In the template, upper case letters describe the words and phrases which make up the command. Square brackets enclose optional phrases. Variables or parameters are described in lower case. A choice between two or more options is presented as multiple phrases placed vertically above, each other. In the following diagram the blocks of executable code are enclosed in boxes.

Figure 2 IMAGE Program Structure

4.2 DEFINITIONAL STATEMENTS

The definitional statements determine the structure of a program by defining the block structure, the graphics environment and all variable types to be used. The following subsections will describe the syntax and semantics of each definitional statement.

4.2.1 BLOCK DEFINITION STATEMENTS

Each of the following statements defines a block of executable code.

SUBPROGRAM name [(parameters)]

a) - defines a program to be a subprogram of name 'name' with an optional parameter string (parameters). If used, this statement must be the first statement in the program. If this statement is not used, then the program is a main program.

ENTRY

b) - defines the beginning of an ENTRY block. There must be no more than one ENTRY block per program, to be used for initialization. This block usually terminates in some transfer of control statements such as the display and activation of a light-button menu. If no display control transfer statements occur in the ENTRY block, or if the ENTRY block is omitted, a default sequence will be executed. All OBJECTS will be DISPLAYed and the identifier interrupt mechanism enabled (SEEK). If a WAIT statement is not explicitly written at the end of an ENTRY block, then it is assumed.

OBJECT [name]

c) - defines the beginning of an OBJECT block. Statements within an OBJECT block are executed when the OBJECT is DISPLAYed. Any display code generated is marked (TAGged) so that an associated ACTION routine may be executed upon the stimulus of the displayed OBJECT (light-button). OBJECT blocks may be named so that they may be selectively displayed or re-displayed. They may be made conditional by the use of IF statements within the body of their code.

ACTION

d) - defines the beginning of an ACTION block. An ACTION block describes the action to occur upon the stimulus of the preceeding OBJECT block. There may be only one ACTION block associated with an OBJECT block and ACTION blocks must follow OBJECT blocks. A stand-alone ACTION block is illegal. All ACTION blocks must terminate in some type of transfer of control of the form SEEK / WAIT, RESUME or SEEK / / WAIT. If this control sequence is omitted, then a SEEK / WAIT sequence is assumed.

PROCEDURE NAME ((parameters))

e) - defines the beginning of a PROCEDURE block. Procedure blocks are local subroutines with global access to all program variables. Procedures may be called recursively in order to easily program highly symmetric shapes. Because recursion is limited to procedure calls, which must be local within a program, recursion may be used freely. External programs which do not support recursion will not be called recursively even if they are called from within a recursive procedure, because all references to external routines are self contained.

END

f) - defines the physical end of the program, and must be included.

4.2.2 DECLARATION STATEMENTS

Declaration statements define either the type or access to variables and arrays, or they define some aspect of the graphics environment such as the display screen co-ordinates to be used. Variable declarations will be discussed first.

All variables used in an 'IMAGE' program must be declared either at the beginning of the program or at the beginning of a PROCEDURE block. Not only does this total declaration of all variable types simplify the task of parsing statements of

programs being compiled, but it also forces the programmer to be aware of the type of each variable, and it allows for better compiler diagnostics. All variables except the dummy variables used as formal parameters to a procedure are global to an entire program, and therefore the location of a declaration statement either at the beginning of the program or the beginning of a procedure block does not matter. The definition of variables within the header of a procedure block is provided purely for the readability of the procedure block code.

STRING (nc) variable name, ... , array name (nl), ...

- defines string variable buffers 'nc' characters long of name 'variable name' and arrays of name 'array name' containing 'nl' elements.

INTEGER variable name, ... , array name (nl), ...

- defines integer variables and arrays of integer variables. Only one-dimensional arrays of length 'nl' are allowed.

REAL variable name, ... , array name (nl), ...

- defines real variables and arrays in the same manner as in the INTEGER statement.

PARAMETER variable name, ...

- other types of data may be passed between external programs through the graphics program, but no operation may be performed on these data elements. This type of statement only defines the names of these other data elements for use in parameter strings.

EXTERNAL VARIABLE[S] [/label name/] variable name, ...

- previously defined variables declared external are treated as global variables to all programs. The order of the variable names in the list defines the correspondence between variables named here and the variables declared in other programs. This is similar to the manner used by the FORTRAN language for COMMON, and variables may be passed between EXTERNAL VARIABLE statements and FORTRAN COMMON. The equivalent of both blank and labelled common is provided in the form of an optional label specified for the variable name list. STRING variables appear as arrays of maximally packed characters in FORTRAN. (A4, A5 or other depending on the machine)

EXTERNAL PROGRAM[S] program name, ...

- all external programs which are called must be explicitly defined.

The remaining declaration statements define the factors affecting the graphical environment, and may only be defined at the beginning of a program. The PAGE command allows the programmer to define both the shape and the co-ordinate system of the screen for which he is programming. Co-ordinates are specified inclusive of the first addressable point and exclusive of the last point so that 0 to 100 indicates a range of 100 points as well as co-ordinates 0 to 99. If the hardware-defined co-ordinate system of the display device used differs from the one defined in the PAGE command, then suitable scaling is done by the runtime software. Shape differences are harder to deal with. The 'aspect ratio' of a picture is the ratio of height to the width of a picture. Television screens have an aspect ratio of 3 by 4, while a square screen would have a ratio of 1 by 1. There is no standard shape for a graphics screen. Some are higher than wide, some are wider than high, such as TV-based raster displays, but most are square. The user of a non-square

screen would be frustrated if he was limited to a square sub-area of his screen, but programs written for anything but a standard screen shape are incompatible. The 'IMAGE' language solves this problem by freely allowing the programmer to define both the shape and the co-ordinate system he will use. If a program is moved to another machine with a differently shaped screen, then the runtime software will scale the picture and circumscribe it within the other screen shape. The other environment-defining statements cause specific character or symbol sets to be associated, and define the length of the 'inking' queue.

PAGE xmin:xmax

PAGE xmax, ymax

PAGE xmin:xmax, ymin:ymax

- defines the dimensions of the display screen, in the co-ordinate system to be used. If no min term is specified, 0 is assumed. The default screen size is 0:1000, 0:1000. The natural meaning of a range specification of 0 to 1000 is to include 1000 drawing intervals, but on a graphics display screen a point is really a dot of the finite size of one drawing interval centered at the co-ordinate position. To avoid the possible confusion of having one more co-ordinate position than drawing interval the following interpretation is used. The min values are inclusive and the max values exclusive. A display screen defined as PAGE 0:1000 would have the valid addressable co-ordinate points of 0 to 9999.

CHARACTER SET[S] nameA, nameB, ...

- makes available additional character sets for the display of text. If this statement is not written, then a default set is used. The default is standard upper case English alphanumerics or STANDARD. The default set is always available. Specific character sets can be invoked by display modifier commands.

SYMBOL SET[S] name of set, ...

- makes available the sets of symbols specified. Symbols may be drawn by the 'SYMBOLS' command which references the symbol-by-name. The symbol sets are searched for the specific symbol in the order they are named. This means that only the first definition of a multiple defined symbol is accessible. For example, a 'resistor' symbol would be found in the symbol set 'electrical'. Symbol sets are produced by a special 'Symbol Set Builder' and are stored in the library along with library functions, or may be specifically loaded with the program.

SKETCH Q[UEUE] (var)

- defines the size of the queue of points used in 'inking' the lines sketched on the display. A more complete description of this command is given the section on interaction control (sect 4.3.5).

4.3 EXECUTABLE STATEMENTS

Executable statements are normally executed serially within the blocks defined by the definitional statements. They are used to manipulate character and arithmetic expressions, perform I/O with the display, invoke procedure and subprogram calls, set display modification variables and control the flow of execution. Commands to execute simple control within a block are the IF ... ORIF ... ELSE ... FIN, CASE, REPEAT ... FIN and WHILE ... FIN constructs. These commands delimit the executable code within an OBJECT, ACTION, or ENTRY block into sub-blocks. The detailed workings of these commands are presented in Appendix C. The 'FIN' statement ends all sub-blocks of code. The use of this single block terminator prevents the possible error of interlacing sub-blocks. Sub-sub-blocks must be totally contained within a single sub-block. The 'IF' and 'WHILE' constructs

completely eliminate the need for a 'GO TO' statement but are awkward to program with in some situations [23]. Some additional looping and escape mechanisms are provided to prevent this.

The 'ESCAPE' statement allows for the conditional termination of the execution of a block and has the form 'ESCAPE' or 'ESCAPE (condition)'. Upon the execution of this instruction, control is transferred to the statement following the FIN of the current block.

A 'REPEAT' loop provides a loop with built-in counters similar to the DO statement in FORTRAN, but without a label reference. It counts up or down using INTEGER or REAL numbers and can be used without a counter to provide an infinite loop.

The 'CASE' statement provides the same facility as the FORTRAN 'computed GO TO' statement in providing a multi-way conditional branch, but in a block structured manner. It has the advantage over an IF ... ORIF ... FIN construct of requiring only one test.

String and arithmetic assignment, conditionals and subroutine calling commands are provided in a conventional manner and will only be discussed briefly. Arithmetic expressions are evaluated in a similar manner to those in FORTRAN and BASIC and will not be presented in detail here. The major difference is that a multiple arithmetic assignment notation and a special structure for string assignment formats

is used. Appendix B contains a more comprehensive discussion. The graphical drawing, display editing and interaction control statements will be discussed in greater detail because, together, they provide a novel approach to manipulating graphical images.

4.3.1 DISPLAY EDITING COMMANDS

The display editing instructions are provided to permit the creation and modification of displayed pictures. These commands are only legal in ENTRY or ACTION blocks or in PROCEDURES called by them, because these blocks control the manner in which material defined in OBJECT blocks is to be presented. The following commands provide a control structure which defines a framework for the man-computer interaction dialogue. ACTION blocks are executed upon a user generated identifier 'strike' upon the associated OBJECT block, if the SEEKING mechanism has been enabled. The display command 'SEEK' enables the interrupt system on some or all OBJECTs. The five commands DISPLAY, ERASE, SEEK, WAIT and RESUME form the core of the control structure. Variations on these five commands are available to append material to the screen and to erase dynamic material from the screen. An 'EXCLUDING' phrase may be added to those commands which reference named OBJECTs to indicate a reference to all OBJECTs except those named. The syntax of these commands is presented below.

DISPLAY [name, ...]

- erases the screen. Executes all OBJECT blocks or all named OBJECT blocks if specific names are given. Multiple names are separated by commas. The unconditional DISPLAY is the normal method of generating a picture and associated light-button menu on the screen.

DISPLAY EXCL[UDING] name [, name, ...]

- performs the DISPLAY function on all OBJECTS except the named OBJECTS.

REDISPLAY name [, name, ...]

- erases the named OBJECT or OBJECTs from the display and re-executes those OBJECT blocks.

APPEND [name, ...]

- does not erase the screen. Executes all of the named OBJECT blocks. The image generated is added to the display and is available to the identifier interrupt mechanism for SEEKing purposes. If the same object block is appended to the display many times then a 'strike' on any instance of it will cause the associated ACTION to be executed. No special control is provided over particular instances of a multiple displayed object. If there is any reason to differentiate, then there is sufficient reason to define more than one object. PROCEDURE blocks may be used to prevent duplication of object definition code in such cases.

APPEND EXCLUDING name [, name, ...]

- performs the APPEND function on all OBJECTS except the named OBJECTS.

ERASE [name, ... , name(sd), ...]

- erases the named OBJECT, or OBJECTs from the screen. If no name is specified erase all of the OBJECTs and all material generated in ACTION routines in this program, or in programs called by this program. If this statement is executed in a main program, then the entire screen is cleared. OBJECTs may be sub-delimited by TAG codes. To erase a section of an OBJECT specify the OBJECT name followed by a single or a range of sub-delimiting codes in brackets.

ERASE EXCL[UDING] name [, name, ...]

- performs the ERASE function on all OBJECTs except the named OBJECTs.

REMOVE

- removes from the display all images generated in ACTION routines. This allows the erasure of any image generated in other than an OBJECT block and which is therefore not named.

SEEK [name, ...] [ON]

- enables the identifier interrupt mechanism for all OBJECTs or for the named OBJECTs if names are specified.

SEEK EXCLUDING name, ...

- performs the SEEK function on all OBJECTs except the named OBJECTs.

SEEK [name, ...] OFF

- disables the identifier interrupt mechanism for all OBJECTs or for only the named OBJECTs if names are specified.

WAIT

- suspends execution of the program and waits for an identifier interrupt. An error condition results if a WAIT command is executed and the SEEKing mechanism is disabled.

RESUME

- continues execution of the section of the program which was interrupted when the current ACTION routine was entered. The status of the previously executing program is restored. Since most languages do not support re-entrancy of programs, care must be taken when using the RESUME command. There is no potential conflict if the ACTION routine which issues a RESUME does not invoke any subroutines in common with those in the main body of the program. IMAGE procedures may be re-entrantly invoked, because they support recursion by stacking return address (but not variables), and the IMAGE run-time software guards against coincident multiple use by delaying interaction interrupts until the particular support function is complete.

The normal manner of ending an ACTION or an ENTRY block is by executing the command sequence DISPLAY then SEEK then WAIT. This clears the display and executes all of the OBJECT blocks, thus creating a new picture and menu of light-buttons. This may

be the same picture and menu as before or a different one, depending on the condition of flags and program variables. It is only necessary to use the display editing capabilities above if the picture on the screen requires a large number of calculations to produce. It is usually easier to clear the display and regenerate the entire picture. The SEEK command enables the identifier interrupt mechanism and the WAIT suspends program execution waiting for an interrupt. A typical sequence would have the form:

```
OBJECT
:
:
ACTION
:
:
DISPLAY
SEEK
WAIT
```

When an ACTION routine is executed the interrupt system is turned off. If it is not turned on completely or partially (by a SEEK command) at or before the end of the ACTION routine, and if a WAIT command is not given when the execution of the ACTION routine is complete, then SEEK and WAIT are assumed as follows:

```
OBJECT
:
:
ACTION
:
:
OBJECT
```

a SEEK
WAIT

is assumed here because the ACTION routine was ended by the definition of another OBJECT.

The DISPLAY command may be written anywhere within an ACTION or ENTRY block, and multiple occurrences of it merely multiply execute the OBJECT blocks. A SEEK command may also be written anywhere within an ACTION or ENTRY sequence. The SEEK commands enable specific or all interrupts. These interrupts remain active until a SEEK OFF command is executed or another ACTION routine is entered. If the SEEK command is written before the end of an ACTION block then the interrupts are enabled and the remainder of an action block may be aborted by an identifier interrupt on another object. An identifier 'strike' will cause the abort of any currently executing ACTION or ENTRY block if another valid ACTION routine is invoked as a result of that 'strike'. This is illustrated below.

OBJECT
.
.
ACTION
.
.
SEEK
.
.
OBJECT DOG
.
.
ACTION
.
.
OBJECT CAT
.
.
.

Since SEEK has been issued before the end of the ACTION block, the code following the SEEK may be aborted by an identifier strike on any OBJECT. If the strike was on OBJECT 'DOG' the execution of this code would be aborted and the ACTION associated with OBJECT 'DOG' would be executed. If the OBJECT 'CAT' was identified, no action would occur because there is no ACTION associated with it, and the execution of the currently executing code would not be interrupted.

Two other commands which control the flow of execution are STOP and RETURN.

STOP

- stops execution of the program; closes all files on I/O devices; erases the display.

RETURN

- returns execution to the calling program. This command is only valid in a subroutine or a procedure.

The TAG command is used to delimit displayed material. When issued within an OBJECT block the TAGs sub-delimit the OBJECT. For example, if an OBJECT was an entire electrical circuit diagram drawn from a data base, TAGs could be used to sub-delimit the OBJECT into individual electrical components. The TAG is issued as:

TAG

When part of an OBJECT is identified the particular sub-delimiting TAG number is made available to the program in the reserved variable ITEM. It is only meaningful to issue TAGs to sub-delimit OBJECTs. TAGs may also be issued in PROCEDURES or external routines called in OBJECT blocks. TAGs are ignored elsewhere. The variable ITEM will equal 1 in the ACTION routines associated with non sub-delimited OBJECTs.

4.3.2 GRAPHICAL DRAWING INSTRUCTIONS

Four types of primitives may be placed on the graphics screen: POINTS, LINES, TEXT and SYMBOLS. For each, the drawing location may be explicitly specified or left at the end of the

last item drawn. In order to define the drawing location a command is provided to set the location of the drawing beam.

SET AT(x,y)

- sets the drawing beam to the x,y location in the current screen co-ordinates.

To draw a point at a given location.

POINT [AT(x,y)]

- draws a point at the specified position or at the current position if the 'AT' phrase is omitted.

The following commands allow vectors or lines to be drawn. Three modes of drawing are allowed. A line or a concatenated series of lines may be specified by relative displacements in a LINE command, or a series of lines may be drawn through a group of points by a LINES THROUGH command. A LINES TO command allows a line to be drawn from one specified location to another specified location.

LINE dx,dy [AT(x,y)]

LINES dx1,dy1 / dx2,dy2 / ... [AT(x,y)]

- draws a line delta-X by delta-Y units as a relative displacement. The multiple lines command 'LINES' allows the drawing of concatenated lines. The phrase 'AT(x,y)' defines the starting position of the line. If it isn't specified then the starting position is at the end of the last item drawn.

LINES THROUGH x1,y1 / x2,y2 / ... [AT(x,y)]

- starting at x1,y1 draws a sequence of concatenated lines through the points x2,y2 The points x1,x2, etc, form a grid system about the current beam position or the point defined by the AT phrase in a similar manner to placing a piece of graph paper on the screen at a given position and drawing the lines through points drawn on that grid. For example, the command given below would draw a unit square, the four co-ordinates of which are given. The lower left hand corner of the square would be pre-positioned on the screen at location 100,100.

LINES THRU 0,0/0,1/1,1/1,0/0,0 AT (100,100)

The last of the three line drawing commands is somewhat different. The LINE TO command allows one line to be drawn to a specified screen location. Since both this location and the starting location are defined in the current page co-ordinates, the position the line will appear at in the current drawing page is fixed. The transformation modifiers (sect. 4.3.3) of ROT, REF, SCALE and the translation AT may not be appended to this command.

LINE TO x1,y1 [FROM x,y]

- draws a line to a specified page location, from the current beam position or the optionally specified 'FROM' position.

Textual material may be drawn by using the command:

TEXT string [AS (format list)] [AT(x,y)]

- the string and format list have the same usage as for string variables. The string may be a string variable, a string literal enclosed in quotation marks or an arithmetic variable. The format list is used to define the manner in which the character string will appear on the display screen. If it is not specified, a default free format is used. In the case of string variables the default format is to output the entire string. In the case of arithmetic variables the number is outputted in I (INTEGER) or R (REAL) format, whichever is appropriate. The special characters line feed or carriage return have meaning when displayed. A line feed changes the Y drawing position down (or minus) one character height. A carriage return returns to the location of the beginning of the text string and down one character height.

As an example the following text string will put the literal text string 'ABC' on the screen at location (500,500):

TEXT 'ABC' AT(500,500)

It is sometimes desirable to be able to output arrays of string variables in table format. The LIST command allows for the outputting of vertical column tables.

The LIST command has the format:

LIST (str var [, str var, ...] [STEP (displacement)] [AT(x,y)]

- display in table format the named string variables or arrays of string variables. These string variables may be either single variables, array elements, ranges of array elements or whole arrays.

The vertical separation between lines is set in the STEP phrase. A displacement of -50 would cause each successive line to be displayed down or minus 50 scope units. If the STEP phrase is not present a default separation of down one character height is used. The separation may be specified in character heights if the displacement term has the form of 'nCHAR[IS]'. The AT phrase defines the starting location of the table.

LIST TAB(1:3) STEP(-50) AT(500,500)

The command above will display the contents of the string variable TAB in a vertical column. It has the same effect as the following TEXT statements.

TEXT TAB(1) AT(500,500)

TEXT TAB(2) AT(500,450)

TEXT TAB(3) AT(500,400)

A SYMBOL may be put on the screen by name in the form:

SYMBOL name [, AT(x,y)]

- the symbol sets associated with the program in the type statements are searched for the specifically named symbol. Then the symbol is drawn.

The drawing of smooth arcs or other special shapes is handled by calls to library routines.

4.3.3 GRAPHICAL DRAWING MODIFIERS

Only a very basic drawing capability is provided within the 'IMAGE' language. The power of this facility is augmented by allowing the effect of each drawing command to be altered by specifying one or more modifiers along with the command. The drawing commands are used to specify the particular graphical shape and the graphical drawing modifiers specify how it will appear on the screen. Individual drawing commands, display procedures (sect 4.3.4) and the graphical effect of subroutine calls may be so modified.

Modifiers may be specified in two ways. A modification or a list of modifications may be appended to the end of a drawing command separated by the character ':' in order to affect only the drawing command to which they are attached. It is also possible to use some of these modifier commands as global modifiers by using them singly on a line. When modifiers are used in this way they affect all the code following within a specified range. Global modifiers within an ENTRY block define the status of the display status flags for the whole program. Modifiers within an OBJECT block apply only to the material

within that block. Because these blocks may be executed in any order, all display status flags are reset to the value they had at the end of the ENTRY block, or after the last ACTION block. The line texture and character set commands may be used as global modifiers.

The display modifier commands are described below. Default values (where applicable) are marked with an asterisk.

for transformations

AT(x,y)

- translates the referenced picture element to screen location x,y.

ROT[ATE] (rotation angle in degrees)

- rotates the referenced picture element. A positive rotation is defined in a counter-clockwise direction.

SCALE (real or integer number scaling factor)

- adjusts the size of the referenced picture element by the scale factor. Negative scale factors are meaningless. Rounding error can be encountered in large-size adjustments due to the finite word length of the computer.

REF[LECT] X

or

REF[LECT] Y

or

REF[LECT] (angle of line about which to reflect)

- reflects the referenced picture element about the x, y axis or about a line through the origin tilted by the specified angle.

for line texture

DOT
or
DASH
or
* SOLID

- modifies the appearance of the referenced picture element so that it is composed of solid, dotted or dashed lines.

INT[ENSITY] (factor from 0 to 1)
or
INT[ENSITY] DIM
or
* INT[ENSITY] NORMAL
or
INT[ENSITY] BRIGHT

- modifies the brightness of the referenced picture element.

FLASH [ON]
or
* FLASH OFF

- causes or inhibits the referenced picture element from repeatedly flashing.

COLO[UR] (factor from 0 to 1)
or
[COLO[UR]] RED
or
[COLO[UR]] GREEN
or
[COLO[UR]] BLUE

- adjusts the colour of the referenced picture element.

If a particular display device is incapable of a given type of line texture, then that modifier is ignored.

for character set

CHAR[ACTERS] name of character set

- causes a specified character set to be used for the referenced TEXT or LIST command.

for viewing

WINDOW (xmin:xmax,ymin:ymax)

or

WINDOW (xymin:xymax)

- clips the referenced picture elements at the specified co-ordinates.

WITHIN (xmin:xmax,ymin:ymax)

or

WITHIN (xymin:xymax)

- scales the previously windowed picture element to fit into the specified area. If no window statement had been stated, a window of the current drawing PAGE size is assumed.

Display modifier commands are interpreted in a left to right manner after the command on which they have an effect. For most of these operators order doesn't matter, but for the transformation and windowing operators it is very important. The AT phrase associated with most commands is a transformation operator. For this reason display modifier commands may be placed left or right of it on a line, separated by semi-colons

For example:

✓ . LINES 2,8/4,2/-6,-10;REF X;ROT(48);AT(20,0)

A triangle is defined at the origin, reflected about the x axis, rotated about the origin by 48 degrees and then positioned at position 20,0. See Figure 3.

LINES -5,5/-5,-1/10,-4 ;AT(20,0);REF Y

The above statement will be interpreted as: draw the defined triangle in a conceptual drawing space centered about the origin, position it at location 20,0, and then reflect about the -Y axis. See Figure 4.

LINE TO 10,10 FROM 5,5 ; WINDOW(7:9,7:9)

The above statement defines a line to location 10,10 from 5,5 , but it is windowed by the box with the corners (7,7),(7,9),(9,9),(9,7). Therefore, only a line from 7,7 to 9,9 will be drawn. See Figure 5.

Fig. 3

Fig. 4

All the graphical modifiers may be applied to a string of characters as output by a TEXT or LIST statement. The transformation operators do not operate on the individual characters but rather on the string as a whole. The statement:

```
TEXT 'ABCD' ; ROT(45); AT(500,500)
```

will put the string of characters 'ABCD' on the screen at position 500,500 tilted by 45 degrees, but each character will be upright. (See Figure 6) This provides considerable efficiency in generating characters on most display devices, as only the starting position of each character needs to be altered. Most hardware character generators only produce upright characters and could not be used otherwise. No hardware dependencies are introduced by this method of rotating character strings because it can be easily realized on any display device. The method of rotating character strings as pictures are rotated is machine dependent as it requires that characters be generated in a manner so that they can be individually re-oriented.

In order to understand the detailed operation of the transformation and viewing commands it is necessary to have a conceptualization of what happens in a drawing operation. All picture elements (lines, etc.) are described on a conceptual drawing space centered about the origin. This drawing is built using the drawing commands POINT, LINE, LINE THRU, LINE TO, TEXT, LIST, and SYMBOL either directly or indirectly by calling a subprogram. The picture element so defined can be modified.

using Graphical Drawing Modifiers to produce a picture defined in the conceptual space (Fig. 7 part a). This picture is then WINDOWed or clipped. The full screen size has been defined by the PAGE definitional command or is assumed to be the default 0:1000. Therefore, the default window is clipped at the dimensions of the edge of the screen (Fig. 7 part b). This WINDOWed drawing may then be viewported. The viewport command (WITHIN) will automatically scale a WINDOWed drawing so that it will fit WITHIN a defined area on the conceptual screen (Fig. 7 part c). If the viewport area or window area are not the same shape, the WINDOWed area will be fitted into the viewport area starting in the lower left hand corner. There will not be any distortion on the pictorial image. Another transformation will automatically be performed to change the co-ordinates specified in the PAGE command into the real hardware co-ordinates of the system. This transformation is device dependent but is transparent to the application program.

The defaults have been defined so that there is a minimum of transformation calculations required to put an object on the screen. If no windowing or viewporting is requested, none is done. The default windowing at the edge is done as a clipping operation just before the image is finally put on the screen.

As will be indicated in the following section on procedures and subprograms, (Sect. 4.3.4), it is permissible to apply modifiers to blocks of display commands grouped as a display procedure or within a subroutine. Since display procedures may invoke other display procedures to any depth, it is possible to multiply modify the effect of a display command. This capability is powerful and it allows quite complex structures to be easily defined. It can also be implemented in an efficient manner. A single transformation matrix is constructed from the concatenated string of transformation modifiers and the most current line texture or character set modifiers are in effect. Viewing modifiers present a problem. If a section of a picture is clipped, transformed and then clipped again, there is no possible mathematical method which would avoid multiple levels of clipping, or complex shape clipping. In order to allow an efficient single clipping routine to be implemented, it is not permissible to use multiple WINDOW or VIEWPORT statements on one section of a picture, or to apply transformation operators after a WINDOW or VIEWPORT statement. It is permissible to apply different window statements to independent portions of a picture. Foley and Van Dam [24] point out that: "We have yet to see any area in which this restriction imposes a hardship".

4.3.4 PROCEDURES AND EXTERNAL SUBPROGRAMS

A local subroutine capability is useful to prevent the duplication of code and to allow the definition of display procedures. Common groups of commands could be written as an external subroutine, but this involves the overhead of parameter passing. A local subroutine or 'PROCEDURE' is more readable since it exists within a program, and it provides certain efficiencies if it has global access to all program variables as well as a specific parameter capability.

A display procedure [18] is a programming construct which groups together a set of instructions for picture drawing. This set of instructions may include primitive line drawing instructions or calls to other display procedures or external subprograms. The important feature of a display procedure is that the call to the procedure may modify the result of the procedure. For example, if a procedure has been defined to draw a square, then multiple instances of squares of different sizes and orientations may be placed on the screen by multiply invoking the procedure with different modifiers.

The display procedure facility in IMAGE is similar to that of EULER/G and to the function capability of GRAPPLE. It provides the key to a powerful picture description syntax because sub-pictures may be defined and invoked as if they were primitive drawing commands.

A PROCEDURE is called by the 'DO' statement which has the form:

DO procedure [(parameters)][[graphical modifiers; ...]

- invokes the named procedure with the specified parameters. An 'AT' phrase may set the drawing beam position before entering the procedure. Other graphical modifiers may be used to set the default display conditions for graphical drawing commands within the procedure.

The definitional statement PROCEDURE defines a procedure block. PROCEDURE blocks may be placed anywhere within the body of a program except between OBJECT blocks and their associated ACTION blocks. They may be defined before or after they are used and may invoke other PROCEDURES.

In a special purpose language such as a graphics language it is necessary to have easy access to external programming systems and languages. A general subprogram calling sequence is provided here which is similar to that of FORTRAN. A subprogram is called by the statement CALL subprogram name [(parameters)]. All subroutines CALLED must be declared in a declaration at the beginning of the program. Access to variables in other subprograms is also provided by the use of an 'EXTERNAL VARIABLES' statement.

A set of subprograms are available in a system library to provide standard system functions, and to make graphics capabilities available to external programs (See Appendix D). External programs may also call graphic subprograms in order to make use of some special features of the graphic language. A

graphical subprogram which is called to make use of some special executable statement would require only an ENTRY block.

The subprogram call has the form:

CALL subprogram [(parameters)][;graphical modifiers: ...]

- executes the named subprogram with the specified set of parameters. The modifier phrases behave as in the procedure calling sequence.

A graphical subprogram may be called by another graphical program. The graphical subprogram may clear the display screen with a DISPLAY command or it may append objects to the screen. It could also consist only of an ENTRY block and a RETURN statement. The following five examples will illustrate these cases:

(1) Replacing the current light-button menu with one defined in another subprogram.

Upon execution of the CALL statement, control is transferred to subprogram SUB1. The display screen is erased by the DISPLAY command and all OBJECTS defined in the subprogram are executed, creating a new page of graphical images. On return from the

subprogram it is necessary to erase this new page and re-create the old page by using a command such as DISPLAY. It is not meaningful to execute a call to a graphical sub-program from within an OBJECT block, which ~~erases~~ the display screen.

(2) Including a sub-menu defined in an external subprogram within the current menu.

In the above example a graphics subprogram is called from within an object block, but the subprogram does not erase the screen. It causes the execution of all the object blocks within the subprogram and connects them and their associated action routines to the identifier interrupt structure. If any of the OBJECTS within the subprogram is identified, the associated action routine is executed and then the ACTION associated with the OBJECT which put this subprogram on the screen is executed. The OBJECTS within the subprogram can be considered as sub-OBJECTS to the OBJECT which issued the CALL. In this manner predefined sections of program may be used. It is necessary to use a RETURN statement following the APPEND statement in order to continue executing the OBJECT blocks in the main program.

(3) upon an identifier strike on the current menu, appending a sub-menu defined in an external program to the current menu.

In the above situation the program SUB3 is executed only after an identifier strike on object A. The subprogram draws new OBJECTs and associates them with the identifier interrupt structure. These OBJECTs are made active after the general SEEK statement in the main routine.

(4) Including a section of pre-defined display code in the current program.

The above subprogram SUB4 is called only to allow the execution of the statements within its entry block.

In a graphics programming environment it is necessary to provide easy access to external routines in order to perform specialized graphical operations. These library subprograms have

a general calling sequence and may be invoked by any standard language supported by the operating system. Some of the features of the graphics language are also provided so that images may be drawn by calls from standard programming languages. The following routines are typical of those provided in the library. (Reference Appendix D).

SET (x,y)	- position the beam
TEXT (n,array)	- display the n characters of textual information stored in the referenced array.
LINE (dx,dy)	- draw a line by the relative displacement dx,dy
LINETO (x,y)	- draw a line to the position x,y
PLOT	- copy the image on the screen onto the hardcopy plotter.
CIRCLE (radius)	- draw a circle of a given radius.

4.3.5 INTERACTION CONTROL

Interaction with the graphics display can be attained through the use of many hardware input devices such as tablets, light-pens, knobs, switches, pushbuttons, and keyboards, but whatever the device there are only a few basic modes or classes of interaction. A real hardware device may be ideally suited to one class of interaction or it may be able to handle several classes of interaction with varying ease. The actual hardware

devices used are determined at load time by device assignments. Conceptually, this language recognizes six input functions and associates with them six virtual input devices. These six input functions can be further divided into two groups. Three input functions are general in nature and could be associated with a non-graphical language (but only the keyboard usually is). The other three functions relate specifically to the graphics display. The general input functions are:

- Input of textual strings (with or without a carriage return delimiter).
- The use of switches or pushbuttons.
- Input a numeric value using a 'valuator'. A 'valuator' [26] is a device which allows the input of a number without the need of syntax checking. The number is input by some type of continuous device such as a potentiometer.

The specific graphic functions are:

- Identify specific OBJECTs drawn on the display screen.
- Position a marker to a position on the display screen.
- Sketch (accumulating the x,y positions visited).

There are six virtual devices, each ideally suited to performing a particular input function. The real device assigned at run time to perform the task of a virtual device emulates the function in the best manner possible.

The virtual devices are:

FUNCTION =====	DEVICE =====
Input of textual strings	KEYBOARD
Pushbuttons or switches	PUSHBUTTONS
Input a numeric value	VALUATOR
Identifying	PICKER
Sketching	DIGITIZER
Positioning	LOCATOR

In the class of graphical input functions it is possible to interchange the associated device by command within the program. By using suitable software techniques, any of the three virtual graphic devices can be made to perform any of the three graphical functions. The default condition is for the PICKER to be used for identifying, for the DIGITIZER to be used for sketching, and for the LOCATOR to be used for positioning. This gives the programmer the ability to control the form of input he requires. The following executable commands allow the runtime specification of the graphical input device.

```

IDENTIFY [ USING ] : [ PICKER ] *
                  [ DIGITIZER ]
                  [ LOCATOR ]
and
POSITION [ USING ] : [ LOCATOR ] *
                  [ PICKER ]
                  [ DIGITIZER ]
and
SKETCH [ USING ] : [ DIGITIZER ] *
                  [ PICKER ]
                  [ LOCATOR ]

```

There may be only one device active at one time per input function, but one device may be assigned to several functions. If there is not an explicit definition then the devices are

default. The defaults have been marked above with an asterisk.

For example:

IDENTIFY USING : PICKER

This command causes the virtual device PICKER to be used for identifying OBJECTs. A real hardware device which can handle the task of picking should be assigned to the virtual device PICKER at load time. What the real devices are and how well they match the virtual tasks is an implementation consideration. For example, a light-pen is an excellent PICKER, but the task can be adequately accomplished using one of several other devices.

A PICKER is particularly good at identifying OBJECTs, while a tablet is particularly good at providing absolute x,y positions and is a good digitizer. With certain software techniques it is possible to indicate OBJECTs with a tablet or draw with a light-pen. A 'track-ball' [8], a 'joy-stick' [8] or the Stanford 'Mouse' [25] are all good examples of locators. The co-ordinates they work in are relative and so a marker must be placed on the display screen to indicate the absolute position referenced. If a digitizer or locator is used to indicate, upon the indication interrupt (pressing a button on the stylus, etc.) a search has to be done either by the systems software or by a hardware device in order to determine what, if anything, was indicated. Drawing or locating can be done with a light-pen by providing a tracking mark which follows the light-pen movements.

4.3.5.1 IDENTIFYING

IDENTIFYing is handled by the OBJECT / ACTION structure of the program. All material displayed on the screen is delimited by a TAG code. Upon an identifier interrupt caused by the identifying instrument 'seeing' the displayed OBJECT, the ACTION routine associated with that OBJECT is executed. Also, if the indicated OBJECT is in a graphics subroutine and is a sub-object of another OBJECT in the calling program, the ACTION associated with that OBJECT is executed. TAGs may be explicitly issued while defining an OBJECT in order to sub-delimit it. During the execution of the ACTION associated with the identified OBJECT, the reserved integer variable 'ITEM' contains the sub-delimiting tag count, indicating which part of the OBJECT was identified. It is also possible to obtain the x,y co-ordinates at which the identifier stylus 'struck' the displayed OBJECT. The executable command,

```
COORDINATES] ( xvar, yvar )  
or  
COORDINATES] ( xvar, )  
or  
COORDINATES] ( ,yvar )
```

obtains this information and stores it in the indicated variables. The screen position obtained is in the drawing co-ordinates in effect at the time.

The following example makes use of both the TAG sub-delimiting feature and the ability to obtain the x,y co-ordinate 'struck' with the stylus. This program draws two

parallel horizontal lines on the screen. Upon a strike with the identifier on either of the lines, a vertical arrow is drawn between the two lines to mark the indicated spot. The arrow points up or down depending on which line was indicated. The diagram below shows how the screen would look after the upper line was struck in the middle.

```

INTEGER X
OBJECT
  LINE 500,0 AT(100,200) ** DRAW THE FIRST LINE
  TAG ** DELIMIT WITH A TAG
  LINE 500,0 AT(100,100) ** DRAW THE SECOND LINE
ACTION
  REMOVE ** ERASE ALL MATERIAL
  * ** DRAWN BY PAST ACTIONS
  COORD (X,) ** GET THE X POS. INDICATED
  IF (ITEM = 1) ** UPPER LINE ?
    DO ARROW AT(X,100) ** YES, DRAW ARROW UP
  ELSE ** NO, THEN LOWER LINE.
    DO ARROW ;ROT(180);AT(X,200) ** DRAW ARROW DOWN
  FIN
*
PROCEDURE ARROW
  LINE 0,100 AT(0,0) ** DRAW THE VERTICAL SHAFT
  LINE -10,10/-10,-10 AT(10,90) ** DRAW THE ARROW HEAD
END

```

At times the action which results from a identifier interrupt produces no visual feedback to the user. Visual feedback is important because the user needs to know whether or not he actually 'hit' the particular light-button. The 'WINK' command provides him with this capability.

WINK

- turns off the display for a fraction of a second in order to alert the user. Execution of the ACTION routine continues during the duration of the wink, except for the execution of a command which generates displayable material. Execution of the ACTION routine is suspended upon a command which generates displayable material until the wink is over.

It is the programmer's responsibility to issue this command. A wink feature cannot be automatically incorporated into the system because of the annoying delay it might introduce into some forms of interaction.

4.3.5.2 SKETCHING

When a user sketches on the graphics display screen in a free hand drawing mode he accumulates data as to where the stylus has been. The sketching mechanism provides him with a queue of the x and y positions visited. Every valid point accumulated is stored in this queue in a first in, first out FIFO manner. Only minimal filtering is done in the system software. An x,y position is accepted as valid only if it differs from the last valid point by a given amount. This amount is an insensitive zone about the last valid point and it may also be specified in the SKETCH RESOLUTION command. If it is not specified, a default value of .002 of the screen width is assumed. By increasing the zone of insensitivity the queue will accumulate fewer points which more coarsely fit the actual drawing.

There are two methods by which a digitizer can be used. Points may be accumulated on a point-by-point basis by touching the stylus only to specific points, or an outline may be traced with the stylus. In both cases the same queue of points is maintained, but the amount of data is drastically different. It

is the responsibility of the programmer to provide some visual feedback to the user, and also to provide a method of controlling the amount of data. The queue assigned will not overflow. Data accumulation will stop when it is full. The programmer should provide some data smoothing or similar algorithm to consume this data. The following commands allow control of the sketching mechanism.

SKETCH [ON]
or
SKETCH OFF

- disables or enables the sketching mechanism. The sketching mechanism is normally off and must be turned on in order to be used. The contents of the queue is not affected by this command.

SKETCH RESOLUTION] (var)

- defines the zone of insensitivity about the last valid point as a square of $2 * \text{'var'}$ on a side. 'var' is an integer or real variable representing a value in the current screen co-ordinates.

SKETCH QUEUE] (var)

- this command is a definitional command and must be placed at the beginning of the program if a queue of other than 50 points is required. Fifty is the default size of the queue of points obtained from drawing. The size of the queue used for a particular task should be determined not from the maximum number of points to be entered, but from the expected statistical delay for points waiting on the queue.

SKETCH LOCATION] (xvar, yvar)
 (xvar,)
 (,yvar)

- Obtain the x, y location visited by the drawing stylus, which is now at the top of the queue. If there is a data item in the queue it is put in the variables xvar and yvar and removed from the queue. If there is no data in the queue, execution halts at this point until a data item becomes available. It is then placed in variables xvar and yvar and execution continues.

SKETCH (QUEUE) CLEAR

- this command clears the drawing queue. It can be used to re-initialize drawing.

The following program illustrates the use of some of the commands described above. The purpose of this example is to allow sketching in a free-hand manner on the display screen. A line will appear on the screen inking in the points visited by the stylus. A coarse resolution of 20 screen co-ordinates on a 1000 unit square screen was chosen so that the number of data points and the rate at which they are collected are not too great. Straight lines are drawn between the points to provide a continuous curve. In order to keep the program simple no facility has been provided to allow 'lifting' the stylus. Only one continuous line may be sketched. The sketching is enabled and restarted by touching the light-button 'SKETCH'. The light-button 'STOP' halts execution.

```
INTEGER X1,X2,Y1,Y2
SKETCH QUEUE (20)  ** DEFINE THE LENGTH OF THE SKETCH QUEUE
*
ENTRY
  IDENTIFY USING : PICKER
  SKETCH USING : DIGITIZER
  SKETCH RESOLUTION (20)
*
OBJECT
  TEXT 'SKETCH' AT(900,700)
ACTION
  SEEK          ** ENABLE THE IDENTIFIER INTERRUPTS SO
                ** THIS ACTION ROUTINE MAY BE ABORTED.
*
  SKETCH Q CLEAR ** REMOVE THE EFFECTS OF PREVIOUS SKETCHES
  REMOVE        ** SO THAT SKETCHING MAY BE RESTARTED.
  SKETCH ON     ** ENABLE THE SKETCHING MECHANISM.
  SKETCH LOC(X1,Y1) ** GET THE FIRST POINT.
  REPEAT
    SKETCH LOC(X2,Y2) ** GET THE NEXT POINT.
    LINE TO X2,Y2 FROM X1,Y1 ** JOIN WITH A LINE
    LET X1 = X2              ** SAVE THIS POINT FOR
    LET Y1 = Y2              ** REFERENCE.
  FIN
*
OBJECT
  TEXT 'STOP' AT(900,640)
ACTION
  STOP          ** TERMINATE EXECUTION
*
END
```

4.3.5.3 POSITIONING

A controllable marker is provided through the positioning facility to allow a user to indicate specific screen locations. The position of this marker is controlled by a device such as a locator in a relative manner. For example, the left-right rotation of a tracking ball would cause the marker to move in a horizontal direction an amount relative to the rotation of the track-ball. Motions are relative to the current position of the marker.

The general distinction between positioning and sketching is

that in sketching the location of the drawing stylus is monitored, while in positioning the location of a marker is monitored. This marker may be positioned on the screen by program control or by the stylus. An important aspect to the positioning facility is that motions may be constrained. A marker may be fixed to indicate a location or constrained to move only in a single direction. A single marker is available on the display and the following commands modify its operation. The default marker is off at the centre of the current screen.

MARKER [AT(x,y)]	{	[; OFF]
[LOCATION] (x,y)]		[; ON]
		[; FIXED]
		[; HOR[IZONTAL]]
		[; VERT[ICAL]]
		[; SLOPE[D] (angle)]

The following examples indicate the operation of the MARKER.

MARKER AT(x,y)

- sets the marker to the position x,y .

MARKER ; OFF

- disables the marker by turning it off.

MARKER ; HOR[IZONTAL]

- turns on the marker and restricts its motion to horizontal

MARKER ; VERT[ICAL]

- turns on the marker and restricts its motion to vertical.

MARKER : SLOPE[D] (angle)

~~turns on the marker and restricts its motion along a~~ line
sloped by an angle of 'angle' degrees. For example, if
'angle' = 0 then the motion is restricted along the
horizontal.

MARKER : FIXED

- displays the marker but does not allow it to move.

MARKER : ON

- enables the display marker with no restrictions on its
motion.

MARKER LOCATION] (vari, var2)

- stores the current marker position in the arithmetic
variables vari and var2. It is not meaningful to use
modifiers with the location phrase.

The following program allows the use of the marker to do
constrained drawing. Only horizontal or vertical lines may be
drawn. In this program the virtual device picker has been
assigned the job of positioning in order to indicate the syntax
of such an assignment. When this program is executed the marker
appears ON and FREE in the default position in the centre of the
screen, and it may be freely positioned to anywhere on the
screen. The first menu item is a horizontal line. A strike on
this OBJECT constrains motion to a horizontal direction. The
character 'O' is placed on the screen to indicate the current
marker position and a scale is drawn on the screen along the
axis of allowed motion. The marker may be moved left or right
along this scale. If either the vertical line or 'FREE'

light-buttons is 'struck' a line is drawn from the position of the marked character '0' to the current marker position, the scale is erased, and the task of the selected light-button is performed. The vertical line light-button allows vertical lines to be drawn in the same manner in which the horizontal line light-button allows the marker to be placed anywhere on the screen. The screen would appear as below after the horizontal light-button was struck.

FREE

INTEGER X,Y,X0,Y0,FLAG
ENTRY

LET FLAG = 0
POSITION USING : PICKER
DISPLAY EXCLUDING SCALE

* OBJECT

LINE 100,0 AT(900,700)

ACTION

IF (FLAG = 0)
MARKER LOC (X0,Y0)

ORIF (FLAG = 2)

ERASE SCALE

MARKER LOC (X,Y)

LINE TO X,Y FROM X0,Y0

LET X0 = X

LET Y0 = Y

FIN

LET FLAG = 1
MARKER HORIZONTAL
DISPLAY SCALE

* OBJECT

LINE 0,100 AT(950,550)

ACTION

IF (FLAG = 0)

MARKER LOC (X0,Y0)

ORIF (FLAG = 1)

** INDICATE THE MARKER IS FREE
** USE THE PICKER DEVICE TO POSITION
** EXECUTE ALL OBJECT BLOCKS EXCEPT
** THE OBJECT NAMED 'SCALE'

** HORIZONTAL LIGHT-BUTTON

** IF THE MARKER IS FREE
** GET THE CURRENT MARKER POS.
** IF THE MARKER IS VERTICAL
** ERASE THE OLD SCALE
** GET THE CURRENT MARKER POS.
** DRAW A LINE FROM THE LAST POS.
** TO THE CURRENT MARKER POS.
** SAVE THE CURRENT MARKER POS.

** IF THE MARKER IS CONSTRAINED
** TO HORIZONTAL, NO CHANGE.
** INDICATE HORIZONTAL
** CONSTRAIN MARKER TO HORIZ.
** DISPLAY THE SCALE ABOUT X0,Y0

** VERTICAL LIGHT-BUTTON

** IF THE MARKER IS FREE
** GET THE CURRENT MARKER POS.
** IF THE MARKER IS HORIZONTAL

```

ERASE SCALE ** ERASE THE OLD SCALE
MARKER LOC (X,Y) ** GET THE CURRENT MARKER POS.
LINE TO X,Y FROM XO,YO ** DRAW A LINE FROM THE LAST POS.
* ** TO THE CURRENT MARKER POS.
  LET XO = X ** SAVE THE CURRENT MARKER POS.
  LET YO = Y
FIN ** IF THE MARKER IS CONSTRAINED TO
* ** VERTICAL ; NO CHANGE
  LET FLAG = 2 ** INDICATE VERTICAL
  MARKER VERTICAL ** CONSTRAIN MARKER TO VERTICAL.
  DISPLAY SCALE ** DISPLAY THE SCALE ABOUT XO,YO
*
OBJECT
  TEXT 'FREE' AT(900,400) ** 'FREE' LIGHT-BUTTON
ACTION
  ERASE SCALE ** ERASE THE DRAWING SCALE
  MARKER LOC (X,Y) ** GET THE CURRENT MARKER POS.
  LINE TO X,Y FROM XO,YO ** DRAW A LINE FROM THE LAST POS.
* ** TO THE CURRENT MARKER POS.
  LET FLAG = 0 ** INDICATE THE MARKER IS FREE
  MARKER ON ** ENABLE THE MARKER WITH NO
* ** RESTRICTIONS.
  OBJECT SCALE ** OBJECT TO DRAW A SCALE
    IF (FLAG = 1) ** HORIZONTAL SCALE
      REPEAT X = 100,800,100 ** DRAW THE TICKS 100 UNITS APART
        LET Y = Y + 10 ** TOP OF TICK
        LINE 0,-10/100,0 AT(X,Y) ** DRAW A TICK AND LINE SECT.
      FIN
    LINE 0,10 ** DRAW THE LAST TICK
    ORIF (FLAG = 2) ** VERTICAL SCALE
      REPEAT Y = 100,800,100 ** VERT. TICKS 100 UNITS APART
        LET X = X + 10 ** TOP OF TICK
        LINE 0,-10/100,0 AT(X,Y) ** DRAW A TICK AND LINE SECT.
      FIN
    LINE 10,0 ** DRAW THE LAST TICK
  FIN
  TEXT '0' AT(XO,YO) ** MARK THE CURRENT POINT
END

```

4.3.5.4 INPUT OF TEXTUAL STRINGS

The input of textual information in the form of character strings can be performed in several ways. Conventional programming languages such as FORTRAN have record-oriented textual input facilities. One string of characters (a record) is entered at one time and execution of the program halts until the entire record is inputted. The end of a record is usually

terminated by a carriage return or a line feed character. This type of input is useful but it is often necessary in a highly interactive environment to perform interrupt driven, no-wait input. A unique feature of this graphics language is its ability to handle interrupts, so interrupt-based input is easily performed. Both record-oriented and interrupt-based facilities to input textual strings are provided.

The keyboard command provides the ability to perform interrupt-based input and is a special case of the OBJECT block as described below.

KEYBOARD
or
KEYBOARD (activation character) [, (activation char), ...]
or
KEYBOARD CHARACTER

-enable interrupts from the keyboard associated with the display. This disables the keyboard's use for other purposes. There are two reserved string variables associated with the keyboard. They are KEYBUF and KEY. KEYBUF is a string buffer of default width 80 characters. If so desired, other widths may be set by explicitly defining KEYBUF. KEY is an automatically defined string buffer of width 1 character. When a character is typed on the keyboard it becomes the current character in the string buffer KEY and it is appended to KEYBUF. The Carriage Return and Line Feed characters are the default activation characters but others may be specifically stated after the word KEYBOARD. If an activation character is typed, the ACTION associated with the OBJECT containing the KEYBOARD statement is executed. If CHARACTER activation is specified, then the ACTION routine is activated upon every character typed. Tests against special non-printing or other characters may be done with an IF statement in the ACTION routine. Special names are provided for the common non-printing characters such as carriage return.

Examples of the use of the KEYBOARD statement are:

OBJECT	
KEYBOARD	** DEFINE THE USE OF THE KEYBOARD
ACTION	** PERFORM THIS ACTION WHEN EITHER
.	** A CARRIAGE RETURN OR LINEFEED IS
.	** TYPED.
.	
OBJECT	
KEYBOARD CHAR	** DEFINE THE USE OF THE KEYBOARD
ACTION	** SINGLE CHARACTER ACTIVATION
.	** ACTION PERFORMED REGARDLESS
.	** OF THE CHARACTER TYPED

There may be only one KEYBOARD OBJECT active at any one time. If there is a conflict because the keyboard has been requested in more than one place, the ACTION routine which will be executed is associated with the last OBJECT to request the keyboard.

Record-oriented input is provided through the use of the INPUT statement. An INPUT statement reads in a single record delimited by a carriage return or line feed character according to the optional format list in the AS phrase. Refer to Appendix B for a description of the AS phrase and the use of character string variables.

INPUT variable [,variable , ...] [AS (format list)]

-) read from the keyboard associated with the display terminal into the specified variable, performing conversions according to the format list if it is specified. A WAIT is issued and execution of the program is suspended until the requested number or character string is typed. Carriage Return and Line Feed are the activation characters. Execution continues on the next statement after the INPUT statement is completed. Like any other statement, an INPUT may be aborted by an interrupt on another OBJECT. Because of the wait involved, an INPUT statement should not be in an OBJECT block. Issuing an INPUT statement disables the special OBJECT KEYBOARD until the INPUT is completed.

4.3.5.5 PUSHBUTTONS

A pushbutton is a highly interactive input device. By pushing a button a user desires to indicate that some particular task should be performed. A set of buttons can form a function keyboard to be used to invoke functions or to enable or disable features of the program. A button on a light-pen can be used to indicate to the system information such as: a marker is now positioned to its final position. The operation of a pushbutton is similar to the operation of an OBJECT block within the identifier structure of the program. Displayed OBJECTs are sometimes termed 'lightbuttons'.

A pushbutton is treated as a special type of object within an OBJECT block in the same manner as the text KEYBOARD was treated. Upon a pushbutton 'strike' (push) the ACTION associated with the OBJECT containing the pushbutton statement is executed. Distinguishing one pushbutton from another is a programming problem because there is no standard terminal and therefore no standard location for pushbuttons. One installation may have footpedals while another may have a function keyboard or individual buttons or a combination of these. To avoid this problem at the programming level a simple numbering scheme is assumed. Pushbuttons are numbered from 1 to n, where n is the number of pushbuttons available. Variables can be used instead of numbers in the PUSHBUTTON statement so that pushbutton location variations may be accounted for. The format of the

pushbutton command is illustrated below.

PUSHBUTTON[S] (number, ...)

- enables pushbutton interrupts from the indicated pushbuttons. 'number' may be a single literal integer pushbutton number or an integer variable, or it may be a range specification. A range of pushbuttons is specified in the form 'number1:number2' where number1 and number2 are integer variables or literals indicating the minimum and maximum of the range. It is a runtime error to doubly specify a pushbutton. Upon a 'push' the reserved variable ITEM is set to the pushbutton stimulated. The following example illustrates the use of the pushbutton command.

OBJECT
PUSHBUTTON (6)
ACTION

** ENABLE PUSHBUTTON 6

** PERFORM THIS ACTION ON PB 6

OBJECT
PUSHBUTTON (1:5,7)
ACTION
CASE OF ITEM
CASE 1

** ENABLE PB 1,2,3,4,5,7

** THE VARIABLE 'ITEM'
** CONTAINS THE NUMBER OF
** THE PUSHBUTTON STRUCK;
** USED HERE TO BRANCH TO
** A PARTICULAR PIECE OF
** ACTION CODE.

CASE 2

4.3.5.6 VALUATORS

A valuator is a device which allows the input of numeric information in a manner which requires none of the syntax checking needed in character-oriented operations. [26]. Numbers are input by a continuous device such as a potentiometer and in a range -1 to 1. As a valuator is adjusted, the value is updated continuously. In order for a valuator to be useful it is necessary to provide immediate feedback of the value in a user-defined format. This means that the transformation from the -1 to 1 form must be done rapidly and the resulting number displayed. As the valuator is adjusted the displayed number

should represent the current value. There may be several valuator available on a system at one time and they are distinguished by number. The association of real hardware devices and valuator numbers is performed at load time.

The valuator is treated as a special interrupt-generating object within an OBJECT block in the same manner as the PUSHBUTTON or KEYBOARD devices, but the VALUATOR is not inherently an interrupt device, rather a continuous device. It always has a value associated with it. The ACTION routine associated with the special object VALUATOR is executed if a resolvable change in the value associated with the valuator is observed. Valuator resolution is determined by the hardware device used. The valuator command has the form:

VALUATOR [# num] (var)

- enables the valuator device of number 'num'. The value associated with the valuator will be stored in variable var. The value in var will always be in the range -1 to 1. If the optional number phrase is omitted then the valuator referenced is #1.

The following program segment illustrates the use of the valuator. Note the use of the RESUME command at the end of the ACTION block. Whatever command sequence was in progress when the valuator was adjusted is continued after the valuator ACTION is completed. Since the valuator is a continuous device it is possible for 'noise' to produce a resolvable change in the value and therefore cause a valuator interrupt at a random time. The 'RESUME' command is necessary to prevent such an interrupt from

aborting another action.

REAL V,C
STRING(6) COST

*
* THIS OBJECT BLOCK DEFINES A VALUATOR WHICH CAN BE USED TO
* ADJUST A VARIABLE COST FROM 0.00 TO 100.00 DOLLARS
*

OBJECT

VALUATOR: (V)

** DEFINE THE VALUATOR

ACTION

LET C = (V+1)* 50

** TRANSFORM THE VALUE

REDISPLAY VALUE

** REDISPLAY THE VALUE.

RESUME

** RETURN TO WHATEVER SEQUENCE

** OF INSTRUCTIONS WAS

** INTERRUPTED BY THE

** EXECUTION OF THIS ACTION.

OBJECT

TEXT 'COST = \$' AT(840,900) ** PUT A TITLE ON THE

* ~~**~~ VALUE.

OBJECT VALUE

PUT C INTO COST AS (\\\\.\\) ** FORMAT AS CHARACTERS

TEXT COST AT(900,900) ** DISPLAY ON THE SCREEN

The function of the valuator can be achieved without a special hardware device by display programming. Newman's 'light handle' [27] and several other dials and sliding scales can be easily programmed to allow the PICKER or LOCATOR devices to be used to input numeric values. The techniques used in the example program in section 4.3.5.1 could be used directly to form a 'light-potentiometer'.

4.4 AN EXAMPLE PROGRAM

The following program illustrates the use of 'IMAGE'. The program generates a menu of three light-buttons and displays a picture on the screen of a stylized lunar lander and command ship. The first light-button causes the lander to rotate by 10 degrees counterclockwise. The second light-button moves the lander to the left, towards but not beyond, the command ship, and the third light-button moves the lander to the right, away from the ship. Successful docking is indicated when accomplished. Figure 8 indicates the initial status of the screen.

Figure 8 Initial Status of the Screen for the Example Program

* AN EXAMPLE 'IMAGE' PROGRAM == LUNAR LANDER DOCKING ==

*
PAGE 0:1000 ; ** OPTIONAL SCREEN SIZE DEFINITION
INTEGER X ** DEFINE ALL VARIABLES TO BE USED
REAL ANGLE

*
ENTRY ** EXECUTED ON ENTRY TO THE PROG.
LET ANGLE = 0.0
LET X = 500 ** INITIALIZE VARIABLES
DISPLAY ** EXECUTE ALL OBJECT BLOCKS
SEEK ** ENABLE IDENTIFIER INTERRUPTS
WAIT ** SUSPEND EXECUTION

*
* DEFINE THE MENU OF LIGHT-BUTTONS
*

OBJECT
TEXT 'ROTATE LANDER 10 DEGREES' AT(900,600)
ACTION
LET ANGLE = ANGLE + 10.0 ** UPDATE THE ANGLE
IF (ANGLE = 360.0) LET ANGLE = 0.0 ** MODULO 360
REDISPLAY SPACE ** ERASE AND RE-EXECUTE OBJECT 'SPACE'

*
OBJECT
TEXT 'MOVE LANDER LEFT' AT(900,450)
ACTION
IF (X > 420) ** PREVENT CRASH BETWEEN SHIPS
LET X = X - 20
FIN
REDISPLAY SPACE

*
OBJECT
TEXT 'MOVE LANDER RIGHT' AT(900,400)
ACTION
IF (X <= 980) ** REMAIN ON THE SCREEN
LET X = X + 20
FIN
REDISPLAY SPACE

*
OBJECT SPACE ** DEFINE THE NAMED PICTURE
LINES 0,80/60,-30/20,0/0,-20/-20,0/-60,-30/-10,20/0,40/
& 10,20 ; AT (300,400) ** DEFINE THE COMMAND SHIP
DO LANDER ; ROT(ANGLE) ; AT (X,440) ** DRAW THE LANDER
IF (X = 420.AND.ANGLE = 0.0) TEXT 'DOCKED' AT (400,200)
** INDICATE DOCKING
* DEFINE THE LANDER AS A PROCEDURE SO IT MAY BE TRANSFORMED AS A UNIT
*

PROCEDURE LANDER
LINES THRU 0,0/ 0,20/-20,40/-40,10/-40,-10/-20,-40/0,-20/0,0
LINE -30,20 AT (30,-40)
LINE -30,-20 AT (30,40)
LINE 0,20 AT (30,-50)
LINE 0,20 AT (30,50)

*
END

5. IMPLEMENTATION CONSIDERATIONS

In the past, few years the price of graphics hardware has dropped considerably. Devices such as the Tektronix 4013 [28] storage tube display terminal and the D.E.C. GT-40 [29] mini-processor based refresh display terminal are inexpensive enough that many thousands are being installed. With the advent of graphics on a massive scale, the format of interactive programming has been changing. In the sixties, large time-sharing computers were designed with which users accessed a central computer via teleprinter terminals. The delay in interaction due to the teleprinter terminal itself limited the bandwidth of interaction between the user and the computer. Time-sharing was possible because of the limited CPU load caused by the low bandwidth communication. Graphics devices allow a high bandwidth of interaction, and are therefore not particularly suited for use on a time-sharing computer. A graphics user demands too many of the central computer's resources. This may not be the case if the particular application program is simply a data presentation program requiring few resources, but in general, different architectures are needed in order to provide a graphics user satisfactory response.

Good response is guaranteed with a dedicated machine; however, this was exorbitantly expensive in the past. The current generation of mini-computers is more sophisticated and less expensive. For many applications a dedicated mini-computer

provides an excellent graphics system; however, it may not be capable of the speeds required to perform the calculations necessary in real-time animation and similar applications. The low cost of these machines allows multi-processor systems which can satisfactorily handle these problems to be economical. In the last part of this chapter several architectures will be examined.

There are a large number of unique mini-computer designs. In order to be practical, a graphics language must be easily implementable on most of them. It is desirable to design the translator in such a way that it may be transported from machine to machine. Since it is highly probable that a mini-computer will be the machine on which one would desire the translator to execute, only the available resources of a mini-computer should be used in the implementation of the translator.

5.1 PORTABILITY

The task of writing application programs is greatly eased by the use of a high-level language. The applications programmer does not have to concern himself with the particular details of the machine he is programming. This facilitates the task of programming and it also makes his program portable. If he is writing in the FORTRAN language he is, in effect, programming a virtual FORTRAN machine. Both the internal language statements and the I/O commands have been standardized to produce a virtual FORTRAN environment which may be implemented on various machines. This has not been the case for graphics. Graphics

programs have tended to rely on the particular display hardware of a given machine, thus making it difficult to define a virtual graphics environment. It was only possible to achieve machine independence, not device independence.

The graphics language defined in chapter 4 performs graphical I/O through six virtual input devices, and allows drawing on a generalized display screen. The internal commands of 'IMAGE', including the character manipulation command, are machine independent; therefore this machine and device independence allows the syntax of 'IMAGE' to be visualized as defining a virtual graphics machine. Programs written in 'IMAGE' should operate on any machine which has an implementation of 'IMAGE'. The only differences between implementations would be efficiency considerations.

The first level of software portability requires that the syntax of the language in which the application programs are written is machine and device independent, but in order for an applications program to operate on many machines, it is necessary to have many implementations of the language. It is a big task to implement a language and one would hope that there is a way to avoid doing it many times. The next level of software portability involves making the implementation portable.

In order to write portable software, one must be concerned with the hierarchy of modules within a program package. There

are basically three layers of software, and only the bottom layer needs to interface with the real machine. This hierarchy is shown by the following pyramid:

a) On the top layer is the applications program, which is written in a machine and device independent language. The layer of software below it presents the programmer with a virtual machine. Some form of compiler, translator or interpreter is used to make this program execute using the instructions of the real machine. This compiler, translator or interpreter must also be written in a portable manner if the entire programming system is to be portable.

b) A run-time support package is required to supply the intrinsic functions used by the language. For example, in the 'IMAGE' system, the routine used in order to identify which ACTION block to execute upon an identification interrupt, as well as routines to convert variable data types, are run-time support routines. In order for the system to be portable, these routines must be written in a portable manner.

c) Since the software system must eventually interface with real I/O devices, some software must be written which is particular to that device. The programmer has a well-defined set of attributes for each virtual device. A program must be written for each particular device and machine which causes the real device to emulate the virtual device. The commands to this program or set of programs can be very well defined. In the 'IMAGE' system, the commands at this level form the GII code (Graphics Task Instructions), which are a set of instructions defining each task to be performed by the virtual display device. Typical commands are: to draw a line with a specific set of graphics attributes; to erase a group of lines from the screen; or to read a set of X,Y co-ordinates from an input device.

2 OF/DE 3

d) There must always be a few routines which interface directly with the particular operating system or machine features. An example of this is the initialization and control of the machine's interrupt hardware. These routines are not portable, but they are few and well-defined, so there is not too much work involved in re-writing them.

Several methods are available with which to convert the applications program written in the high-level graphics language 'IMAGE' into code which will execute on a particular machine. Probably the easiest to implement is an interpreter. Since an interpreter effectively executes the source code by interpreting it command by command and making calls to the run-time support package, the task of scanning the source code is executed many times for commands which are multiply executed (in loops, etc.). A prime criteria for a good graphics language is fast response, and an interpreter is not capable of providing the best possible response. A compiler takes source code as input and produces object code (machine instructions) as output. A translator takes source code as input and produces an equivalent program written in another code or language as output. Most machines have an assembler language and an available assembler. If one arranges that the output code of the translator is the assembler language of a particular machine, then a translator and an assembler can perform the same task as a compiler. The time required to perform a translation and then an assembly may be longer than that required to do a compilation; but the result is the same. Because a higher level code and not absolute machine code is output by a translator, it is easier to write a portable translator than a portable compiler. If the output code of a translator is a 'universally available' high-level language such

as FORTRAN, then only a much simpler translator need be written because many of the facilities of the target high-level language may be used. Of course, portability is restricted to those machines which have an implementation of the target language.

The translator itself must be written in some language. If it is written in the assembler language of the machine, it defeats the purpose of portability. A class of languages exist for generating translators called 'Translator Writing Systems'. These languages allow the definition of a translator by defining the syntax and semantics of the source and target languages. Since a translator generator is itself a translator, it can be written in itself and is therefore potentially highly portable. Unfortunately, the run-time support package associated with most translator generators is too large and complex to be easily transported, especially to a mini-computer. The author only knows of large machine implementation of systems such as XPL [30], NEST [31], or BCPL [32].

In the following discussion, a few translator writing systems will be outlined and their portability features presented. As portability is the prime consideration, powerful but machine dependent systems such as XPL will not be discussed. Efficiency, and therefore parsing algorithms, are of only secondary concern. What is most important is to make it possible to easily implement the source language on a variety of computers. If 'IMAGE' becomes a high volume production language on a particular machine, then it should be implemented in as

efficient a manner as possible with no concern as to machine independence.

W. Waite of the University of Colorado, Boulder, Colorado, U.S.A., has designed a truly mobile programming system (MPS) [33]. The core component of the MPS system is a macro processor 'STAGE2'. In a macro processor, a line of input code is first read, then a series of templates are fitted to the line until a match takes place. The line is then expanded into one or more new lines which are optionally re-submitted to the template matching algorithm.

The 'STAGE2' macro processor is written as a series of macro templates for the 'STAGE2' macro processor. The run-time support package for 'STAGE2' consists of a group of routines written for the abstract machine 'FLUB'. Character strings, trees and integers are the basic data items of 'STAGE2' and 'FLUB' must provide instructions for handling them. The basis of much of the 'FLUB' machine design is the operating structure of SNOBOL4 [34]. 'FLUB' is an idealized character-manipulative machine with a word consisting of three fields; a flag field (FLG), a value field (VAL) and a pointer field (PTR). Only 28 short primitives need to be implemented in the assembler code of a machine in order to have 'FLUB' operating on that machine. Since 'FLUB' forms the run-time support section of 'STAGE2', 'STAGE2' is therefore a portable translator generator. A portable graphics language can be written by using 'STAGE2' as the tool with which the translator is implemented.

The 'STAGE2' macro processor can be implemented on a machine by coding the 28 primitives of 'FLUB' in assembler language for the target machine and then passing the macro template for the 'STAGE2' through 'SIMCMP'. 'SIMCMP' is a specialized macro processor written in 91 lines of A.S.A. FORTRAN IV, which is sufficient to translate 'STAGE2'. If FORTRAN is unavailable, then 'SIMCMP' is short enough to be hand coded in assembler.

The 'STAGE2' macro processor has the virtue of being small enough that it can be implemented on a mini-computer. Maclean implemented it on a 16K PDP-9/15 computer [1,2], with success, and it has been implemented on more than eleven other machines of varying sizes, including PDP-11. The 'ICPL' language [2,3] has been implemented as a translator using 'STAGE2'.

The 'STAGE2' macro processor is portable and can therefore easily be made available on any machine on which one wishes to implement 'IMAGE'. This only provides a portable tool for writing a translator for 'IMAGE', not a portable translator. A translator can be conceived of as a 'black box' into which is fed the syntax of a language and which outputs assembler language for the target machine. This is obviously not portable because code is generated for a particular target machine. A general intermediate language, which embodies the information normally passed from the analysis of a compiler to the code generators, is needed. A portable translator can be written which generates a program written in the intermediate language. It is then necessary to provide another translator to convert a

program written in the intermediate language into the assembler language of the target machine. This split allows the structural language decisions to be carried out in the portable translator and the representational decisions to be performed efficiently for a particular machine.

If we use a 'universally available' high-level language such as A.N.S.I FORTRAN for the intermediate language, and accept the inefficiencies of doing so, 'IMAGE' can be written and made portable quite easily. The high-level target language may not be capable of performing all the functions required of it. A large number of specialized subroutines would have to be included in the run-time section in order to provide a language like FORTRAN with the capability of drawing, manipulating character strings and processing interrupts, etc. These facilities are, of course, directly available in an assembler language.

'JANUS' [35] was designed to be a universal intermediate assembler language. The primitive components are those that are basic to all assembler languages, such as addressing and assignment. Constructs such as subroutine calls that must be dealt with in a special way on certain machines are handled by action descriptors. 'JANUS' was designed for ease in translating into efficient machine code for a variety of hardware.

Up to now, only methods of describing a portable translator have been discussed. Since a translator from 'JANUS' code to assembler language must be written as part of the above

structure, it is also available to make the run-time support structure portable.

The steps required to transfer the 'IMAGE' graphics language from one machine to another using 'JANUS' as the intermediate language consists of the following:

- 1) The macro processor 'STAGE2' must be implemented for the target machine. To start this, a simple macro processor equivalent to approximately 91 lines of FORTRAN must be implemented. If FORTRAN is available, it should run immediately. The 28 primitives of 'FLUB' must be coded for the target machine and a simple I/O package written. FORTRAN's I/O package may be used but it is usually too inefficient. A man week of work is typically necessary in order to implement 'STAGE2' and possibly another two man weeks are necessary in order to make it run efficiently.

- 2) The 'STAGE2' macros to translate 'JANUS' code to the assembler language of the target machine must be written. Weber [35] has indicated that approximately two man weeks of work are needed to generate the primitives for 'JANUS'. He has also indicated that the inefficiencies introduced by the use of an intermediate language are small. In various tests performed, only a 10% speed penalty and no size penalty were introduced.

- 3) The interface with the monitor consists of a small number of specialized utility routines and a host program. The host program must be a standard program for the target machine which can be invoked by the target machine's operating system. These routines are straightforward and little effort is required in writing them.

- 4) The major portion of the work required in transporting 'IMAGE' is the rewriting of the display device handler. The GTI interface communications code provides a well-defined set of tasks which must be performed by the display device. A careful design must be done on the architecture of the display handler. In fact, this handler could be entirely implemented in specialized hardware for optimal response.

'BCPL' [32] is a tool for compiler writing which has the characteristic feature of using the binary bit pattern as its only data type. Like 'STAGE2' it is built around the virtual

machine concept and makes use of this to define an intermediate language. The OCODE [35] intermediate language consists of a memory, a processor and two memory address registers. It is a simple language whose statements cause transformations on an imaginary stack. In order to transfer 'BCPL' to a new machine, a new code generator for OCODE and a suitable interface with the new operating system must be written. Richards [35] predicts a one to five man month effort would be required to transport BCPL and states that it has been moved to "between ten and twenty different machines".

Both 'SNOBOL4' [34] and 'PASCAL' [36] have also been implemented using specialized intermediate languages in order to facilitate portability. Griswold [35] lists ten machines on which the intermediate language 'SIL' has been implemented and used as a base for SNOBOL4. An alternate approach has been used to implement the 'SIMPL' [37] family of languages. First a compiler for a base language is implemented (presumably by hand in assembler code) for the target computer. This compiler is also written using the base language so it may compile itself. The language is then extended and the old compiler used for generating a new version of the compiler. By reproducing the history of the extensions of the language from the base language, a compiler for the complete language can be generated. This development history can be quite complex, and it must be modified at each step in order to change the code emitting routines.

Due to the proven capability and availability of 'STAGE2' on a small machine, it appears to be the best candidate for a system in which to write the 'IMAGE' translator. In order to reduce the number of run-time routines which have to be implemented, such as the real arithmetic support package, it is proposed to use A.N.S.I FORTRAN as the intermediate language. The task of writing the translator is greatly reduced since arithmetic expressions do not need to be parsed and symbol ~~tables~~ do not have to be formed for variables. FORTRAN is inadequate to perform some of the tasks required by 'IMAGE' so a specialized set of subroutines are provided. This makes the run-time section more complex, but it is compensated by the greater simplicity in the translator. Due to the block structure of 'IMAGE' the resultant FORTRAN program will require a heavy use of labels and the GO TO statement. This generated FORTRAN program would be difficult for a person to read, but it is machine generated and intermediate in nature and so the use of the GO TO statement is of no consequence.

5.2 RUN-TIME SYSTEM CONFIGURATION

Newman [8] has identified the components of a general-purpose graphics system as shown in Figure 9. There are three types of components: processes, represented by rectangular boxes; data files, shown as boxes with curly ends; and arrows representing information transfers. The application program may contain its own data base, particular to that application. This data base may be generated by a special data base language or a general programming language. From this data base and the

program structure, the application program makes calls to generate a picture. A structured picture definition is formed which may be stored in a second data base. This file is only necessary if one wishes to optimize communications in a dual processor system or to minimize recalculation. It is not usually worth the amount of space it requires. Transformation routines and the display handler operate on this picture definition to generate a low-level, machine-dependent display file. This file is then executed by a display processor unit (DPU) or equivalent. Identification interrupts to this system require information from the display file in order to determine which displayed objects caused the interrupt. The 'IMAGE' language provides great freedom in the definition and transformation of pictures and therefore requires comprehensive and efficient transformation routines.

The system illustrated in Figure 9 may be configured to execute on a variety of different hardware designs. On a single processor dedicated machine the implementation would appear very much as laid out in Figure 9. It should be pointed out that a display file describing what is on the screen is necessary even if a storage tube is used if any picture editing is to be done. Otherwise, the entire picture must be recalculated if an entity is erased.

Figure 10 describes a dual processor graphic system. The second processor frees the main computer from the task of generating the display file and from handling device interrupts.

The main CPU may be calculating the next subpicture to draw while the second processor is drawing the present one. Special hardware and tightly-written software available on the second processor can provide a considerable speed advantage. The initial implementation of the 'IMAGE' language will be on a dual processor system using a PDP-9 computer as the main computer and a PDP-11/GT-40 as the second processor.

A clean interface between the display handler and the 'IMAGE' language has been defined. GTI (graphical task instructions) represent an extensible set of commands for an idealized graphics terminal. It is the task of the display handler for a real set of hardware to emulate the functions of the virtual terminal defined by the GTI commands. See Appendix E for a list and description of the GTI commands.

Figure 9 Conceptualization of a Graphics System

Figure 10 A Dual Processor Graphics System

6. CONCLUSIONS

This thesis presents the syntax and design philosophy of an interactive computer graphics language. This language has been specifically designed for interactive graphics applications and as such contains a powerful set of display creation, editing and interactive control instructions. The primary emphasis of the design is on solving the man-machine interaction problem. The OBJECT / ACTION structure and the interaction control mechanisms supplied, provide a powerful and easy-to-use tool for solving this interaction problem. The OBJECT / ACTION ideas in Maclean's 'ICPL' [1,2,3] and the display procedure structure of Newman [18] were assimilated with a conceptualization of the manner in which a man interacts with a machine, based on work by Foley and Wallace [26] to produce a device independent method of controlling man-machine interactions. The second emphasis of this design is on providing a good picture definition language. The procedure and function oriented approaches of Newman's EULER/G [17] and Scott's GRAPPLE [4,5] were combined with the structured programming concepts of IFTRAN (Bezanson [22] and Miller [20,21]) to produce an easy-to-use picture grammar.

Chapter 3 presents a set of facilities which are desirable in a graphics language. IMAGE has been designed especially to provide these facilities. Graphical input response facilities are provided in a device independent manner through the use of six virtual input devices. A SKETCH and a POSITION mechanism

5

allow X,Y co-ordinates to be input, while an IDENTIFYING mechanism based on the OBJECT/ACTION structure provides a powerful interaction mechanism. A KEYBOARD and a PUSHBUTTON mechanism provide interrupt based character and control input, and a VALUATOR [26] allows the input of data in numeric form. Graphical drawing facilities are provided in a procedure-oriented manner with the free use of modifiers allowed. Modifiers may scale, translate, rotate, reflect, clip, change character set, or modify the brightness or line texture of a drawing. The IMAGE language is block structured and does not require the use of labels thereby clarifying and emphasizing the program logic. The CASE, IF ... ORIF ... ELSE ... FIN, REPEAT ... FIN, and WHILE ... FIN constructs control the execution flow within the OBJECT and ACTION blocks, and the interaction structure controls the execution of these blocks. Although powerful data manipulation facilities are not provided, specialized languages providing these facilities can be accessed because easy access is provided to external software.

The unique feature of IMAGE is its marriage of a powerful yet device independent interaction control structure to a display procedure-oriented picture description grammar.

REFERENCES

- 1) Maclean, M.A., "Designing a Language for Interactive Control Programs", 2nd Man-Computer Communications Seminar, 31 May - 1 June, 1971.
- 2) Bown, H.G., Hartman, W.A. and Warburton, R.E., "Applications of the Interactive Computer Graphics Language, ICPL", NRC 3rd Man-Computer Communications Seminar, May, 1973.
- 3) O'Brien, C.D., "Implementation of the 'ICPL' Graphics Language on the PDP-15 Computer", NRC 3rd Man-Computer Communications Seminar, May, 1973.
- 4) Williams, D.L., "GRAPPLE - Graphics Applications Programming Language", NRC 3rd Man-Computer Communications Seminar, May, 1973.
- 5) "GRAPPLE Language Reference Manual", Bell-Northern Research, Edition 4.0, Sept. 1973.
- 6) "Graphics 15 Programming Manual", DEC-15-ZFSA-D, Digital Equipment Co., Maynard, Mass.
- 7) "MACRO-9/15 Assembler", DEC-15-AMZB-D, Digital Equipment Co., Maynard, Mass.
- 8) Newman, W. and Sproull, R., "Principles of Interactive Computer Graphics", McGraw Hill, 1973.
- 9) Woodsford, P.A., "GINO: Graphical Input/Output", University of Cambridge Computer-Aided Design Group, June, 1969.
- 10) Newman, W.M., "A System for Interactive Graphical Programming", Proc. SJCC, 1968.
- 11) Bork, A.M., "A Graphics Extension of Sigma 7-APL", 19th International meeting of EXCHANGE (the Xerox computer users group), El Segundo, Calif., Dec. 1972.
- 12) "UTS APL Language and Operations Reference Manual", 90 19 31B, Xerox Data Systems, El Segundo, Calif.
- 13) Luehrmann, A.W. and Garland, S.J., "Graphics in the BASIC Language", Computer Graphics, SIGGRAPH-ACM, Vol. 8, No. 3, Fall 1974.
- 14) "BASIC/GT Language Reference Manual", Digital Equipment Co., Maynard, Mass.
- 15) Wirth, N. and Weber, H., "EULER: Formal Definition", Part I: CACM, 9, 1, 13, Jan., 1966; Part II: CACM, 9, 2, 89, Feb., 1966.

- 16) Wirth, N., "A Generalization of ALGOL", CACM, Vol. 6, No. 9, Sept., 1963.
- 17) Newman, W.M., Gouraud, H. and Oestreicher, D.R., "A Programmer's Guide to PDP/10 EULER". Univ. of Utah, Report No. UTEC-CSc-70-105, June 1970.
- 18) Newman, W.M., "Display Procedures", CACM, Vol. 14, No. 11, Oct., 1971.
- 19) Dijkstra, E.W., "A Primer of ALGOL 60 Programming", Academic Press, London, 1962.
- 20) Miller, E.F., "Extensions to FORTRAN to Support Structured Programming (IFTRAN)", SIGPLAN Notices, Vol. 8, No. 6, June, 1973.
- 21) Miller, E.F., "Program Validation Project", General Research Co., 1974.
- 22) Bezanson, W.R., "Teaching Structured Programming in FORTRAN with IFTRAN", Proc. of the 5th ACM SIGCSE symposium, Feb. 1975.
- 23) Mills, H.D., "The New Math of Computer Programming", CACM, Vol. 18, No. 1, Jan. 1975.
- 24) Foley, J.D. and VanDam, A., "Interactive Computer Graphics", Chapter 8, 'Mathematics', unpublished manuscript.
- 25) English, W.K., Englebart, D.C. and Berman, M.L., "Display - Selection Techniques for Text Manipulation", IEEE Transactions on Human Factors, HFE-8, 1, 5, 1967.
- 26) Foley, J.D., Wallace, V.A., "The Art of Natural Graphic Man-Machine Conversation", Proc. IEEE, Vol. 62, No. 4, April, 1974.
- 27) Newman, W.M., "A Graphical Technique for Computer Input", Computer Journal, Vol. 11, May 1968.
- 28) "The Tektronix Family of Computer Display Terminals", Brochure, Tektronix Inc., Information Products Division, Beaverton, Ore.
- 29) "GT-40 Users Guide", Digital Equipment Co., DEC-11-HGTGA-A-D, Sept. 1973.
- 30) McKeeman, W.M., Horning, J.J., Wortman, D.B., "A Compiler Generator", Prentice-Hall, 1970.
- 31) Scott, G., "NEST, The Northern Electric Syntax Directed Translator", Northern Electric Co., 1968.
- 32) Richards, M., "A Tool for Compiler Writing and Systems Programming", Proc. AFIP SJCC, p557, 1969.

- 33) Waite, W. M., "Implementing Software for Non-Numeric Applications", Prentice-Hall, 1973.
- 34) Griswold, R. E., Poage, J. F. and Polansky, I. R., "The SNOBOL4 Programming Language", Prentice-Hall, 1971.
- 35) Coleman, S. S., "JANUS: A Universal Intermediate Language", PHD thesis, Dept. of Electrical Engineering, University of Colorado, Boulder, Col., Report NSF-OCA-GJ32471-J2A.
- 36) Wirth, N., "The Programming Language PASCAL", Acta Informatica, 1, 35, 1971.
- 37) Basili, V. R., "The SIMPL Family of Programming Languages and Compilers", Univ. of Maryland, Computer Science Centre, College Park, Maryland, Technical Report TR-305, 1974.
- 38) Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R., "Structured Programming", Academic Press, London, 1972.
- 39) Knuth, D. E., "A Review of 'Structured Programming'", Stanford Computer Science Dept., Stan-CS-371, June 1973.
- 40) Waite, W. M., "The STAGE2 Macro Processor", Report No. 69-3 and No. 69-3B, Graduate School Computing Centre, Univ. of Colorado, Boulder Col. Jan. and Sept. 1969.
- 41) Waite, W. M., "The Implementation of STAGE2", Dept. of Electrical Engineering, Univ. of Colorado, Boulder, Col., 1969.
- 42) Waite, W. M., "The Mobile Programming System STAGE2", CACM, Vol. 13, No. 7, July 1970.
- 43) Waite, W. M., "Building a Mobile Programming System", Computer Journal, Vol. 13, No. 1, Feb. 1970.
- 44) Thomas, D., Desriviers, J., Radford, F., "Processor Interfaces to Effect Better Utilization of Library Software", EXCHANGE (the Xerox computer users group), El Segundo, Calif., May, 1974.
- 45) Short, G., "Clipping Standard Graphics Interface", Technical Memorandum, Carleton University Computing Centre, Ottawa, Ontario, 1974.

APPENDIX A -- SUMMARY OF COMMANDS

BLOCK DEFINITION STATEMENTS

SUBPROGRAM name [(parameters)]

- define a program to be a subprogram of name 'name'

ENTRY

- define the beginning of an entry block

OBJECT [name]

- define the beginning of an optionally named object block

ACTION

- define the beginning of an action block

PROCEDURE name [(parameters)]

- define the beginning of a named procedure block

END

- define the physical end of the program

TYPE DEFINITION STATEMENTS

STRING (nc) variable name, ... , array name(n1), ...

- define a character string variable or array of 'n1' elements, 'nc' characters long

INTEGER variable name, ... , array name(n1), ...

- define an integer variable or array

REAL variable name, ... , array name(n1) , ...

- define a real variable or array

PARAMETER variable name, ...

- define a dummy variable name for parameter passing

EXTERNAL VARIABLE[S] [/label name/] variable name, ...

- declare the named variables available to other programs. The optional label defines a block of available variables under a label name.

EXTERNAL PROGRAM[S] program name, ...

- define the names of all external programs referenced.

PAGE xmin:xymax

or

PAGE xmax, ymax

or

PAGE xmin:xmax, ymin:ymax

- define the dimensions of the display screen in the co-ordinate system to be used.

CHARACTER SET[S] nameA , nameB , ...

- define the character sets to be used for text

SYMBOL SET[S] name of set, ...

- define the symbol sets to be used

• SKETCH Q[QUEUE] (var)

- define the length of the sketch queue

EXECUTABLE STATEMENTS
EXECUTION FLOW CONTROL

IF (conditional)

FIN

- a simple conditional

IF (conditional)

ELSE

FIN

and

IF (condition) statement

- a two-way conditional

IF (condition)

ORIF (condition)

ELSE

FIN

- a multi-test conditional

CASE OF (variable)

CASE number

CASE number

FIN

- a multi-way conditional where a branch is made to 'CASE' of number if 'variable' is equal to 'number'.

WHILE (condition)

FIN

- a simple loop

WHILE (condition 1)

ESCAPE (condition 2)

FIN

- a simple loop with an escape exit

REPEAT var=strt,max[,incr]

- a loop with built-in counters var is a counter; strt is the starting count; max is the maximum; incr is the increment (default 1)

REPEAT

FIN

- an infinite loop

CONDITIONALS

THE LOGICAL TEST OPERATORS ARE:

=	or	.EQ.
><	or .NEQ. or	.NE.
<	or	.LT.
>	or	.GT.
>=	or	.GE.
<=	or	.LE.

ARITHMETIC ASSIGNMENTS

LET variable = arithmetic expression

or

LET variable, variable, ... = arithmetic expression

- assign the value of the arithmetic expression to the specified variables. An arithmetic expression is composed of literals, variables and the operators + - * / ^ in the normal precedence order.

STRING ASSIGNMENTS

PUT var list INTO variable [,var, ...] [AS (format list)]

- assign the variable list to the referenced variables optionally using the specified format

DISPLAY EDITING COMMANDS

DISPLAY [name, ...]

- erase the display and execute the referenced or all object blocks

DISPLAY EXCLUDING name [,name, ...]

- display all objects except those referenced

REDISPLAY name [,name, ...]

- erase the named objects and re-execute them

APPEND [name, ...]

- execute all or the named object blocks, adding them to the display.

APPEND EXCLUDING] name[, name, ...]

- append all objects except the named objects

ERASE [name, ... , name(sd), ...]

- erase all or the named objects from the display, 'sd' indicates sub-items of a named object.

ERASE EXCLUDING] name [, name, ...]

- erase all objects except the named objects

REMOVE

- erase all material drawn in ACTION routines

SEEK [name, ...] [ON]

- enable the identifier interrupt mechanism for all objects or just the named objects.

SEEK [name, ...] OFF

- disable the identifier interrupt mechanism for all objects or just the named objects.

SEEK EXCLUDING] name [, name, ...]

- seek on all objects except those named

WAIT

- suspend execution of the program waiting for an identifier interrupt

RESUME

- continue execution of the program which was interrupted when the current ACTION routine was entered. A precaution must be taken to prevent non-reentrant programs from being disrupted when action routines execute and then 'RESUME'.

TAG

- delimit displayed material. When part of an OBJECT is identified the particular sub-delimiting TAG number is made available to the program in the reserved variable item.

ADDITIONAL CONTROL INSTRUCTIONS

STOP

- stop execution of the program; close all files on I/O devices; erase the display.

RETURN

- return execution to the calling program. This command is valid only in a subroutine or procedure.

GRAPHICAL DRAWING INSTRUCTIONS

SET AT(x,y)

- set the drawing beam to x,y

POINT [AT(x,y)]

- draw a point

LINE dx,dy [AT(x,y)]

or

LINES dx1,dy1/dx2,dy2/ ... [AT(x,y)]

- draw a line or concatenated lines with the displacement dx,dy

LINES THRU[()U[GH] x1,y1/x2,y2/ ... [AT(x,y)]

- draw concatenated lines through the referenced points.

LINE TO x,y [FROM x1,y1]

- draw a line to a specified location, from the current beam position or the optionally specified 'FROM' position.

TEXT string [AS (format list)] [AT (x,y)]

- display the string of textual material

LIST str var [, str var, ...] [STEP (displacement)] [AT(x,y)]

- display as a column separated by the vertical displacement given (default -50) the referenced character string variables or arrays.

SYMBOL name [AT(x,y)]

- display the referenced symbol

GRAPHICAL DRAWING MODIFIERS

AT(x,y)

- translate the referenced picture element to screen co-ordinates x,y

ROT[ATE] (angle)

- rotate the referenced picture element

SCALE (value)

- adjust the size of the referenced picture element

REF[LECT] X

or

REF[LECT] Y

or

REF[LECT] (angle)

- reflect the referenced picture element about the X or Y axis or about a line tilted by the specified angle.

DOT

or

DASH

or

SOLID

- line texture is solid, dotted, or dashed

INT[ENSITY] (factor 0 to 1)

or

INT[ENSITY] DIM

or

INT[ENSITY] NORMAL

or

INT[ENSITY] BRIGHT

- modify the brightness of the referenced picture element

FLASH [ON]

or

FLASH OFF

- cause or inhibit the referenced picture element from flashing.

or COLO[UIR (factor 0 to 1)
or COLO[UIR RED
or COLO[UIR GREEN
or COLO[UIR BLUE
- adjust the colour of the referenced picture element
CHAR[ACTERS] name of set
- specify a particular character set
WINDOW (xmin:xmax,ymin:ymax)
or WINDOW (xymin:xymax)
- clip the referenced picture element at the specified co-ordinates.
WITHIN (xmin:xmax,ymin,ymax)
or WITHIN (xymin:xymax)
- scale the windowed picture to fit the specified area

PROCEDURES AND SUBROUTINES

DO procedure name [(parameters)] [; graphical modifiers, ...]
- invoke the named procedure
CALL subprogram name [(parameters)] [; graphical modifiers, ...]
- branch to the named subprogram

INTERACTION CONTROL

IDENTIFY [USING] : [PICKER] *
[DIGITIZER]
[LOCATOR]
and
POSITION [USING] : [LOCATOR] *
[PICKER]
[DIGITIZER]
and
SKETCH [USING] : [TABLET] *
[PICKER]
[LOCATOR]

- assign the referenced input device to the referenced function. An asterisk marks the defaults.

COORDINATES] (xvar,yvar)

or

COORDINATES] (xvar,)

or

COORDINATES] (,yvar)

- obtain the screen co-ordinates 'struck' with the identifier

WINK

- turn off the display for a fraction of a second

SKETCH [ON]

or

SKETCH OFF

- enable or disable the sketching mechanism

SKETCH RES[OLUTION] (var)

- define the sketching resolution

SKETCH LOC[ATION] (xvar,yvar)

or

SKETCH LOC[ATION] (xvar,)

or

SKETCH LOC[ATION] (,yvar)

- obtain the x,y locations visited by the drawing stylus

SKETCH Q[UEUE] CLEAR

- clear the drawing queue

MARKER [AT (x,y)]	{	[: OFF]
[LOC[ATION] (x,y)]		[: ON]
		[: FIXED]
		[: HORIZONTAL]
		[: VERTICAL]
		[: SLOPE[D] (angle)]

- control or obtain the position of the locator marker

KEYBOARD

or

KEYBOARD (activation char) [, (activation char), ...

or

KEYBOARD CHAR[ACTER]

- enable the keyboard associated with the display using different forms of activation.

INPUT variable [,variable, ...] [-AS (format list)

- read from the keyboard and wait for the line to be inputted

PUSHBUTTON[S] (number, ...) 0

- enable the referenced pushbuttons

VALUATOR [# num] (var)

- enable the valuator device of number 'num'

APPENDIX B -- ASSIGNMENT STATEMENTS

There are two types of data items stored in variables in this language. They are: numerical data stored in REAL or INTEGER variables, and character string data stored in character STRING variables. Character string literals are represented as strings of characters enclosed in quotation marks, while numeric literals are actual numbers. Variables of any type must be explicitly defined and their name may be up to six characters in length. The first character of a name must be an alphabetic. All other characters may be any alphanumeric. Special characters are not allowed in names. An array element is referenced by a variable name and a subscript enclosed in brackets. The subscript or element number may be any integer arithmetic expression or literal with a positive non-zero value.

There are two types of assignment statements, string variable assignments and arithmetic assignments. The distinction is maintained by having a different syntax for each type of assignment. It is possible to convert from string to arithmetic variables and vice versa using an automatic conversion capability built into the string assignment statement. Only arithmetic variables may be used in arithmetic assignment statements but mixed mode arithmetic is allowed.

ARITHMETIC ASSIGNMENTS

This graphics language is not a general calculation language and a sophisticated calculation capability is not provided. The arithmetic assignment statement described here allows for the evaluation and assignment of any arithmetic expression, but it does not allow general in-line function calls. Intrinsic functions such as SIN(x) are allowed, but the invocation of user defined external functions in this manner requires the definition of these functions as external programs. If greater arithmetic capabilities or more sophisticated data types are required, other outside languages should be used. Calls should be made to subprograms to perform tasks in a language particularly suited to the problem.

An arithmetic assignment is composed of an arithmetic expression and a variable into which the value of the expression is placed. Arithmetic expressions are composed of arithmetic literals and variables and use the five operators +, -, *, / and ^ to represent the operations of addition, subtraction, multiplication, division and exponentiation. The minus sign - may also be used as a unary minus to indicate a negative number.

INTEGER literal numbers consist of a string of decimal numeric characters prefixed by a unary minus in the case of negative numbers. The syntax of the language places no

restriction on the size of the number, but a maximum will be defined by a particular implementation. REAL literal numbers consist of a string of decimal numeric characters, with a decimal point, and optionally followed by a decimal exponent. If a decimal exponent is present it is indicated by the letter 'E' immediately followed by the constant. The decimal point may be omitted if an exponent is specified. The exponent is an optionally signed INTEGER number indicating the appropriate power of ten. In both INTEGER and REAL literals leading zeros are ignored.

The following arithmetic literals are examples:

INTEGER	97	-40	24793	0.
REAL	42.9	-347.	0.	014.002
	243.7E-42	4E2	5.E+4	0.0EO

An arithmetic expression is a collection of numeric literals and variables connected by arithmetic operators in such a manner as to yield a numeric value. A valid expression may be as small as a single arithmetic literal or variable, or it may contain many operators.

The operators $+$, $-$, $*$, $/$ and $^$ are binary operators represented in infix notation. An arithmetic expression is evaluated in accordance with the precedence rules for operators given below and in a left to right manner. Brackets may be used to modify the order of evaluation.

The precedence order is:

- 1) $^$ exponentiation
- 2) unary minus
- 3) $*$ multiplication, $/$ division
- 4) $+$ addition, $-$ subtraction

For the expression:

```

-J + B * 10 / A + A ^ 3
:  :  :      :  :...:
:  :  :      :  :    |
2   :.....:
  \   3
:.....:

```

4

- 1) The variable A is raised to the power 3.
- 2) The value of J is complemented.
- 3) B is multiplied by 10 and then divided by A.
- 4) The remainder of the expression is evaluated left to right.

INTEGER and REAL variables may be used within the same expression but the resulting numeric value of such an expression is always REAL. An expression is REAL if it contains any REAL literals or variables, and all INTEGER variables or literals are converted to REAL equivalents when they are used in REAL arithmetic expressions. The conversion is done when an operator has operands of two different modes, so a part of an expression may be evaluated in INTEGER mode and the remainder in REAL mode.

In the expression,

$(2+6)/14.5$

the integers 2 and 6 would be added to give 8. This would be converted to 8.0 and then divided by 14.5 to give .551724.

An arithmetic assignment statement has the form:

LET variable = arithmetic expression

This statement causes the evaluation of the arithmetic expression and the assignment of its value to the variable. Mode conversion is done across the '=' sign, so that the value takes on the mode of the variable it is assigned to. REAL values are truncated to integer values by dropping the fractional part. INTEGER variables are converted to their equivalent REAL form.

Another form of the arithmetic assignment statement allows for the multiple assignment of one evaluated value to more than one variable.

LET variable, variable [, variable, ...] = arithmetic expression

Arrays of variables are allowed and individual items of the arrays may be referenced by a subscript enclosed in brackets. An array element subscript may be any valid INTEGER arithmetic

expression. A range of elements within an array may be specified by specifying the subscript in the form:

arithmetic expression : arithmetic expression

Reference to a whole array is made by using the name of the array without a subscript.

Arrays in which a range of elements have been referenced may not appear in arithmetic expressions. On the left side of an arithmetic assignment statement a multiple referenced array makes the assignment a multiple assignment.

LET ARRAY(1:5+7) = 0

The above statement assigns zero to elements one to twelve of array 'ARRAY'.

The following statements are all valid arithmetic assignment statements.

LET I = 5

LET J = B

LET I,J,K,LL(N) = 23 * (B/C)

LET AR(1:5) = 2

LET BB(7:2) = 35.7E+23

STRING ASSIGNMENT STATEMENTS

The string variable assignment command has the ability to add characters to a character string variable, and the ability to concatenate and break up such textual strings. A textual string variable may be a single variable of length n characters or it may be an array of variables, each n characters long. Run time checks discard any overflow above the n character limit, and a character string variable is always blank filled from the right if it is not full.

An example definition of a single string variable would look like:

STRING (nc) variable name

-where 'nc' is the maximum number of characters permitted and the variable name is any valid six character variable name.

An array may be specified as:

STRING (nc) array name (nl)

-where 'nl' is the number of strings or lines of length 'nc' in the array.

An array may be referenced by element as in :

array name (element number)

or by range of elements as in :

array name (element number : element number)

or by the array name alone referencing all elements.

The string manipulation commands are outlined below, along with specific examples. For the purpose of these examples, A is assumed to be a character string variable. The general character string assignment statement would have the form:

PUT variable INTO variable [, var, ...] [AS (format list)]

The assignment of the character string literal 'xyz' to the character string variable A would have the form:

PUT 'xyz' INTO A

Either string variables or string literals or a concatenation may be assigned to a string variable. The concatenation operator is '&'. An example of appending to a string variable is:

PUT A & 'xyz' INTO A

Automatic conversion from and into numerical variables is performed by an string assignment statement. If a REAL or INTEGER variable is in the list to be assigned to a string variable, it is converted to an equivalent character string representation and then assigned. If a character string is assigned to a numeric variable a conversion is attempted from the character string to a numeric format. This may fail because the character string may not represent a number. In this case, the statement will be aborted and an error flag, which can be tested by a run-time library function, will be set.

The conversion from numeric variables to string variables is performed by the assignment

PUT I INTO A

where 'A' is a string variable and 'I' an integer variable. The above statement assigns the character representation of 'I' to 'A'. Conversely:

PUT A INTO I

assigns the numeric value represented by 'A' to 'I'. It is permissible to combine numeric variables and character string variables in a string assignment under the concatenation operator as below:

PUT 'GO ' & I & ' TIMES' INTO A

where 'I' is an integer. As an example, if the variable I contained the number 241 then the above string assignment would place the characters:

GO 241 TIMES

into the character string A.

The 'AS' phrase allows the association of a format clause with a string to numeric or numeric to string assignment. The format list allows for the specification of the manner in which the assignment will be performed. It is essentially a restriction or clarification of the otherwise automatic conversion facility. There are four basic types of format list entities: I, R, S and X, representing integer, real and string variables and a space-filler. If a specific form of a number is desired it can be represented by laying out the manner in which it will be formed. For example:

//// represents a 4 digit integer or a 3 digit integer with sign

///./// represents a real number with 3 digits, a decimal point and 3 decimal places

///.//E/// represents a real number in scientific notation with 2 digits, a decimal point and a 3 digit or 2 digit signed integer exponent

I represents a free format integer number

R represents a free format real number

S represents a free format string variable

X represents a single space or a filler

When a format list is used to form a string variable the above free format variable representations cause the variable to occupy as many spaces as required. The defined maximum width of the target string variable will truncate trailing characters above its width maximum. A real number is represented in scientific notation if it requires more than 8 character positions to represent it in fixed point notation. When a format list is used to decipher a string variable it considers a number as being a string of numeric characters, terminated by either a space, comma or an end of line indicator.

A formatted assignment would look as follows:

PUT 'XYZ' & S & 'BDEF' INTO A AS (///./)

where 'S' is a real variable, and 'A' is a string variable. The format '///.' was used to convert the real number 'S' to character form and then the character string was concatenated and put into the string variable 'A'. Multiple format entities in a format list are separated by commas. The following example is similar to the above assignment except that two format entities are used for the conversion of two variables, one integer and one real.

```
PUT 'TEST' & I & ' ' & S INTO A AS (///// , ///.E/// )
```

If a conversion is being done into a numeric variable, the basic format entities may be used to describe a free format conversion of a particular part of a string. A number before a basic format entity indicates the number of occurrences of that basic entity.

```
PUT A INTO I AS (10X , I)
```

Put the string variable 'A' into the integer variable 'I' using the format list for conversion. The format list specifies to skip the first 10 characters, then to do a free format integer conversion on the remaining number.

A character string assignment with multiple variables in the INTO phrase assigns a copy of the formulated character string to each variable. The following example assigns a blank ' ' to each of the variables and the referenced array elements.

```
PUT ' ' INTO A,B,ARRAY(1:5)
```

The above statement has the effect of clearing the specified variables because the variables will be automatically blank filled after the entered character blank.

Certain characters are available which are to be used in character string variables or to be displayed on the screen, but which cannot be entered as literal characters because they interfere with delimiters.

These characters may be described and entered as illustrated below. There is a departure here from the normal meta-language notation of using square brackets to indicate optional phrases in the language, because here the square bracket is used as a language level delimiter.

Carriage Return	[CR]
Line Feed	[LF]
ESC or Altmode	[ALT]
Right Square Bracket	[RSB]
Left Square Bracket	[LSB]

The control character (control -) [CTRL -]
where - is any valid character
A to Z.

APPENDIX C — EXECUTION FLOW CONTROL

The definitional commands (Sect 4.2) delimit an 'IMAGE' program into blocks of code which execute under the control of the interaction control mechanism (Sect. 4.3.1). Commands to exercise simple control within these blocks are described below. The structured programming constructs of Dijkstra [38], Knuth [39], et al have been incorporated in order to make the writing of programs in 'IMAGE' error resistant. A structured program is a program in which the execution flow is easily readable. The programmer is spared from the trap of entangled branch statement by structuring his program into self-contained blocks and sub-blocks of code. A structured programming language is a language which forces a programmer to write structured programs by providing programming constructs which delimit and operate on self-contained blocks of code. Commands such as the FORTRAN GOTO statement are not permitted as they encourage entangled program execution flow.

Interaction control languages of the class of 'IMAGE' have a more difficult time of providing a structured programming capability than do algorithmic languages because the execution flow pattern is closely linked to the interaction dialogue. The result of any interaction by the user must be defined in such a manner so that the programmer handles all possible program state transitions. Newman's DIAL language [8] is a state diagram based interaction control language, which requires all the possible states and state transitions in a program to be explicitly written. The sheer number of states in moderately large programs makes it difficult to correctly define the state transitions. The 'IMAGE' language solves this problem by making the state transitions automatic and invisible to the programmer. The programmer simply defines the ACTION which will occur upon an interaction interrupt on a graphical OBJECT.

The execution flow within the blocks defined by the interaction control definitional statements behaves as any algorithmic language. A GOTO-less control syntax has been developed which follows the style of IFTRAN [20,21,22]. A single terminator for the range of the execution flow control commands has been chosen so as to eliminate the possibility of interlaced control structures. The following flow charts indicate the usage of these commands.

EXECUTION FLOW CONTROL CONSTRUCTS

A SIMPLE CONDITIONAL

IF (condition)

statement S

FIN

or a conditional for a single statement

IF (condition) statement

A TWO WAY CONDITIONAL

IF (condition)

statement S1

ELSE

statement S2

FIN

A MULTI TEST CONDITIONAL

```

IF ( condition 1 )
    statement S1
ORIF ( condition 2 )
    statement S2
ORIF ( condition 3 )
    statement S3
ELSE
    statement S4
FIN
  
```

A MULTI WAY CONDITIONAL

```

CASE OF ( variable )
CASE number1
    statement A
CASE number2
    statement B
CASE number3
    statement C
CASE number4
    statement D
FIN
  
```

A SIMPLE LOOP

WHILE (condition)
statement S

FIN

A SIMPLE LOOP WITH AN
ESCAPE (OR ABORT) TEST

WHILE (condition 1)

statement A

ESCAPE (condition 2)

statement B

FIN

A LOOP WITH BUILT IN COUNTERS

REPEAT AI var=start,max[,incr]

statement S

FIN

where:

var is a counter.

start is the starting
count.

max is the maximum.

incr is the increment.

AN INFINITE LOOP

REPEAT

statement S

FIN

CONDITIONALS

Conditional expressions are used in execution flow control statements, such as IF (condition) and WHILE (condition). A conditional expression may be any logical expression with a true or false result. Since there are no logical variables in this language, a logical expression must be a mathematical expression, combined with a logical test operator and a comparison value.

The logical test operators are:

=	or	.EQ.
><	or	.NEQ. or .NE.
<	or	.LT.
>	or	.GT.
>=	or =>	or .GE.
<=	or =<	or .LE.

This allows for logical expressions of the form: arithmetic expression followed by a logical test operator, followed by an arithmetic variable or literal, for example:

A = 2	C + 2 = 4
B > I	E .NEQ. 2

It is necessary to provide the capability to combine logical expressions using the Boolean operators AND, OR, and NOT in a control-oriented language such as this in order to allow complicated conditions to be tested. A combined logical expression has the form of a logical expression followed by an .AND. or an .OR. operator, followed by another logical expression; or, a combined logical expression may be composed of other combined logical expressions. Brackets may be used to determine the order of precedence in applying the logical operators. The default order of precedence is left to right. The unary operator .NOT. is used only in a prefix manner to negate a single logical expression.

The following examples indicate combined logical expressions.

```
A = 2 .OR. B > 5
( A + 2 > 4 .AND. B = 3 ) .OR. C <= VAL
.NOT. ( A = 1 .OR. B=5 )
```

Because combined logical expressions can become very long and over-complicated, certain simplifications are necessary. Two or more comparisons against a single variable or literal can be

written using one occurrence of the variable or literal and one occurrence of the comparison operator. The .AND. or .OR. operator can be used to associate strings of arithmetic expressions. A multiple expression is a logical expression in which several arithmetic expressions are associated using the .AND. or .OR. operator and commas to form a logical expression. The following examples will illustrate this form.

$$A, B, \text{AND}, C = 4$$

Which is equivalent to

$$A = 4 \text{ .AND. } B = 4 \text{ .AND. } C = 4$$

and

$$A, C, D, \text{OR}, Q < 7$$

Which is equivalent to

$$(A < 7) \text{ .OR. } (C < 7) \text{ .OR. } (D < 7) \text{ .OR. } (Q < 7)$$

APPENDIX D -- GRAPHICS SUBROUTINE LIBRARY

The 'IMAGE' language is designed primarily to solve the problem of graphical interaction and to provide a high-level picture description capability. An 'IMAGE' program must make use of an external routine written in another language in order to perform complex calculations or to access specialized data types. Within such an external program it is sometimes desirable to make use of a limited graphic capability; for example, a program which stores a picture in a specified data base may draw a picture by making subroutine calls. A set of standard subroutines are available in an 'IMAGE' system library to make a graphics capability available to an external program. These programs, along with their functions, are listed below. This set does not provide complete control over the graphics environment as interaction control facilities are not provided, and as with all graphic subroutine packages it is somewhat awkward to use. The purpose of this library is to provide the application programmer a convenience when he is programming in other than 'IMAGE'.

Routines which are capable of generating a picture:

SET (X,Y) --- Set the drawing beam position to the absolute position X,Y.

LINE (DX,DY) --- Draw a line with the relative displacement DX,DY.

LINES (N,ARRAY) --- Draw concatenated lines with the displacements DX1,DY1 / DX2,DY2 / ... stored in 'ARRAY', where 'N' is the number of X,Y co-ordinate pairs.

LINEO (X,Y) --- Draw a line to the specified position.

LNTHRU (N,ARRAY) --- Draw concatenated lines through the referenced points in 'ARRAY', where 'N' is the number of X,Y co-ordinate pairs.

POINT --- Mark the current beam position with a point.

ARC (M,ARRAY) --- Draw an arc of mode 'M' where the parameters are stored in array 'ARRAY'. The arc is an ellipse fitted through either three x,y points or two points and a slope as indicated by 'M'.

CIRCLE (RADIUS) --- Draw a circle of a specified radius about the current beam position.

SYMBOLS (N) --- Cause symbol number 'N' to be drawn from the symbol library.

TEXT (N,ARRAY) --- Display the string of textual material. 'N' characters are stored in array 'ARRAY' in the maximally packed manner. This is somewhat machine dependent as some machines will pack characters in a different manner than others.

Routines which provide the capability of modifying a picture. These modifiers are stacked and are interpreted in the inverse order of their call. They apply to all drawing commands issued up to an end modifier command. The end modifier command unstacks the modifiers.

ROT (ANGLE) --- Rotate the specified picture elements by the angle specified in degrees. Positive rotation is in the counterclockwise direction.

SCALE (FACTOR) --- Scale the referenced picture by the factor 'FACTOR', where 'FACTOR' is a real number >0.

TRANS (DX,DY) --- Translate the referenced picture element by the relative displacement DX,DY.

REFL (PARM) --- Reflect the referenced picture elements about the X or Y axis or about $X = A*Y$, where $A = \tan(\text{PARM})$. 'PARM' may lie in range $(-90 < \text{PARM} \leq 90)$ and X axis reflections occur if $\text{PARM} = 0$; Y if $\text{PARM} = 90$.

WINDOW (XMIN,XMAX,YMIN,YMAX) --- Specify a region for the display of graphical entities, with clipping being done for all graphical items not fully in the region specified. No modifier may be placed on the stack after the 'WINDOW' command except for the 'WITHIN' modifier. This restriction prevents complex-edged clipping and greatly simplifies the run-time system.

WITHIN (XMIN,XMAX,YMIN,YMAX) --- Specifies a region of a page onto which the referenced graphical entities are to be mapped, with the graphical entities being specified in terms of conceptual page co-ordinates.

ENDMOD (N) --- Delimit the range of reference of a graphical modifier. Graphical modifiers are stacked in the order that they are specified, and all modifiers on the stack form the transformation matrix. The 'ENDMOD' command removes the last specified modifier from the stack, thereby limiting the range of a modifier. The parameter 'N' indicates the number of modifiers to delimit.

PAGE (XMIN,XMAX,YMIN,YMAX) --- Define the dimensions of the display screen in the co-ordinate system to be used.

Routines which provide drawing status mode setting:

SOLID --- Display modifier to cause solid lines to be drawn. The 'LINE TEXTURE' may be either dot, dashed or solid, with 'SOLID' as a default. The line texture codes are solid: 1, dotted: 2 and dashed: 3.

DASH --- Display modifier to cause dashed lines to be drawn.

DOTTED --- Display modifier to cause dotted lines to be drawn.

INT (FACTOR) --- Graphical modifier to specify the intensity level. The intensity factor is specified in the range 0.0 - 1.0.

COLOUR (FACTOR) --- Graphical modifier to specify the colour of the displayed item, if applicable. The colour factor is specified in the range 0.0 - 1.0.

FLASH (N) --- Display modifier to cause the graphical entities that follow to flash repeatedly. 'N' = 1 to flash and 'N' = 0 to end flashing. A stack of flash commands is formed so that entities which contain internal flash on and flash off sequences may be flashed.

CHRSET (N) --- Specify which character set to use if multiple sets are available. The standard (default) character set is number 1, and others are ordered by the order in which they are loaded in the display terminal.

SYMSET (N) --- Specify which symbol set is to be used. There is no default symbol set, so this routine must be invoked before any symbol drawing is allowed.

SUBPIC (NUMB) --- Indicate that the code following should be delimited so that it can be referenced as a subpicture. Implied in this definition is a reference that it be used for the first time. In other words, the drawing commands that follow are to be drawn as usual, as they are defined. The subpicture definition merely indicates a point at which the drawing commands may be re-used. The parameter 'NUMB' specifies the number by which the subpicture is to be referenced.

ENDSUB --- Indicate the end of a subpicture definition. If no subpicture is currently being defined, this command is ignored.

USESUB (NUMB) --- Invoke the use of the subpicture numbered 'NUMB'. Invocations of nonexistent subpictures are ignored.

INVIS --- Specify the beginning of a block of display instructions pending later display. This enables display code to be inserted in the display file but not to be immediately viewable.

VIS --- Cause the block of display instructions which have been generated since the call 'INVIS' to be displayed. 'VIS' has no effect if no display instructions have been generated in the invisible mode.

TAG --- Delimit the items displayed on the screen. Tags are issued between logical display items in order to delimit them for use by the ERASE command in 'IMAGE' and for use by the IDENTIFIER structure. A tag number is returned upon an identifier interrupt indicating which item was 'hit'. A tag command is not required at the beginning of the display instructions or at the end.

Routines to provide graphical I/O device control:

DISPL (N) --- Routine to turn the display on or off dependent on the parameter 'N'. (ON with N = 1). Turning the display off does not destroy the stored image. Turning the display on will redisplay any stored image.

WINK --- Cause the display device to momentarily turn off and then come back on (to wink) so as to alert the user. The primary purpose of the 'WINK' is to acknowledge identifier interrupts which themselves do not cause a change on the display screen which would serve as feedback to the user that something has happened.

DEVICE (NUMB, COMFLG) --- Control the I/O device known as 'NUMB', by directing it with the appropriate command code. The numbered devices are:

- 1 KEYBOARD
- 2 PUSHBUTTONS
- 3 VALUATOR
- 4 PICKER
- 5 DIGITIZER
- 6 LOCATOR

Devices 4, 5 and 6 may be assigned to the appropriate handlers to perform the functions of:

4 IDENTIFY	default	4 PICKER
5 SKETCH	default	5 DIGITIZER
6 POSITION	default	6 LOCATOR

Illegal assignments are ignored, and if COMFLG = 0 the particular device is turned off. If COMFLG = 7 the device is enabled.

MKSET (N) --- Marker mode setting; specify the degree of mobility for the marker.

For N = :

0 MARKER OFF

3 HORIZONTAL

1 ON

4 VERTICAL

2 FIXED

5 FREE

See 'MKSETS' subroutine for constraining the marker to a sloped line.

MKSETS (ANGLE) --- Marker mode setting; constrain the marker to move along a line of slope 'ANGLE'.

MKPOS (XPOS,YPOS) --- Specify the setting of the marker in terms of page co-ordinates.

SKTRES (FACTOR) --- Determine the resolution for which changes in the sketcher position are considered significant. 'FACTOR' is the number of co-ordinates in the current drawing page.

KEYCHR (N,CHAR) --- Set up the activation characters. The keyboard handler accepts characters typed and stores them in a buffer. If either the default activation character carriage return or line feed is typed, the buffer is sent to the control program as a keyboard interrupt. The activation characters may be changed to include all or the specified characters.

To specify particular characters as activation characters, set 'N' equal to the number of characters and set 'CHAR' up as an array containing these characters in packed form.

To set all characters as activation characters set N = 0 and the variable CHAR = 1.0.

To return to the default of having (CR) and (LF) as the activation characters, N = 0 and CHAR = 0.

GIPOS (XMIN,XMAX,YMIN,YMAX,XPOS,YPOS) --- Generate a request for the return of the co-ordinates of the identifier strike. The co-ordinates are returned from the display handler in terms of a 0:1000 page. This is converted to the screen co-ordinates which were current at the time the item was defined. These co-ordinates are calculated from the page co-ordinates and from the parameters XMIN,XMAX,YMIN,YMAX.

GMPOS (XMIN,XMAX,YMIN,YMAX,XPOS,YPOS) --- Generate a request for the return of the co-ordinates of the marker. XMIN,XMAX, ... are handled similarly to 'GIPOS'.

PLOT --- Copy the image on the screen onto the hardcopy plotter.

APPENDIX E -- GTI COMMUNICATIONS PROTOCOL

The software modules in the 'IMAGE' system can be divided into two groups. The machine independent software written in the 'IMAGE' language along with some of the run-time software describes the function of a particular program, independent of what hardware is used. This software makes all of its I/O references to a virtual graphics terminal defined by the communications protocol across the interface between it and the display terminal dependent software. The following set of instructions known as GTI (Graphical Task Instructions) describes the commands to this virtual terminal.

GTI INSTRUCTIONS

IMAGE CHARACTERIZATION	FORMAT	DESCRIPTION

I. Display Generation Instructions		
LINES	opcode n rel xl rel yl : rel xn rel yn	- draw a sequence of concatenated lines each of x,y displacements xl and yl respectively. Lines are defined on a conceptual page, transformed and then drawn on the screen.
LINES THRU	opcode n abs xl abs x2 : abs xn abs yn	- draw a sequence of concatenated lines through specified points on the conceptual drawing page then move the origin of the conceptual page to either the current beam position or else to a specified point determined by the AT phrase.
LINE TO	opcode abs x abs y	- draw a line from the current set beam position to the point (x,y) on the page.
SPECIFY COORD. (SET)	opcode abs x abs y	- set the drawing beam to the point (x,y) in the current page co-ordinate system.
CHARACTERS	opcode n char 1,2 : char n	- draw a sequence of characters beginning at the current beam position or as modified by the AT phrase. ROT & REFL modifiers have a limited effect on char. strings.

ARC	opcode m parm 1 : parm n	- draw an arc using the given parameters. 'm' specifies the mode, where an arc is defined defined by 3 x,y points; or 2 points and a slope, etc.
POINT	opcode	- draw a point at the current beam position.
SYMBOLS	opcode sym #	- draw a symbol from the library of symbols.

II. Co-ordinate Specification

PAGE	opcode xmin xmax ymin ymax	- defines the user co-ordinate system & specifies values for unit x & y displacements on the screen.
------	--	--

III. Graphical Modifier Instructions

ROTATE	opcode angle	- rotate a graphical item by the stated angle where: (-360<angle<=360)
REFLECT	opcode parm	- reflect graphical items about the line $y=a*x$ where $a=\tan(\text{parm})$ (-90<parm<=90)
SCALE	opcode factor	- scale graphical items by a factor of K (K>0)
TRANSLATE	opcode dx dy	- translate graphical items by an amount dx in the x-direction & dy in the y-direction.
WINDOW	opcode xmin xmax ymin ymax	- specifies the region for the display of graphical items, with the clipping being performed for all graphical entities, not fully in the region specified.
WITHIN	opcode xmin xmax ymin ymax	- specifies a region of a page for which graphical items are to be mapped onto with the graphical items being specified in terms of conceptual page co-ordinates.
END MODIFIER	opcode	- remove the last specified graphical modifier from the stack of current graphical modifiers.

IV. Status Mode Setting Instructions

INTENSITY	opcode level	- specifies the intensity level for graphical items (0 - 1).
COLOUR	opcode value	- specifies the colour of a graphical item.
LINE TEXTURE	opcode type #	- specifies line texture (solid, dash, dot-dash, etc.)
FLASH ON	opcode	- specifies that graphical items to follow are to flash repeatedly when being displayed.
FLASH OFF	opcode	- disable FLASH.
CHARACTER SET SPECIFICATION	opcode set #	- specifies the character set to be used.
SYMBOL SET SPECIFICATION	opcode set #	- specifies the symbol set to be used.

V. Subpicture Definition Instructions

BEGIN SUBPICTURE	opcode subp #	- delimits the beginning of a new subpicture.
END SUBPICTURE	opcode	- delimit the end of a subpicture definition.
USE SUBPICTURE	opcode subp #	- cause an instance of a subpicture to be re-used.

VI. Display File Modifier Instructions

a) delimiting

INVIS	opcode	- specifies the beginning of an entity pending later display.
VIS	opcode	- add INVIS entity to the display.
TAG	opcode	- delimit a picture for identification.

b) control

ERASE	opcode begin tag # end tag #	- delete specified tagged objects in the display file.
CLEAR	opcode	- clears the display file.

ON	opcode	- initiates the display of the display file.
OFF	opcode	- suspends the display of the display file.
WINK	opcode	- cause an ON/OFF/ON sequence for the display to acknowledge interaction.

VII. Interactive Device Control Commands

INPUT DEVICE ON	opcode device #	- enable the specified input where the device refers to one of the six virtual device types.
DEVICE OFF	opcode device #	- disable a specified device
ASSIGN DEVICE	opcode device # funct #	- determine which input functions will be performed by which device
SET MARKER POSITION	opcode x pos y pos	- specifies the position at which a markae will appear on the current page.
MARKER MODE	opcode code	- specify the marker constraints 0 marker on 3 horizontal 1 marker-off 4 vertical 2 fixed 5 free
	opcode angle	- constrained to a tilted line
SKETCH RESOLUTION	opcode factor	- determine the level for which changes in the sketcher position are considered significant.
KEYBOARD ACTIVATIO L CHAR	opcode n char 1,2 char n	- enable specified characters for use as activation characters on entry.
X & Y IDENTIFIER POS. REQ.	opcode	- request the return of the coordinates of the last identifier interrupt.
MARKER POS. REQ	opcode	- request the return of the marker position.

The following GTI codes are returned from the graphics terminal.

IDENTIFIER RETURN TAG	opcode tag #	- return the tag number for the object interactively selected.
IDENTIFIER RETURN POS	opcode x pos y pos	- return the x & y co-ordinates struck with the identifier.
MARKER POS. REQ.	opcode x pos y pos	- return the current x & y marker position.
SKETCH POS. RET.	opcode xpos ypos	- return an x,y point visited by the stylus, from the queue of such points.
CHARACTER STRING RETURN	opcode n char 1,2 : char n	- return the input character string terminated by the the activation character.
PUSHBUTTON RETURN	opcode pb #	- upon a pushbutton interrupt return the pushbutton code.
VALUATOR SETTING RETURN	opcode valu # value	- upon a significant change in valuator setting return the value.

ERROR REPORT 1	opcode error #	- error report from the terminal software to the 'IMAGE' run-time software.
ERROR REPORT 2	opcode error #	- error report to the terminal software from the 'IMAGE' run- time software.

APPENDIX F -- THE 'STAGE2' MACRO PROCESSOR

In this Appendix some details of the structure of Waite's 'Mobile Programming System' [33] are presented. Only the flavour of 'STAGE2' and the structure of its portable implementation are outlined. For a complete definition of 'STAGE2' and a description of its implementation procedure, refer to the published works of Waite [40,41,42,43].

The principle use of a 'macro' is for text substitution. A 'Macro Call' in a source program is expanded into a string of output code. A macro template is defined and matched to the code in the program being translated. Basically, a macro processor is an expansion mechanism. Individual lines of code are recognized and expanded into other lines of code, which in turn may be expanded again of outputted. Macros may call other macros to any depth of recursion. The 'STAGE2' macro processor can be used to write a translator since it accepts characters (represented in any code), allows the manipulation of these characters and then outputs the characters.

The following example 'STAGE2' macro template recognises the simple assignment statement of LET <variable name> = <variable name>. In this example the character '#' is used to represent a parameter and the character '\' to indicate the end of macro template.

```
LET # = #\           Macro template
```

When this template is recognised in the input string the macro body will be output. As an example, the assembler language of the PDP-9/15 computer is used. The assignment will be expanded to a fetch instruction (LAC) and a store instruction (DAC). The character '@' represents a parameter in the macro body and the character '\$' indicates the end of macro body line. The macro template and body would be represented as:

```
LET # = #\           Macro template
LAC @20$             Macro body
DAC @10$
$$                   End of macro
```

The symbols #\@ and \$ delimit the macro template and body and are reserved characters. They are defined at translation time so that they may be easily changed if these special characters must be used in a template body. The following is an example input string to the macro defined above and the resultant output.

```
LET A = B           This command expands to
LAC B               this code
DAC A
```

Figure F1 illustrates the 'STAGE2' macro definitions more completely.

'STAGE2' allows recursive macro calls and logic may be performed in the macros. A powerful set of processor functions (#FI etc.) allow for I/O, device control, etc. Figure F2 demonstrates the usage of 'STAGE2' to perform extended precision division. The 'source' program consists of statements of the form:

DIVIDE 10.BY 7 TO 30 PLACES.

and the 'output code' is actually the evaluation of this statement.

Chapter 5 has described the portability concept of 'MPS' and 'STAGE2' and discussed its basis on abstract machine modelling. Figure F3 presents the structure of the 'FLUB' machine used to achieve this portability, and Figure F4 presents schematically the bootstrapping technique used by 'MPS'. The description of 'STAGE2' written in the assembler language for the 'FLUB' machine and the macro definitions used to translate 'FLUB' programs into programs for the target machine are submitted to the rudimentary macro processor SIMCMP (which is written in FORTRAN). This results in a version of 'STAGE2' written in the assembler language of the target machine, which may be assembled or compiled into absolute machine code.

Fig.F1 STAGE2 macro definitions

This example shows how extended-precision division can be programmed.

	STAGE2 flag line
<pre> \$0 (+-*/) DIVIDE * BY * TO * PLACES ANSWER = '10/'20\$ AUGMENT ANSWER BY .5 DIVD = '10\$ DIVS = '20\$ DIVIDE '30 TIMES\$ PRINT '10/'20 = ANSWERS\$ \$ = '24'26\$ 'F3\$ \$ AUGMENT * BY '11'20'26\$ 'F3\$ \$ DIVIDE * TIMES '10'F7\$ DIVD = (DIVD - ((DIVD/DIVS)*DIVS))*10\$ QUOT = DIVD/DIVS\$ ADD QUOT TO ANSWERS 'F8\$ \$ PRINT = '10 = '21'F1\$ \$ ADD * TO AUGMENT '20 BY '11\$ \$ END 'F0\$ \$ \$ DIVIDE 10 BY 7 TO 30 PLACES DIVIDE 1 BY 3 TO 6 PLACES DIVIDE 1000 BY 3 TO 6 PLACES DIVIDE 2 BY 4 TO 3 PLACES DIVIDE 17 BY 9 TO 30 PLACES DIVIDE 17000 BY 9 TO 10 PLACES DIVIDE 17 BY 9000 TO 10 PLACES END </pre>	<pre> 'source code' template calculate integer part generate decimal point set up dividend and divisor for division routine calculate decimal part print complete answer a utility macro, stores the result of an arithmetic expression in memory concatenation of digits the division algorithm used for the decimal part prints formatted result we need this to make use of the contents of QUOT shuts off STAGE2 'source code' output </pre>

```

10/7 = 1.428571428571428571428571428571
1/3 = 0.333333
1000/3 = 333.333333
2/4 = 0.500
17/9 = 1.888888888888888888888888888888
17000/9 = 1888.8888888888888888888888888888
17/9000 = 0.00188888888888888888888888888888

```

Fig. F2 STAGE2 usage - extended precision division

Fig. E3 The Paedl machine FINE (first language under bootstrap)

Fig. F4 STAGE2 Bootstrapping and Use

APPENDIX G — TWO EXAMPLE PROGRAMS IN A VARIETY OF LANGUAGES

The following two types of programs illustrate two different forms of graphical application programs. One is a graphical presentation problem while the other is a graphical interaction problem. The graphical presentation problem consists of displaying upon command a curve calculated from data also displayed on the screen and previously entered via a keyboard. The primary job is the calculation and presentation of a graph, and it involves a minimum of user interaction. The graphical interaction problem involves a high degree of interaction and the creation and maintenance of a dynamic graphical data structure. It involves the interactive creation and editing of line drawings using both the identifying and positioning functions. The appearance of the display screen during the execution of the programs is shown in figures G1 and G2.

The graphics programming systems presented here are IMAGE, GRAPPLE, ICPL, PDP-15 VT-15 FORTRAN graphics package and PDP-9 assembler language. These programs were used in the comparison of languages in Chapter 2.

Appearance of the screen during execution of SHDEMO

FREQUENCY	AMPLITUDE	PHASE
1	800	-90
2	400	0
3	0	0
4	400	180
5	0	0
6	200	0
7	0	0
8	0	0

PLOT

REFRESH

EXIT

Appearance of the screen during execution of LINE EDITOR

STAY CO-ORD

MOVE CO-ORD

DELETE

A GRAPHICAL PRESENTATION PROBLEM IN IMAGE

```
*****
*
*          HARMONIC SYNTHESIS DEMONSTRATION
*
*****
```

```
* THIS HARMONIC SYNTHESIS DEMONSTRATION PROBLEM ILLUSTRATES ON
* THE DISPLAY SCREEN THE SUPERPOSITION OF HARMONICALLY RELATED
* SINE WAVES OF VARYING AMPLITUDES AND PHASES. THE AMPLITUDE
* AND PHASE ARRAYS ARE DISPLAYED TO PERMIT THE USER TO SELECT
* NEW AMPLITUDE AND PHASE VALUES. THE LIGHT-BUTTON PLOT WHEN
* SELECTED DRAWS THE GRAPH OF THE SUPERPOSITION OF THE SINE
* WAVE SPECIFIED IN THE AMPLITUDE AND PHASE TABLES. TABLE
* ENTRIES MAY BE CHANGED BY SELECTING THEM AND TYPING THE NEW
* VALUE.
```

```
*
EXTERNAL PROGRAM PLOTTER      ** DEFINE THE EXTERNAL PROG. CALLED
PAGE 300:1500                ** DEFINE THE SCREEN COORDINATES
INTEGER XPOS,YPOS,I,THETA    ** DEFINE THE VARIABLES USED
REAL TOTAL,COMP
INTEGER A,P
STRING(5) AMPL(8),PHASE(8) ** DATA ARRAYS STORED AS CHARACTERS
```

```
*
ENTRY
  PUT '800' INTO AMPL(1)      ** SET THE INITIAL VALUES OF THE
  PUT ' -90' INTO PHASE(1)    ** ARRAYS AMPL AND PHASE
  PUT ' 0' INTO AMPL(2:8),PHASE(2:8)
  DISPLAY                     ** DISPLAY ALL OBJECT BLOCKS
  SEEK                         ** ENABLE THE INTERRUPT STRUCTURE
  WAIT                        ** WAIT FOR AN IDENTIFIER INTERRUPT
```

```
*
OBJECT                         ** LIGHT-BUTTON 'PLOT'
  TEXT 'PLOT' AT(1350,650)
ACTION
  REMOVE                       ** UPON A STRIKE ON 'PLOT'
                                ** ERASE THE PREVIOUS 'PLOT'
```

```
*
  REPEAT THETA = 0,360,4      ** LOOP THROUGH EACH X AXIS POINT
  LET TOTAL = 0                ** CLEAR THE SUPERPOSITION TOTAL
  REPEAT I = 1,8              ** LOOP THROUGH EACH HARMONIC
    PUT AMPL(I) INTO A         ** CONVERT AMPL AND PHASE DATA
    PUT PHASE(I) INTO P        ** INTO NUMBERS
    CALL PLOTTER (A,P,THETA,COMP,I) ** CALCULATE THE VALUE
    LET TOTAL = TOTAL + COMP   ** ADD TO THE SUPERPOSITION
  FIN
  LET YPOS = TOTAL + 1200     ** CALCULATE Y POSITION
  IF (THETA = 0)               ** FIRST TIME ?
    LET XPOS = 600            ** INITIAL X POSITION
    SET AT(XPOS,YPOS)         ** SET THE BEAM
  ELSE
    LET XPOS = XPOS + 10      ** INCREMENT X POSITION
    LINE TO (XPOS,YPOS)       ** JOIN EACH POINT WITH A LINE
  FIN
  SEEK                         ** RE-ENABLE THE IDENTIFIER
```

```

OBJECT      ** LIGHT-BUTTON 'REFRESH'
TEXT 'REFRESH' AT(1350,600)
ACTION      ** UPON A STRIKE ON 'REFRESH'
  REMOVE    ** ERASE THE PLOTTED CURVE
  SEEK
*
OBJECT      ** LIGHT-BUTTON 'EXIT'
TEXT 'EXIT' AT(1350,550)
ACTION      ** UPON A STRIKE ON 'EXIT'
  STOP      ** RETURN TO THE MONITOR
*
OBJECT      ** DRAW THE TITLES AND AXIS
LET YPOS = 925
TEXT 'FREQUENCY' AT (600,YPOS) ** TITLES
TEXT 'AMPLITUDE' AT (750,YPOS)
TEXT 'PHASE' AT (900,YPOS)
REPEAT I=1,8 ** TABLE LINE NUMBERS
  LET YPOS = YPOS -50
  TEXT I AT(600,YPOS)
FIN
DO TMARK(25,25,36,0,8,8):AT(600,1200) ** AXIS
*
PROCEDURE TMARK(TX,TY,XP,XM,YP,YM) ** PROCEDURE TO DRAW A TICK
INTEGER TX,TY,XP,XM,YP,YM ** MARKED AXIS
  IF (XP>0) DO ARM(XP,TX) ** DRAW X AXIS ARM
  IF (XM>0) DO ARM(XM,TX):REF Y:AT (0,0) ** -X AXIS
  IF (YP>0) DO ARM(YP,TY):ROT(90):AT (0,0) ** Y AXIS
  IF (YM>0) DO ARM(YM,TY):ROT(90):REF X:AT(0,0) ** -Y
  RETURN
*
PROCEDURE ARM(XA,T) ** DRAW A TICKED AXIS ARM
INTEGER LARM,POS,XA,T
  LET LARM = XA * T ** CALCULATE LENGTH OF AXIS ARM
  LINE LARM,0 ** DRAW IT RELATIVE TO 0,0
  LET POS = 0
  REPEAT I=1,XA ** DRAW THE TICKS
    LET POS = POS + T ** CALCULATE WHERE
    LINE 0,10:AT(POS,0) ** DRAW IT
  FIN
  RETURN
*
* DISPLAY THE TABLES OF AMPLITUDE AND PHASE VALUES
* UPON A STRIKE ON A PARTICULAR TABLE ITEM READ IN A
* NEW VALUE FOR THAT ITEM
*
OBJECT ALIST ** LIST THE TABLE OF AMPLITUDES
LIST AMPL(1:8) STEP(-50) AT(750,875)
ACTION      ** IF A TABLE ITEM IS STRUCK
  PUT ' ' INTO AMPL(ITEM) ** CLEAR THE INDICATED ITEM
  REDISPLAY ALIST ** REDISPLAY THE TABLE
  INPUT AMPL(ITEM) ** READ IN A NEW ITEM
  REDISPLAY ALIST ** REDISPLAY THE NEW VALUE
  SEEK
*
OBJECT PLIST ** LIST THE TABLE OF PHASES
LIST PHASE(1:8) STEP(-50) AT(900,875)

```

```

ACTION          ** IF A TABLE ITEM IS STRUCK
PUT / / INTO PHASE(ITEM) ** CLEAR THE INDICATED ITEM
REDISPLAY PLIST ** REDISPLAY THE TABLE
INPUT PHASE(ITEM) ** READ IN A NEW TABLE ENTRY
REDISPLAY PLIST ** REDISPLAY WITH A NEW VALUE
SEEK
*
END

```

```

CCCCCCCC CCCCCCCCCCCCC CCCCCCCCCCCCCCCCC CCCCCCCCCCCCCCCCCCCCCC
C
C
C  FORTRAN PROGRAM TO CALCULATE THE VALUE OF THE
C  HARMONICALLY RELATED SINE WAVES SPECIFIED IN THE CALLING
C  PROGRAM
C
C      SUBROUTINE PLOTERR(IA,IP,ITHETA,COMP,I)
C
C      PI = 3.14159
C  SCALE IS 25 SCOPE UNITS FOR 100 LOGICAL UNITS IN THE Y AXIS
C      SCALE = 25./100.
C
C  CALCULATE THE AMPLITUDE AT A PARTICULAR PHASE AND POSITION
C  ALONG THE WAVE FOR THE HARMONICALLY RELATED COMPONENT.
C
C      FREQ = ITHETA * I
C      SFREQ = FREQ + IP
C      RFREQ = SFREQ * PI/180.
C      COMP = IA * SCALE * COS(RFREQ)
C
C      RETURN
C      END

```

A GRAPHICAL INTERACTION PROBLEM IN IMAGE

*
* A SIMPLE GRAPHICS LINE EDITOR *
*

* THIS SIMPLE GRAPHICS LINE EDITOR PERMITS A USER TO INPUT,
* DELETE OR MOVE LINES ON THE DISPLAY SCREEN. A LINE IS CREATED
* BY POSITIONING THE MARKER TO A CHOSEN SCREEN LOCATION,
* SELECTING THE LIGHT-BUTTON 'START CO-ORD', REPOSITIONING
* TO THE DESIRED END OF THE LINE AND SELECTING 'END CO-ORD'.
* SIMILAR OPERATIONS ALLOW ONE TO MOVE OR DELETE LINES.
*

PAGE 300:1500

** DEFINE THE SCREEN CO-ORDINATES

*
* THE DATA BASE CONTAINING THE POSITIONAL INFORMATION ABOUT
* THE LINES DRAWN IS SIMPLY 4 ARRAYS CONTAINING THE BEGINNING
* AND END POINTS OF EACH LINE.

INTEGER XST(100),YST(100),XET(100),YET(100) ** DEFINE THE TABLES

INTEGER PNT ** DEFINE THE TABLE POINTER

INTEGER XS,YS,XE,YE ** START AND END CO-ORD OF WORKING LINE

INTEGER NUM ** POINTER TO THE LAST INDICATED LINE

INTEGER FLAG1,FLAG2 ** STATUS FLAGS

*
ENTRY

POSITION USING : PICKER ** USE THE PICKER (EG. LIGHT-PEN) TO

* ** CONTROL THE MARKER

MARKER AT(1000,1000):FREE ** SET UP THE MARKER FREE TO MOVE

LET PNT = 0 ** INDICATE THE DATA BASE IS EMPTY

LET NUM = 0 ** INDICATE NO LINE HAS BEEN SELECTED

LET FLAG1,FLAG2 = 0 ** SET INITIAL LIGHT-BUTTON STATUS

DISPLAY ** DISPLAY ALL OBJECTS

SEEK ** ENABLE THE INTERRUPT STRUCTURE

*
* MENU OF LIGHT BUTTONS AND ASSOCIATED ACTIONS
*

OBJECT

IF (FLAG1 = 0) TEXT 'START CO-ORD' AT(1450,700) ** CONDITIONAL

ACTION ** OBJECT AND ASSOCIATED ACTION

WINK ** ACKNOWLEDGE HIT

MARKER LOC(XS,YS) ** GET THE MARKER CO-ORD.

LET FLAG1 = 1 ** ENABLE 'END CO-ORD' LIGHT-BUTTON

*
OBJECT ** MARK THE STARTING CO-ORD

IF (FLAG1 = 1) TEXT 'O' AT(XS,YS) ** IF CO-ORD. SPECIFIED

*
OBJECT ** CONDITIONAL LIGHT-BUTTON 'END CO-ORD'

IF (FLAG1 = 1) TEXT 'END CO-ORD' AT(1450,1700)

ACTION

LET FLAG1 = 0 ** ENABLE 'START CO-ORD' LIGHT-BUTTON

MARKER LOC(XE,YE) ** GET THE MARKER CO-ORDINATES

IF (PNT < 100) ** STORE THE LINE IN THE DATA STRUCTURE

```

    LET PNT = PNT + 1    ** IF THERE IS ROOM, AND INCREMENT THE
    LET XST(PNT) = XS    ** THE POINTER PNT
    LET YST(PNT) = YS
    LET XET(PNT) = XE
    LET YET(PNT) = YE
FIN
DISPLAY                ** REDISPLAY THE SCREEN WITH THE NEW LINE
*
OBJECT                ** CONDITIONAL LIGHT-BUTTON 'MOVE CO-ORD'
    IF (FLAG2 = 0) TEXT 'MOVE CO-ORD' AT(1450,600)
ACTION
    WINK                ** WINK THE SCREEN TO ACKNOWLEDGE 'HIT'
    MARKER LOC(XS,YS)   ** GET THE MARKER CO-ORD.
    LET FLAG2 = 1       ** INDICATE CO-ORDS OBTAINED
*
OBJECT                ** LIGHT-BUTTON 'MOVE'
    IF (NUM.NEQ.0 .AND. FLAG2=1) TEXT 'MOVE' AT(1450,600)
ACTION
    LET FLAG2,NUM = 0    ** RESET THE FLAGS
    LET XET(NUM) = XS+XET(NUM)-XST(NUM)
    LET YET(NUM) = YS+YET(NUM)-YST(NUM) ** CHANGE THE REFERENCED
    LET XST(NUM) = XS    ** LINE
    LET YST(NUM) = YS
*
OBJECT                ** 'DELETE' LIGHT-BUTTON
    IF (NUM>0) TEXT 'DELETE' AT(1450,500)
ACTION
    LET NUM = 0          ** REFERENCED LINE TAKEN CARE OF
    REPEAT I = NUM,99    ** DELETE THE REFERENCED LINE
        LET XST(I) = XST(I+1) ** SHIFT UP THE ARRAY
        LET YST(I) = YST(I+1)
        LET XET(I) = XET(I+1)
        LET YET(I) = YET(I+1)
    FIN
    LET PNT = PNT - 1    ** INDICATE 1 LESS ELEMENT
    DISPLAY              ** REDISPLAY THE SCREEN WITHOUT THE DEL. LINE
*
OBJECT                ** DISPLAY ALL LINE IN DATA BASE
    IF (PNT.NE. 0)       ** IF THERE IS SOMETHING TO DISPLAY
        REPEAT I = 1,PNT ** THEN LOOP THROUGH ALL LINES
            LINE TO XET(I),YET(I) FROM XST(I),YST(I)
            TAG           ** DELIMIT EACH LINE
        FIN
    FIN
ACTION                ** UPON A 'STRIKE'
    LET NUM = ITEM       ** DETERMINE WHICH LINE WAS HIT
    WINK                 ** ACKNOWLEDGE THE 'HIT'
*
END

```


ICPL
Interactive Control Program Language

programs by:

C.D.O'Brien
H.G.Bown

HARMONIC SYNTHESIS DEMONSTRATION

CONTROL PROGRAM SHDEMO
EXTERNAL PROGRAM: PLOT, MARK, READ
STRING(S) AMPL(S), PHASE(S), NUM
EFFECTS TEXT I
END

SET 02, 51, 13
APPEND TO AMPL(1) // 022
APPEND TO PHASE(1) // -02
APPEND TO AMPL(2-8), PHASE(2-8) // 0
DISPLAY
SEEK

MENU OF LIGHT-BUTTONS AND ASSOCIATED ACTIONS

OBJECT
TEXT 1352, 552 / PLCT
ACTION
DISPLAY
I/O FILE AMPL(1-8), PHASE(1-8)
DO PLOT (522, 1222)
SEEK

OBJECT
TEXT 1352, 552 / REFRESH
ACTION
DISPLAY
SEEK

OBJECT
TEXT 1352, 552 / EXIT
ACTION
STOP

OBJECT
TEXT 622, 925, 25, 25, 35, 0, 9, 8
WHERE THE PARAMETERS OFF IMAG ARE
X, Y, CO, XLOC, YLOC, I, J, K, L, AND THE TICK NUMBERS X, Y, X, Y, Y

OBJECT
TEXT 622, 925 / FREQUENCY
TEXT 752, 925 / AMPLITUDE
TEXT 922, 925 / PHASE
TEXT 622, 925 / 1
TEXT 622, 925 / 2
TEXT 622, 725 / 3
TEXT 622, 725 / 4
TEXT 622, 575 / 5
TEXT 622, 425 / 6
TEXT 622, 275 / 7
TEXT 552, 525 / 8

OBJECT
LIST 752, 975, 50 / AMPL (1-8)
ACTION
CLEAR NUM, AMPL (ITEM)
I/O FILE NUM
DISPLAY
DO READ
APPEND TO AMPL (ITEM) / NUM
DISPLAY
SEEK

OBJECT
LIST 922, 375, 50 / PHASE (1-8)
ACTION
CLEAR NUM, PHASE (ITEM)
I/O FILE NUM
DISPLAY
DO READ
APPEND TO PHASE (ITEM) / NUM
DISPLAY
SEEK

END

SUBROUTINE READ

C ROUTINE TO RETURN TO THE CALLING PROGRAM THE CHARACTER
C STRING-READ IN AS AN ASCII STRING BUFFER
C
C A WRITE IS DONE TO I/O UNIT NUMBER 6, THE GRAPHICS DISPLAY.

C PROMPT THEN READ

121 WRITE (4,101)
121 FORMAT (' ')
122 READ (4,102) SUM
122 FORMAT ('X',10)
123 WRITE (6,103) SUM
123 RETURN

SUBROUTINE IMARK (XLOC,YLOC,TICKX,TICKY,PX,MX,PY,MY)
INTEGER PX,PY,XLOC,YLOC,TICKX,TICKY

C ROUTINE TO DRAW TICK MARKED AXIS ON THE SCREEN AT POSITION
C XLOC,YLOC. THE SPACING OF THE TICKMARKS ARE TICKX, TICKY
C SCREEN UNITS. PX,MY,MY,MY INDICATE HOW MANY TICK MARKS
C ARE TO BE DRAWN IN THE X,X,Y,Y DIRECTIONS.

IF (PY.LE.0) GOTO 2
LARM = PY * TICKY
CALL VECT (XLOC,YLOC,LARM,3)
IX = XLOC
DO 1 I = 1,PX

IX = IX + TICKX
CALL VECT (IX,YLOC,2,12)
CONTINUE

1 IF (MY.LE.0) GOTO 4
LARM = MY * TICKX
CALL VECT (XLOC,YLOC,LARM,3)
IX = YLOC
DO 3 I = 1,MY

IX = IX + TICKY
CALL VECT (IX,YLOC,2,10)
CONTINUE

4 IF (PX.LE.0) GOTO 6
LARM = PX * TICKY
CALL VECT (XLOC,YLOC,LARM,3)
IX = YLOC
DO 5 I = 1,PX

IX = IX + TICKY
CALL VECT (XLOC,YLOC,2,10,0)
CONTINUE

5 IF (PY.LE.0) GOTO 8
LARM = PY * TICKX
CALL VECT (XLOC,YLOC,LARM,3)
IX = YLOC
DO 7 I = 1,PY

IX = IX + TICKY
CALL VECT (XLOC,YLOC,2,10,0)
CONTINUE

7 RETURN
END

7

NEW YORK, N.Y., MAY 1, 1934

CE-11T 1F SLWZ EO 2
 1EW LAGZ 4127 MOVZ CO-3D
 active
 CC-11C00YS WZ
 1F LAGZ = 1
 1F SLWZ

COPIES IN FILE 230
IF SUPPLY OF THEM TEXT + ASS. 5200 + 5000

INITB S02 PAGE

1 APRIL 30, 1974

SUBROUTINE INITB

C INITIALIZE THE DATA BASE ROUT: 05

C COMMON IXS(100), IYS(100), IXE(100), IYE(100), IPMT

C IPMT = 0
C RETURN
C END

RECORD S02 PAGE

1 APRIL 30, 1974

C SUBROUTINE RECORD(XS,YS,XE,YE)

C ROUTINE TO ADD A VECTOR TO THE DATA BASE

C INTEGER XS,YS,XE,YE

C COMMON IXS(100), IYS(100), IXE(100), IYE(100), IPMT

C IF (IPMT.GE.100) RETURN

C IPMT = IPMT + 1

C SAVE COMPONENTS OF NEW VECTOR IN DATA BASE

C IXS(IPMT) = XS

C IYS(IPMT) = YS

C IXE(IPMT) = XE

C IYE(IPMT) = YE

C RETURN

C END

SUBROUTINE CHANGE (XS,YS,INUM)

ROUTINE TO REPLACE A CHANGED VECTOR IN THE DATA BASE

INTEGER XS,YS
COMMON IXS(100),IYS(100),IXE(100),IYE(100),IPNT

IDX = IYE(INUM) - IYS(100)
IDY = IYE(100) - IYS(100)

IXS(INUM) = XS
IYS(INUM) = YS
IXE(INUM) = XS + IDX
IYE(INUM) = YS + IDY

RETURN
END

SUBROUTINE DEL (INUM)

ROUTINE TO DELETE THE 'INUM' TH ITEM FROM THE DATA BASE AND
TO COMPRESS THE DATA BASE

COMMON IXS(100),IYS(100),IXE(100),IYE(100),IPNT

DO I = INUM,99
IXS(I) = IYS(I+1)
IYS(I) = IYS(I+1)
IXE(I) = IYE(I+1)
IYE(I) = IYE(I+1)

CONTINUE
IPNT = IPNT - 1
RETURN
END

PLOT? S22 PAGE 1 APRIL 30, 1974

SUBROUTINE PLOT?

C PLOT THE LINES AS VECTORS

C COMMON IYS(120), IYS(120), IXS(120), IVE(100), IPMT

IF (IPMT .EQ. 0) RETURN

DO 1 I=1, IPMT

IXS=IXS(I)-IXS(I)

IVE=IVE(I)-IVE(I)

CALL VECI (IXS(I), IYS(I), IXY, IDY)

C IDENTIFY AS SEPARATE ITEMS ON THE DISPLAY

CALL TAG

CONTINUE

RETURN

END

GRAPPLE
Graphical Application Programming Language

programs by:
Bell Northern Research [5]

PDP-9 Assembler Language

programs by:

C.D.O'Brien
H.G.Bown

HARMONIC SYNTHESIS DEMONSTRATION

DEFINE ALL EXTERNAL (GLOBAL) VARIABLE AND PROGRAMS USED BY THIS PROGRAM

```
.GLOBL STRIKE DSP, PLOT, READ, INIT, DA, IMARK
.GLOBL CSTR, DPAR, VECT
.GLOBL DS2, DS2, DSA
.GLOBL WSA, WSA, WSA, WSA
.GLOBL R5
```

SPECIFY ALL MACRO DEFINITIONS USED BY THIS PROGRAM

MACRO DEFINITIONS FOR SEEKING A DISPLAY FILE

```
.DEFIN SEEK
JMS DSA
.ENDM
```

MACRO DEFINITION FOR STARTING A DISPLAY FILE

```
.DEFIN START
JMS DS2
.ENDM
```

MACRO DEFINITION FOR CLEARING (COMPLETING) A DISPLAYFILE UNTO THE DISPLAY DEVICE

```
.DEFIN CLEAR
JMS DS2
.ENDM
```

MACRO DEFINITION FOR TAGGING (MAKING IDENTIFIABLE) ITEMS ON THE DISPLAY

```
.DEFIN TAG
JMS WSA
.ENDM
```

MACRO DEFINITION FOR PUTTING UPPER CASETEXT ON THE SCREEN

```
.DEFIN UTEXT, NUPC, CNSTG
CLA
JMS WSA
JMS NUPC
JMS CNSTG
.ENDM
```

MACRO DEFINITION FOR SETTING THE INTENSITY OF THE DISPLAY

```
.DEFIN INTY, N
JMS WSA
```

```
.DSA
.ENDM
```

MACRO DEFINITION FOR DISPLAYING A CHARACTER BUFFER

```
.DEFIN DBUF, N
JMS R5
.DSA
.ENDM
```

MACRO DEFINITION FOR SETTING THE DRAWING POSITION (BEAM POSITION)

```
.DEFIN SEIXY, IX, IY
JMS WSA
LAC IX
LAC IY
.ENDM
```

MACRO DEFINITION FOR DRAWING A VECTOR

```
.DEFIN VECTOR, DELX, DELY
JMS DRAW
JMS +3
.DSA DELX
.DSA DELY
.ENDM
```

START OF PROGRAM EXECUTION

INITIALIZE THE DISPLAY HANDLER

FIRST ENTER THE HANDLAE INTO THE DAT TABLE AND THEN INITIALIZE IT

FOR THE DOS MONITOR SETUP TO GET THE ADDRESS OF THE DAT TABLE

SCOM=102

```
LAC* (.SCOM+23
YAD C3
DAC #DATAP
```

MODEV FOR DSP.

```
LAC DSP. /ADDRESS OF DSP.
DAC DATAP
```

INITIALIZE DS.

```
.INIT 3, 2, 0
```

INITIALIZE FLAGS

SET THE PLOT NO PLOT FLAG INITIALLY TO ZERO

```
DZF #FLAG
```

INITIALIZE THE FORTRAN DATA ARRAYS: AMPLITUDE AND PHASE

JMS* INIT

SET INTENSITY TO NORMAL LEVEL

INTY 2

DISPLAY LIGHTBUTTIONS, THEN PICTURE, THEN SEEK FOR LIGHT PEN STRIKES

JMS DISP

JMP SEEK1

ROUTINE TO DISPLAY THE LIGHT BUTTIONS, THE GRAPH AXIS AND THE GRAPH

ITSELF

DISP

INITIALIZE DISPLAY HANDLER

START

WRITE LIGHT BUTTIONS ON THE SCREEN

```

SEIYX (1132, (1635
UTEXT 11, <(FREQUENCY)>
SEIYX (1356, (1635
UTEXT 11, <(AMPLITUDE)>
SEIYX (1524, (1635
UTEXT 5, <(PHASE)>
SEIYX (1132, (1553
UTEXT 1, <(1)>
SEIYX (1132, (1471
UTEXT 1, <(2)>
SEIYX (1132, (1407
UTEXT 1, <(3)>
SEIYX (1132, (1325
UTEXT 1, <(4)>
SEIYX (1132, (1243
UTEXT 1, <(5)>
SEIYX (1132, (1161
UTEXT 1, <(6)>
SEIYX (1132, (1077
UTEXT 1, <(7)>
SEIYX (1132, (1013
UTEXT 1, <(8)>

```

DRAW GRAPH AXIS

```

JMS*
JMP
JSA
JSA
JSA
JSA
JSA
JSA
JSA
JSA

```

DISPLAY THE PLOT/CONDITIONAL ON FLAG

```

LAC
SNA
JMP
JMS*
JMP
JSA
JSA
JSA
JSA

```

TAG

DISPLAY THE LIGHT BUTTIONS

```

SEIYX (2506, (1212
UTEXT 4, <(PLOT)>
JMS
SEIYX (2506, (1150
UTEXT 7, <(REFRESH)>
TAG
SEIYX (2506, (1046
UTEXT 4, <(EXIT)>
TAG

```

DISPLAY THE AMPLITUDE ARRAY AND THE PHASE ARRAY

```

LAC
DAC
LAC
DAC
JMS
LAC
DAC
JMS
LAC
DAC
JMS

```

CLEAR THE PICTURE ONTO THE DISPLAY

CLEAR

RETURN JMP* DISP

DISPATCHER

SEEK THE LIGHT BUTTIONS LIGHT PEN STRIKES AND PASS AND TRANSFER CONTROL TO THE APPROPRIATE ROUTINES

SEEK1 SEEK

END AC WITH THE STRIKE #, STORED IN GLOBAL 'STRIKE'

LAC* STRIKE

SHOEMO 524 PAGE 9 APRIL 30, 1974

AND
TAD
DAC

6772000
SIXEUF+1
SIXEUF+1

LAC
LLS
AND

CARRY+1
FIFTH CHARACTER

TAD
DAC

0027700
SIXEUF+1
SIXEUF+1

LAC
TAD
DAC

0222200
SIXEUF+1
SIXEUF+1

PUT A BLANK IN THE LAST CHARACTER (OF 6)

JMP*
USC
RETURN

DEFINE DATA AREA FOR ROUTINE USC

SIXEUF

0

CARRY

0

END

SUBROUTINE READ (INUM, CARPAY)

C ROUTINE TO RETURN TO THE CALLING PROGRAM THE INTEGER NUMBER READ
C IN AND TO SAVE THE NUMBER IN AN ARRAY FOR LATER USE IN THE PLOT.
C ROUTINE

C DIMENSION ARRAY(1)
C COMMON IAMPL(8), IPHASE(8)

C WRITE (4,101)
101 FORMAT (' ')
C READ (4,1) IVALUE

C PUT THIS VALUE IN THE AMPLITUDE OR PHASE ARRAY, THE ELEMENT OF WHICH
C IS INDICATED BY INUM. AMPLITUDE IS INDICATED IF 1<= INUM <= 16
C PHASE IS INDICATED IF 17<= INUM <= 32

C IF (INUM.LE.16.AND.INUM.GE.1) IAMPL(INUM) = IVALUE
C IF (INUM.LE.32.AND.INUM.GE.17) IPHASE(INUM-8) = IVALUE

C ENCODE (3,ARRAY,122) IVALUE.
122 FORMAT (15)

C CARPAY = ARRAY(1)

C RETURN
END

SUBROUTINE IMARK (XLOC,YLOC,TICKX,TICKY,PX,MX,PY,MY)
INTEGER PX,PY,XLOC,YLOC,TICKX,TICKY

C ROUTINE TO DRAW TICK MARKED AXIS ON THE SCREEN AT POSITION
C XLOC,YLOC. THE SPACING OF THE TICKMARKS ARE TICKX, TICKY
C SCREEN UNITS. PX,MX,PY,MY INDICATE HOW MANY TICK MARKS
C ARE TO BE DRAWN IN THE +X,-X,+Y,-Y DIRECTIONS.

IF (PX.LE.0) GOTO 2
LARM = PX * TICKX
CALL VECT (XLOC,YLOC,LARM,3)
IX = YLOC
DO 1 I = 1,PX

1 IX = IX + TICKX
CALL VECT (IX,YLOC,2,10)
CONTINUE

2 IF (MY.LE.0) GOTO 4
LARM = -(MY * TICKY)
CALL VECT (XLOC,YLOC,LARM,2)
IX = XLOC
DO 3 I = 1,MY

3 IX = IX - TICKY
CALL VECT (IX,YLOC,0,10)
CONTINUE

4 IF (PY.LE.0) GOTO 6
LARM = PY * TICKY
CALL VECT (XLOC,YLOC,0,LARM)
IY = YLOC
DO 5 I = 1,PY

5 IY = IY + TICKY
CALL VECT (XLOC,IY,-10,0)
CONTINUE

6 IF (MX.LE.0) GOTO 8
LARM = -(MX * TICKX)
CALL VECT (XLOC,YLOC,0,LARM)
IX = XLOC
DO 7 I = 1,MX

7 IY = IY - TICKX
CALL VECT (XLOC,IY,-10,0)
CONTINUE

8 RETURN
END

SUBROUTINE INIT

C ROUTINE TO PUT THE INITIAL VALUE IN THE ARRAYS IAMPL & IPHASE

C COMMON IAMPL(8), IPHASE(8)

C IAMPL(1) = 800

C IPHASE(1) = -90

C DO 1,1 = 2,8

C IAMPL(1) = 0

C IPHASE(1) = 0

C CONTINUE

C RETURN

C END

SUBROUTINE PLOT (IX, IY)

C ROUTINE TO CALCULATE AND PLOT THE COMPLEX WAVEFORM

C FROM THE FUNDAMENTAL SINE WAVE AND ITS HARMONICS

C COMMON IAMPL(8), IPHASE(8)

C PI = 3.14159

C SCALE IS 25 SCOPE UNITS FOR 120 UNITS IN Y AXIS (1 TICK MARK)

C SCALE = 25./120.

C LOOP THROUGH THE 91 POINTS IE. 2.5 POINTS PER Y AXIS TICK.

C THETA INCREMENTS FROM 0 TO 360 IN STEPS OF 4 DEGREES. 4 DEGREES

C CORRESPONDS TO 25 SCOPE UNITS ON THIS SYSTEM

C THETA = -4

C LOOP THROUGH EACH X AXIS POINT.

C DO 1, K = 1, 91

C THETA = THETA + 4

C TOTAL = 0

C DO 2, I = 1, 8

C FREQ = THETA * I

C SFREQ = FREQ * IPHASE (1)

C RFREQ = SFREQ * PI/180.

C COMP = IAMPL(I) * SCALE * COS (RFREQ)

C TOTAL = TOTAL + COMP

C CONTINUE

C DRAW A VECTOR FROM THE LAST POINT TO THIS ONE

C EACH POINT IS SEPARATED BY 10 SCOPE UNITS IN THE X AXIS

C IF THIS IS THE FIRST TIME THROUGH THE LOOP WE MUST SET THE

C ORIGINAL POSITION OF THE LEAF

C ITOTAL = TOTAL

C IF (K.EQ.1) GOTO 3

C DSAVE = TOTAL

C IDISP = IY + ITOTAL

C CALL VSTG (IX, IDISP)

C GOTO 4

C ELSE

C CONTINUE

C IDELTA = TOTAL - DSAVE

C DSAVE = TOTAL

C CALL DRAW (12, IDELTA)

C CONTINUE

C RETURN

C END

FILE EDITOR

A SIMPLE GRAPHICS LINE EDITOR

DEFINE ALL EXTERNAL (GLOBAL) VARIABLES AND EXTERNAL PROGRAMS USED BY THIS PROGRAM

```
.GLOBL ICM,TCV,STRINE,TCI,DSR,
.GLOBL DS2,DS2,DS4,
.GLOBL DS4,DS4,DS4,DS4,
.GLOBL DATA1,DATA2,DATA3,DATA4,DATA5
```

SPECIFY ALL MACRO DEFINITIONS USED BY THIS PROGRAM

MACRO DEFINITION FOR SEEKING LIGHT PEN STIPPS ON A DISPLAY FILE

```
.DEFIN SEEK
JMS DS4
.ENDM
```

MACRO DEFINITION FOR STARTING A DISPLAY FILE

```
.DEFIN STRAT
JMS DS4
.ENDM
```

MACRO DEFINITION FOR CLEARING A DISPLAY FILE ONTO THE DISPLAY DEVICE

```
.DEFIN CLEAR
JMS DS2
.ENDM
```

MACRO DEFINITION FOR TRACKING AN ENTITY, IE MAKING IT DISTINGUISHABLE FROM OTHER DISPLAYED ITEMS. (LIGHT EDITORS)

```
.DEFIN TAG
JMS DS11
.ENDM
```

MACRO DEFINITION TO DISPLAY UPPER CASE TEXT

```
.DEFIN UPENT,NUMC,CRSTG
CLA
JMS DS3
JMS NUMC
JMS CRSTG
.ENDM
```

MACRO DEFINITION FOR DISPLAY INTENSITY SETTING

```
.DEFIN INTY,N
JMS DS4
JMS N
.ENDM
```

.ENDM

MACRO DEFINITION FOR SETTING THE DISPLAY DRAWING STARTING POSITION IE THE BEAM POSITION

```
.DEFIN SEIXY,IX,IY
JMS DS6
LAC IX
LAC IY
.ENDM
```

MACRO DEFINITION FOR INITIALIZING THE DATA STRUCTURE

```
.DEFIN INITS
JMS DATA1
.ENDM
```

MACRO DEFINITION FOR DISPLAYING THE DATA STRUCTURE

```
.DEFIN DISPLY
JMS DATAC
.ENDM
```

MACRO DEFINITION FOR FREEING THE TRACKING CROSS TO MOVE IN ANY DIRECTION FOR THE DISPLAY DEVICE

```
.DEFIN TCFREE
JMS TCI
.ENDM
```

MACRO DEFINITION FOR RECORDING A LINE INTO THE DATA STRUCTURE

```
IXS - STARTING X COORDINANT OF THE LINE
IYS - STARTING Y COORDINANT
IXE - ENDING X COORDINANT OF THE LINE
IYE - ENDING Y COORDINANT
.ENDM
```

MACRO DEFINITION FOR CHANGING A LINE IN THE DATA STRUCTURE

```
IXC - CHANGED STARTING X COORDINANT OF LINE
IYC - CHANGED STARTING Y COORDINANT
NUMN - LINE IDENTIFICATION NUMBER (BUFFER INDEX)
```

```
.DEFIN CHANGE,IXC,IYC,NUMN
JMS DATAC
LAC IYC
LAC NUMN
.ENDM
```

MACRO DEFINITION FOR DELETING A LINE FROM THE DATA STRUCTURE

NUMN - LINE IDENTIFICATION NUMBER (BUFFER INDEX)

```

DEFIN DELSTE,NUMN
JMS* DATAD
LAC NUMN
SECM

```

```

START OF PROGRAM : ENTRY POINT OF THE EXECUTABLE CODE

```

```

INITIALIZE THE DISPLAY HANDLER. FIRST ENTER THE HANDLER INTO THE DAT
SLOT TABLES AND THEN INITIALIZE IT.

```

```

FOR THE DCS MONITOR SET UP TO GET THE ADDRESS OF THE DAT TABLE

```

```

SCCM=122

```

```

LAC* (SCCM+23)
TAD C3
DAD* DATAP

```

```

MODEV FOR DSP.

```

```

LAC DSP. /ADDRESS OF DSP.
DAD* DATAP

```

```

INITIALIZE DISPLAY HANDLER

```

```

INIT 3,0,2

```

```

INITIALIZE FLAGS

```

```

SET FLAG1 TO 2 TO INDICATE THAT THE START GO-PRO LIGHT BUTTON
IS TO BE DISPLAYED

```

```

DZM FLAG1

```

```

SET FLAG2 TO 2 TO INDICATE THAT "MOVE GO-PRO" LIGHT BUTTON IS TO BE
DISPLAYED

```

```

DZM FLAG2

```

```

SET FLAG3 TO 2 TO INDICATE THAT THE LIGHT BUTTON "DELETE" IS NOT
TO BE DISPLAYED.

```

```

DZM FLAG3

```

```

SET THE ENTITY NUMBER "NUM" TO ZERO, INDICATING NO INITIAL LIGHT-
PEN STRIPES

```

```

DZM NUM

```

```

SET THE PARKER CIRCLE FLAG TO 0 INDICATING THAT IT IS NOT TO BE
DISPLAYED

```

```

DZM FLAGM

```

```

SET THE LPOFFS COUNTER TO A LARGE +VE NUMBER

```

```

LAC (222077)
DAD* ALPOFF

```

```

SET T1 TO 0 TO INDICATE NO STRUCK RECORD YET

```

```

DZM T1

```

```

TURN THE TRACKING CROSS ON AT LOCATION 1220,1220
IN THE FREE MODE

```

```

LAC (1220)
DAD* TCY
DAD* TCY
TCFREE

```

```

SET THE INTENSITY LEVEL TO NORMAL (2)

```

```

INTY 2

```

```

INITIALIZE THE DATA STRUCTURE

```

```

INIT

```

```

DISPLAY THE LIGHT BUTTONS AND THE PICTURE

```

```

JMS DISP

```

```

SEEK FOR A LIGHT PEN STRIKE.

```

```

DISPATCHER

```

```

SEEK ON LIGHT BUTTONS AND TRANSFER CONTROL TO THE APPROPRIATE PROGRAM

```

```

SEEK1 SEEK

```

```

LOAD ACC WITH THE STRIKE NUMBER, STORE IT IN GLOBAL STRIKE

```

```

LAC* STRIKE

```

```

2 IDENTICAL STRIPES OFF THE SAME OBJECT ARE REQUIRED TO BE RECOGNISED
AS A VALID STRIKE.

```

```

T1 IS SET TO 2 DURING INITIALIZATION

```

```

SAD T1
JMP STI
DAD* T1
JMP SEEK1

```


LAC.	TCX
DAC	TC5R
LAC.	TCY.
DAC	TCOR

SET FLAG2 = 1 (IE DISPLAY LIGHTBULB 'MOVE') AND
DISPLAY NEW PICTURE AND THE OTHER LIGHTBULBS

LAO
CARE
(1)
-PLA
-S

TO BE DISPATCHED TO SEEK OUT A NEW PICTURE AND LIGHT BUTTONS

14-00000

ROUTINE TO HANDLE ACTIONS ASSOCIATED WITH THE 'MOVE' LIGHT BUTTON
IF FLAG2 = 1, THE LINE IN THE PICTURE THAT WAS LAST SEEN (AS
INDICATED BY 'M') IS NOW GIVEN A NEW STARTING COORD.

LAC RUC THIS A VALID STRINE

SEEKING A LOT SEX

Case 1:23-cv-00001-TCB Document 1-1 Filed 07/11/23 Page 1 of 1

SET PAGEZ = 2 (IE DISPLAY 'MOVE CO-ORDS') AND DISPLAY
VE: PIC:VE AND THE OTHER VE: I BUTTONS.

02. . . FLAG 2
545 . . . JISD

RETURN TO DISPATCHER TO SEEK FOR NEW LIGHTEN STRIKES.

! M33S : c14f

THE INFORMATION CONTAINED HEREIN IS UNCLASSIFIED EXCEPT WHERE SHOWN OTHERWISE BY THE MARKING.

NOT A VALID STRIKE

1. A. 2. B. 3. C. 4. D. 5. E. 6. F. 7. G. 8. H. 9. I. 10. J. 11. K. 12. L. 13. M. 14. N. 15. O. 16. P. 17. Q. 18. R. 19. S. 20. T. 21. U. 22. V. 23. W. 24. X. 25. Y. 26. Z. 27. AA. 28. AB. 29. AC. 30. AD. 31. AE. 32. AF. 33. AG. 34. AH. 35. AI. 36. AJ. 37. AK. 38. AL. 39. AM. 40. AN. 41. AO. 42. AP. 43. AQ. 44. AR. 45. AS. 46. AT. 47. AU. 48. AV. 49. AW. 50. AX. 51. AY. 52. AZ. 53. BA. 54. BB. 55. BC. 56. BD. 57. BE. 58. BF. 59. BG. 60. BH. 61. BI. 62. BJ. 63. BK. 64. BL. 65. BM. 66. BN. 67. BO. 68. BP. 69. BQ. 70. BR. 71. BS. 72. BT. 73. BU. 74. BV. 75. BW. 76. BX. 77. BY. 78. BZ. 79. CA. 80. CB. 81. CC. 82. CD. 83. CE. 84. CF. 85. CG. 86. CH. 87. CI. 88. CJ. 89. CK. 90. CL. 91. CM. 92. CN. 93. CO. 94. CP. 95. CQ. 96. CR. 97. CS. 98. CT. 99. CU. 100. CV. 101. CW. 102. CX. 103. CY. 104. CZ. 105. DA. 106. DB. 107. DC. 108. DD. 109. DE. 110. DF. 111. DG. 112. DH. 113. DI. 114. DJ. 115. DK. 116. DL. 117. DM. 118. DN. 119. DO. 120. DP. 121. DQ. 122. DR. 123. DS. 124. DT. 125. DU. 126. DV. 127. DW. 128. DX. 129. DY. 130. DZ. 131. EA. 132. EB. 133. EC. 134. ED. 135. EE. 136. EF. 137. EG. 138. EH. 139. EI. 140. EJ. 141. EK. 142. EL. 143. EM. 144. EN. 145. EO. 146. EP. 147. EQ. 148. ER. 149. ES. 150. ET. 151. EU. 152. EV. 153. EW. 154. EX. 155. EY. 156. EZ. 157. FA. 158. FB. 159. FC. 160. FD. 161. FE. 162. FF. 163. FG. 164. FH. 165. FI. 166. FJ. 167. FK. 168. FL. 169. FM. 170. FN. 171. FO. 172. FP. 173. FQ. 174. FR. 175. FS. 176. FT. 177. FU. 178. FV. 179. FW. 180. FX. 181. FY. 182. FZ. 183. GA. 184. GB. 185. GC. 186. GD. 187. GE. 188. GF. 189. GG. 190. GH. 191. GI. 192. GJ. 193. GK. 194. GL. 195. GM. 196. GN. 197. GO. 198. GP. 199. GQ. 200. GR. 201. GS. 202. GT. 203. GU. 204. GV. 205. GW. 206. GX. 207. GY. 208. GZ. 209. HA. 210. HB. 211. HC. 212. HD. 213. HE. 214. HF. 215. HG. 216. HH. 217. HI. 218. HJ. 219. HK. 220. HL. 221. HM. 222. HN. 223. HO. 224. HP. 225. HQ. 226. HR. 227. HS. 228. HT. 229. HU. 230. HV. 231. HW. 232. HX. 233. HY. 234. HZ. 235. IA. 236. IB. 237. IC. 238. ID. 239. IE. 240. IF. 241. IG. 242. IH. 243. II. 244. IJ. 245. IK. 246. IL. 247. IM. 248. IN. 249. IO. 250. IP. 251. IQ. 252. IR. 253. IS. 254. IT. 255. IU. 256. IV. 257. IW. 258. IX. 259. IY. 260. IZ. 261. JA. 262. JB. 263. JC. 264. JD. 265. JE. 266. JF. 267. JG. 268. JH. 269. JI. 270. JJ. 271. JK. 272. JL. 273. JM. 274. JN. 275. JO. 276. JP. 277. JQ. 278. JR. 279. JS. 280. JT. 281. JU. 282. JV. 283. JW. 284. JX. 285. JY. 286. JZ. 287. KA. 288. KB. 289. KC. 290. KD. 291. KE. 292. KF. 293. KG. 294. KH. 295. KI. 296. KJ. 297. KK. 298. KL. 299. KM. 300. KN. 301. KO. 302. KP. 303. KQ. 304. KR. 305. KS. 306. KT. 307. KU. 308. KV. 309. KW. 310. KX. 311. KY. 312. KZ. 313. LA. 314. LB. 315. LC. 316. LD. 317. LE. 318. LF. 319. LG. 320. LH. 321. LI. 322. LJ. 323. LK. 324. LL. 325. LM. 326. LN. 327. LO. 328. LP. 329. LQ. 330. LR. 331. LS. 332. LT. 333. LU. 334. LV. 335. LW. 336. LX. 337. LY. 338. LZ. 339. MA. 340. MB. 341. MC. 342. MD. 343. ME. 344. MF. 345. MG. 346. MH. 347. MI. 348. MJ. 349. MK. 350. ML. 351. MM. 352. MN. 353. MO. 354. MP. 355. MQ. 356. MR. 357. MS. 358. MT. 359. MU. 360. MV. 361. MW. 362. MX. 363. MY. 364. MZ. 365. NA. 366. NB. 367. NC. 368. ND. 369. NE. 370. NF. 371. NG. 372. NH. 373. NI. 374. NJ. 375. NK. 376. NL. 377. NM. 378. NN. 379. NO. 380. NP. 381. NQ. 382. NR. 383. NS. 384. NT. 385. NU. 386. NV. 387. NW. 388. NX. 389. NY. 390. NZ. 391. OA. 392. OB. 393. OC. 394. OD. 395. OE. 396. OF. 397. OG. 398. OH. 399. OI. 400. OJ. 401. OK. 402. OL. 403. OM. 404. ON. 405. OO. 406. OP. 407. OQ. 408. OR. 409. OS. 410. OT. 411. OU. 412. OV. 413. OW. 414. OX. 415. OY. 416. OZ. 417. PA. 418. PB. 419. PC. 420. PD. 421. PE. 422. PF. 423. PG. 424. PH. 425. PI. 426. PJ. 427. PK. 428. PL. 429. PM. 430. PN. 431. PO. 432. PP. 433. PQ. 434. PR. 435. PS. 436. PT. 437. PU. 438. PV. 439. PW. 440. PX. 441. PY. 442. PZ. 443. QA. 444. QB. 445. QC. 446. QD. 447. QE. 448. QF. 449. QG. 450. QH. 451. QI. 452. QJ. 453. QK. 454. QL. 455. QM. 456. QN. 457. QO. 458. QP. 459. QQ. 460. QR. 461. QS. 462. QT. 463. QU. 464. QV. 465. QW. 466. QX. 467. QY. 468. QZ. 469. RA. 470. RB. 471. RC. 472. RD. 473. RE. 474. RF. 475. RG. 476. RH. 477. RI. 478. RJ. 479. RK. 480. RL. 481. RM. 482. RN. 483. RO. 484. RP. 485. RQ. 486. RR. 487. RS. 488. RT. 489. RU. 490. RV. 491. RW. 492. RX. 493. RY. 494. RZ. 495. SA. 496. SB. 497. SC. 498. SD. 499. SE. 500. SF. 501. SG. 502. SH. 503. SI. 504. SJ. 505. SK. 506. SL. 507. SM. 508. SN. 509. SO. 510. SP. 511. SQ. 512. SR. 513. SS. 514. ST. 515. SU. 516. SV. 517. SW. 518. SX. 519. SY. 520. SZ. 521. TA. 522. TB. 523. TC. 524. TD. 525. TE. 526. TF. 527. TG. 528. TH. 529. TI. 530. TJ. 531. TK. 532. TL. 533. TM. 534. TN. 535. TO. 536. TP. 537. TQ. 538. TR. 539. TS. 540. TT. 541. TU. 542. TV. 543. TW. 544. TX. 545. TY. 546. TZ. 547. UA. 548. UB. 549. UC. 550. UD. 551. UE. 552. UF. 553. UG. 554. UH. 555. UI. 556. UJ. 557. UK. 558. UL. 559. UM. 560. UN. 561. UO. 562. UP. 563. UQ. 564. UR. 565. US. 566. UT. 567. UU. 568. UV. 569. UW. 570. UX. 571. UY. 572. UZ. 573. VA. 574. VB. 575. VC. 576. VD. 577. VE. 578. VF. 579. VG. 580. VH. 581. VI. 582. VJ. 583. VK. 584. VL. 585. VM. 586. VN. 587. VO. 588. VP. 589. VQ. 590. VR

SAC (1) FLAG ONLY LINE IN PICTURE
322. FLAG OFF 'DELAYS'.

100

SEEK FOR NEW LIGHTEN STRIPES

PICTURE

14-55
d516

ROUTINE TO HANDLE ACTION ASSOCIATED WITH 'STRIKES' ON THE PICTURE.
FIRST OBSERVING IF THE PARKER CIRCLE AND THE DELETE LIGHT BUTTON
ARE ON OR OFF. THEN STORE IN 'X'. THE ACTUAL LINE IS SEEN BY LIGHT PEN.

LAC (2, / 2 LIGHT BUTTONS ALWAYS ON.
DAC #COUNT
LAC FLAGM / IS THE MARKER CIRCLE ON?

ISZ	ISZ	LAC	COUNT	/ YES	IS THE 'DELETE' LIGHT BUTTON ON?
32A					

SSA /	COPIES	251	252

FEICH SYRIE, AND SUBTRACT COUNT.

LAC - COUNT

1
COUNT / COUNT

LAC	TIME	STRIKE NO.
TAP	COUNT	
TAP	VAL	

RETURN TO DISPATCHER TO SEEK AGAIN

REDISPLAY PICTURE

14335
dsic

ROUTINE TO DISPLAY THE PICTURE ON THE DISPLAY DEVICE FROM THE DATA BASE.

ROUTINE TO DISPLAY LIGHTS AND PICTURE TO BE CREATED BY THE INTERACTIVE PROCESS

VITALIZE DISPLAY MAJOR

START

WRITE LIGHT BUTTAS ON THE SCREEN

LAC • ELAS1

JUP
 SET
 UITE
 IAG
 MP

G1
 (2552, (1274
 14, < (SIART CO-ORD) >
 G2

```
SET MY 12652, (1274
TEXT 12, '(E: CO-OR2) >
```

100
JAG
SZA
JAG

FJA 62
G5

```

G3 SETXY (2552,1138)
   UTEXT 13, <MOVE CO-CRD>
   TAG
   JMP G4
G4 SETXY (2552,1138)
   UTEXT 4, <MOVE>
   TAG
   LAC FLAG
   SNA FLAG / 'DELETE' FLAG
   JMP G6
G6 SETXY (2552,1754)
   UTEXT 6, <DELETE>
   TAG
   LAC FLAG
   SNA FLAG
   JMP G5
G5 SETXY 1014, 1008
   UTEXT 10, <C>
   TAG

```

// NOW DISPLAY THE PICTURE

```

G5 DISPLAY

```

// CLEAR THE DISPLAY BUFFER OUT TO THE DISPLAY DEVICE

```

CLEAR

```

// RETURN FROM THE DISPLAY ROUTINE

```

JMP DISP

```

```

END

```


ROUTINE FOR DELETING A LINE FROM THE DATA STRUCTURE

DATAU 2

DETERMINE THE LINE NUMBER OF THE LINE TO BE DELETED

```

XCT*   DATAU
TAD     C-1
DAG     NUM
ISZ     DATAU
        /RUMP ARGUMENT POINTER FOR RETURN
    
```

FETCH POINTERS TO THE XSY,SYE,YE CO-ORDS OF THE LINE TO BE DELETED

```

LAC     XYS
TAD     NUM
DAG     PXED
        /POINTED TO X STARTING CO-ORD TO BE DELETED
LAC     XYS
TAD     NUM
DAG     PYSD
        /Y STARTING CO-ORD
LAC     XYS
TAD     NUM
DAG     PXED
        /X ENDING CO-ORD
LAC     XYS
TAD     NUM
DAG     PYED
        /Y ENDING CO-ORD
    
```

FETCH THE LAST LINE ENTERED IN THE DATA STRUCTURE AND USE IT TO
REPLACE THE ONE TO BE DELETED.

```

LAC     PXS
TAD     C-1
DAG     PYS
        /MAKE POINTER, 1 LESS
LAC     PXS
TAD     PYSD
    
```

```

LAC     PYS
TAD     C-1
DAG     PYS
LAC     PYS
TAD     PYSD
    
```

```

LAC     PYS
TAD     C-1
DAG     PYS
LAC     PYS
TAD     PYED
    
```

```

LAC     PYS
TAD     C-1
DAG     PYS
LAC     PYS
TAD     PYED
    
```

```

JMP     DATAU
        /RETURN
END
    
```

PDP-15 VT-15 FORTRAN Graphics Package

programs by:

C.D.O'Brien
H.G.Bown

3 APRIL 30, 1974

IY = 375 + 50

DO 4 I=1,8

IY = IY + 50

CALL SEPT (102,18)

ENCODE (1,NUM,10)

FORMAT (11)

CNUM(I) = NUM(I)

CALL PLOT (3,NUM(I),1)

CONTINUE

CALL TMRP (100,700,25,25,35,0,8,5)

PUT OUT THE ARRAYS OF AMPLITUDE AND PHASE

DELIMITED BY TABS

AS WE WALKED ABOVE WE MUST USE AN ARRAY TO STORE THE CHARACTERS
 THAT WILL BE PUT ON THE SCREEN. THIS ARRAY MUST BE LONGER THAN THE
 ENTIRE LINE. WE WANT TO STORE BECAUSE ALTHOUGH WE WISH TO STORE
 5 CHARACTERS AND WE CAN STORE 5 CHARACTERS TO A WORD, THERE IS AN EXTRA
 CHARACTER INSERTED BY THE PLOT COMMAND, REQUIRING STORAGE EVERY
 SECOND WORD.

IY = 375 + 52

DO 15 I = 1,8

IY = IY + 50

CALL SEPT (102,18)

ENCODE (3,NUM,25) AMPL(I)

FORMAT (15)

IF (ISBLANK(LE,1)) GOTO 15

CALL PLOT (2,NUM,14)

IY = IY + 50

CALL PLOT (3,CNUM(I),5)

CONTINUE

IY = 375 + 52

DO 25 I = 1,8

IY = IY + 50

CALL SEPT (102,18)

ENCODE (1,NUM,25) PHASE(I)

FORMAT (15)

IF (ISBLANK(LE,1)) GOTO 25

CALL PLOT (2,NUM,14)

IY = IY + 50

CALL PLOT (3,CNUM(I),5)

CONTINUE

IY = 375 + 52

DO 25 I = 1,8

IY = IY + 50

CALL SEPT (102,18)

ENCODE (1,NUM,25) PHASE(I)

FORMAT (15)

IF (ISBLANK(LE,1)) GOTO 25

CALL PLOT (2,NUM,14)

IY = IY + 50

CALL PLOT (3,CNUM(I),5)

CONTINUE

IY = 375 + 52

DO 25 I = 1,8

IY = IY + 50

CALL SEPT (102,18)

ENCODE (1,NUM,25) PHASE(I)

FORMAT (15)

IF (ISBLANK(LE,1)) GOTO 25

CALL PLOT (2,NUM,14)

IY = IY + 50

CALL PLOT (3,CNUM(I),5)

CONTINUE

4 APRIL 30, 1974

IF (ISBLANK(LE,1) AND (ISBLANK(LE,5) AMPL (ISBLANK) = NUMB

IF (ISBLANK(LE,3) AND (ISBLANK(LE,16) PHASE (ISBLANK-8) = NUMB

C RESET ISBLANK

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

C

5-05-0 523 PAGE 4

[illegible]

C ACTION ROUTINE FOR THE "PLOT" LIGHT BUTTON

PICT ON

141 FLAG = 1 RETURN TO THE BEGINNING OF THE MAIN FILE

0270 1834

[illegible]

ACTION ROUTINE FOR THE "REFRESH" LIGHT BUTTON.

PLC: OF

RETURN TO THE ESTIMATE OF THE MAIN FILE

1631 2505

[illegible]

ACTING SQUAD FOR 'EX-101' 5-11-2010

2015

[illegible]

ACTION REQUIRED FOR THE AMPLITUDE AND PHASE ARRAY STRIKES

ELABORATE APPARATUS

12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455565758596061626364656667686970717273747576777879808182838485868788899091929394959697989910010110210310410510610710810911011111211311411511611711811912012112212312412512612712812913013113213313413513613713813914014114214314414514614714814915015115215315415515615715815916016116216316416516616716816917017117217317417517617717817918018118218318418518618718818919019119219319419519619719819920020120220320420520620720820921021121221321421521621721821922022122222322422522622722822923023123223323423523623723823924024124224324424524624724824925025125225325425525625725825926026126226326426526626726826927027127227327427527627727827928028128228328428528628728828929029129229329429529629729829930030130230330430530630730830931031131231331431531631731831932032132232332432532632732832933033133233333433533633733833934034134234334434534634734834935035135235335435535635735835936036136236336436536636736836937037137237337437537637737837938038138238338438538638738838939039139239339439539639739839940040140240340440540640740840941041141241341441541641741841942042142242342442542642742842943043143243343443543643743843944044144244344444544644744844945045145245345445545645745845946046146246346446546646746846947047147247347447547647747847948048148248348448548648748848949049149249349449549649749849950050150250350450550650750850951051151251351451551651751851952052152252352452552652752852953053153253353453553653753853954054154254354454554654754854955055155255355455555655755855956056156256356456556656756856957057157257357457557657757857958058158258358458558658758858959059159259359459559659759859960060160260360460560660760860961061161261361461561661761861962062162262362462562662762862963063163263363463563663763863964064164264364464564664764864965065165265365465565665765865966066166266366466566666766866967067167267367467567667767867968068168268368468568668768868969069169269369469569669769869970070170270370470570670770870971071171271371471571671771871972072172272372472572672772872973073173273373473573673773873974074174274374474574674774874975075175275375475575675775875976076176276376476576676776876977077177277377477577677777877978078178278378478578678778878979079179279379479579679779879980080180280380480580680780880981081181281381481581681781881982082182282382482582682782882983083183283383483583683783883984084184284384484584684784884985085185285385485585685785885986086186286386486586686786886987087187287387487587687787887988088188288388488588688788888989089189289389489589689789889990090190290390490590690790890991091191291391491591691791891992092192292392492592692792892993093193293393493593693793893994094194294394494594694794894995095195295395495595695795895996096196296396496596696796896997097197297397497597697797897998098198298398498598698798898999099199299399499599699799899910001001100210031004100510061007100810091010101110121013101410151016101710181019102010211022102310241025102610271028102910301031103210331034103510361037103810391040104110421043104410451046104710481049105010511052105310541055105610571058105910601061106210631064106510661067106810691070107110721073107410751076107710781079108010811082108310841085108610871088108910901091109210931094109510961097109810991100110111021103110411051106110711081109111011111112111311141115111611171118111911201121112211231124112511261127112811291130113111321133113411351136113711381139114011411142114311441145114611471148114911501151115211531154115511561157115811591160116111621163116411651166116711681169117011711172117311741175117611771178117911801181118211831184118511861187118811891190119111921193119411951196119711981199120012011202120312041205120612071208120912101211121212131214121512161217121812191220122112221223122412251226122712281229123012311232123312341235123612371238123912401241124212431244124512461247124812491250125112521253125412551256125712581259126012611262126312641265126612671268126912701271127212731274127512761277127812791280128112821283128412851286128712881289129012911292129312941295129612971298129913001

STATION 10.0, REMOVE PLOT FROM THE SCREEN

100

卷之四

65-1324

[illegible]

FEEDS DISPLAY THE DISPLAY AND REQUEST THE DISPLAY FILE.

101 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535 1536 1

3231 0109

၀၁၇၈၉၀၁၁၂၁၃၁၄၁၅၁၆၁၇၁၈၁၉၂၀၂၁၂၂၂၃၂၄၂၅၂၆၂၇၂၈၂၉၃၀၃၁၃၂၃၃၃၄၃၅၃၆၃၇၃၈၃၉၄၀၄၁၄၂၄၃၄၄၄၅၄၆၄၇၄၈၄၉၅၀၅၁၅၂၅၃၅၄၅၅၅၆၅၇၅၈၅၉၆၀၆၁၆၂၆၃၆၄၆၅၆၆၆၇၆၈၆၉၇၀၇၁၇၂၇၃၇၄၇၅၇၆၇၇၇၈၇၉၈၀၈၁၈၂၈၃၈၄၈၅၈၆၈၇၈၈၈၉၉၀၉၁၉၂၉၃၉၄၉၅၉၆၉၇၉၈၉၉

66

SUBROUTINE IMARK (XLEC,YLOC,TICKX,TICKY,PX,MX,PY,MY)
INTEGER PX,PY,XLOC,YLOC,TICKX,TICKY

C ROUTINE TO DRAW TICK MARKED AXIS ON THE SCREEN AT POSITION
C: XLOC,YLOC. THE SPACING OF THE TICK MARKS ARE TICKX, TICKY
C: SCREEN UNITS. PX, PY, PY2 INDICATE HOW MANY TICK MARKS
C: ARE TO BE DRAWN IN THE X, Y, Y2 DIRECTIONS.

RF (PX-LE.0) GOTO 2

```

LARR = PX * TICAY
CALL SETPT (YLOC,XLOC)
CALL PLOT (I,LARR,Z,I)
IX = YLOC
DO I = 1, PY

```

```

      IX = IX + 1
      CALL SEPT (IX, 100)
      CALL PLOT (I, 2, 1)

```

```
CONTINUE, GO TO 4
F = (X*LE,2)
LARY = -6*(X*PIC,4)
CALL SEPT(X,XLOC,Y,CC)
CALL PLOT(CILARY,2,1)
IX = XLOC
```

```

DO 3, I = 1, NX
  IX = IX - YICMX
  CALL SEPT (IX, YLOC)
  CALL PLOT (I, E, IZ, I)

```

CONFIDENTIAL
F (PYLE-2) GOTO S.
LARY: PY. TICKY
CALL-SCRIPT (XLOC, YLOC)
CALL PLOT (XLOC, YLOC)

IY = YLCC
 20 5 1 = 1.PY
 IY = IY * FICPY
 CALL SETI-OUTLOC(IY)
 CALL PLOT(11-42)

```

CONTINUE
      (MY,LE=2) GOTO 3
      LAB = -(MY-1.0E+05)
      CALL SETPR (NLEC,NLECC)
      CALL PLAT (1.0,LAB,1)
      IF =NLECC

```

```

      GO 7 I = 1, MY
      IY = IY - TICKY
      CALL SETPT (YLOC, IY)
      CALL PLOT (I, IY, S, IY)
      CONTINUE

```

2

10

SUBROUTINE PLOTS (IX, IY)

```

C
C ROUTINE TO CALCULATE AND PLOT THE COMPLEX WAVEFORM
C FROM THE FUNDAMENTAL SINE WAVE AND ITS HARMONICS
C

```

DIMENSION IAMPL(B), IPHASE(B)
COMMON IAMPL, IPHASE
PI = 3.14159

SCALE IS 25-SCOPE UNITS-FOR 100 UNITS IN Y AXIS (1 TICK MARK)

SCALE = 25.7102.
LOOP THROUGH THE 91 POINTS IE. 2.5 POINTS PER AXIS TICK.
THEIR INCREMENTS FROM 0 TO 362 IN STEPS OF 4 DEGREES.
CORRESPONDS TO 25 SCOPE UNITS ON THIS SYSTEM

```

      THETA = -4
      DO I K=1,3
        : LOOP THROUGH EACH X AXIS POINT.

```

```

THETA = THETA + 4
TOTAL = 0
DO 2 I = 1,8
  FREQ = THETA
  LOOP THROUGH ALL HARMONICS

```

```

      SREQ = FREQ + IPHASE(I)
      RREQ = SREQ * PI/180.
      CCOMP = IAYPL(I) * SCALE * COS( RREQ )
      TOTAL = TOTAL + CCOMP

```

CONTINUED
A VECTOR FROM THE LAST POINT IS SEPARATED BY 135 DEGREES FROM THE FIRST. THE THRO

IF (K.NE.1) GO TO 3
DSAVE = TOTL

SUBROUTINE RECORD (XS,YS,YE,YE)

C ROUTINE TO ADD A VECTOR (LINE) TO THE DATA BASE

C INTEGER XS,YS,XE,YE

C COMMON IXS(100),IYS(100),IXE(100),IYE(100),IPNT

C IF (IPNT.GE.100) RETURN

C IPNT = IPNT + 1

C SAVE CO-ORDS OF NEW VECTOR IN DATA BASE

C IYS (IPNT) = YS

C IYS (IPNT) = YS

C IYE (IPNT) = YE

C IYE (IPNT) = YE

C RETURN

C END

-SUBROUTINE CHANGE (XS,YS,INUM)

C ROUTINE TO REPLACE A CHANGED VECTOR IN THE DATA BASE

C INTEGER XS,YS

C COMMON IXS(100),IYS(100),IXE(100),IYE(100),IPNT

C IDX = IXE (INUM) - IXS(INUM)

C IDY = IYE (INUM) - IYS (INUM)

C IXS(INUM) = XS

C IYS(INUM) = YS

C IXE(INUM) = XS + IDX

C IYE(INUM) = YS + IDY

C RETURN

C END


```

C OTHERWISE THE LIGHT BUTTON 'END CO-ORD'
C
C IF (FLAG1.EQ.1) GOTO 1
C   CALL PLOT (2,1,START(1))
C   GOTO 2
C ELSE
C   1 CONTINUE
C   2 CALL PLOT (2,1,END(1))
C   3 ENDP
C   4 CONTINUE
C THE LIGHT BUTTON 'MOVE CO-ORD' IF FLAG2 = 0
C OTHERWISE THE LIGHT BUTTON 'MOVE CO-ORD' IF FLAG2 = 0
C
C CALL SETPT(522,122)
C IF (FLAG2.EQ.1) GOTO 3
C   CALL PLOT (2,1,MOVE(1))
C   GOTO 4
C ELSE
C   3 CONTINUE
C   4 IF (NUM.EQ.3) GOTO 4
C   5 CALL PLOT (2,1,MOVE(1))
C   6 ENDP
C   7 CONTINUE
C THE LIGHT BUTTON DELETE IF NUM = 0
C
C CALL SETPT(522,52)
C IF (NUM.NE.0) CALL PLOT(2,1,DELETE(1))
C
C THE PARKER CIRCLE IF FLAG3 = 1
C IF (FLAG3.NE.1) GOTO 77
C   CALL SETPT (191,191)
C   CALL PLOT (2,1,CIRC(1))
C   77 CONTINUE
C SET THE REFERENCE POSITION FOR THE BEAM FOR DRAWING IN THE
C SUSPECTURE TO THE ORIGINAL POSITION OF THE TRACKING CROSS
C WHICH IS THE CENTRE OF THE SCREEN. THE SUSPECTURE IN PLOT
C DRAWS RELATIVE TO THE BEAM POSITION WHEN CALLED.
C
C CALL SETPT(522,52)
C
C THE PICTURE (LINE DRAWING) THAT HAS BEEN CREATED BY LIGHT PEN INTERA
C
C CALL PLOT
C
C SET THE REFERENCE POSITION FOR THE BEAM TO (0,0) SO THAT WE
C OBTAIN ABSOLUTE SCREEN COORDINATES FROM ROUTINE 'TRACK'
C
C CALL SETPT(2,2)
C
C NOW LOOK FOR LIGHT-PEN STRIKES
C
C LIGHT PEN INSURANCE AGAINST DOUBLE HITS: A 1/2 SEC. DELAY
C   DO 28, IJM=1,128
C   AZ=COS(.5)

```

```

C SEEK LOOP TO TEST FOR LIGHT PEN STRIKES
C
C IF (LTORPB(LPX,LPY,NAMR,PS,IV)) GOTO 6
C   GOTO 5
C IGNORE PUSH BUTTON HITS
C
C IF (IW.EQ.2) GOTO 5
C
C WE HAVE A LIGHT PEN STRIKE, WHICH ONE? (DISPATCHER)
C
C 'START CO-ORD' LIGHT BUTTON
C
C IF (NAMR.EQ.1) GOTO 11
C
C END CO-ORD LIGHTBUTTON
C
C IF (NAMR.EQ.2) GOTO 12
C
C MOVE CO-ORD LIGHT BUTTON
C
C IF (NAMR.EQ.3) GOTO 13
C
C MOVE LIGHT BUTTON
C
C IF (NAMR.EQ.4) GOTO 14
C
C DELETE LIGHT BUTTON
C
C IF (NAMR.EQ.5) GOTO 15
C
C CHECK FOR STRIKES OF THE PICTURE (WE CAN STILL USE THE NAME
C REGISTER FOR THIS BECAUSE THE LIMIT ON THE NUMBER OF LINES IN THE
C PICTURE IS 172 WHICH IS LESS THAN THE 128 MAXIMUM ALLOWED BY THIS
C SYSTEM. PROGRAMMING FOR 128 IDENTIFIABLE OBJECTS ON THIS SYSTEM
C IS VERY AWKWARD AND THIS SIMPLE PROGRAM IS VERY CLOSE TO MAXIMUM
C
C IF (NAMR.GT.5.AND.NAMR.LT.107) GOTO 16
C
C ELSE, INVALID STRIKES, SEEK AGAIN
C
C GOTO 5
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C ACTION ROUTINE FOR STRIKES OFF 'START CO-ORD'
C
C   11 CONTINUE
C
C DISPLAY THE 'TRACKING CROSS'
C
C CALL TRACK (191,191,0,0)
C
C NOW THE TRACKING CROSS IS MOVABLE BY THE USER AND WHEN IN THE DESIRED
C POSITION ANY ONE OF THE SIX PUSH BUTTONS IS DEPRESSED AND IXI AND IYI
C WILL CONTAIN THE NEW TRACKING CROSS POSITIONS
C

```

```

C SAVE IXI, IYI IN TEMPORARIES
C
C IXI = IXI
C IYI = IYI
C
C PUT THE MARKER CIRCLE ON THE SCREEN AT THE POSITION INDICATED
C AND TURN ON THE 'END CO-ORD' LIGHT BUTTON (FLAG1 = 1)
C
C FLAG1 = 1
C GOTO 100
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C ACTION ROUTINE FOR STRIKES FOR 'END CO-ORD'
C
C DISPLAY THE TRACKING CROSS
C
C CALL TRACK (IXI, IYI, 2, 2)
C
C NOW THE TRACKING CROSS IS MOVED AND WHEN IT IS IN THE DESIRED
C POSITION ANY ONE OF THE SIX PUSH BUTTONS IS DEPRESSED AND IXI AND IYI
C WILL CONTAIN THE NEW TRACKING CROSS POSITIONS
C
C SAVE IXI, IYI IN TEMPORARIES
C
C IXI = IXI
C IYI = IYI
C
C NOW THAT WE HAVE THE STARTING AND END CO-ORD OF A LINE RECORD THIS
C LINE IN THE DATA STRUCTURE.
C
C CALL RECORD (IXI, IYI, IXI2, IYI2)
C
C RE-SET THE 'START CO-ORD' LIGHT BUTTON AND RE-DISPLAY
C AND TURN OFF MARKER CIRCLE
C
C FLAG1 = 0
C FLAG3 = 0
C GOTO 100
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C ACTION ROUTINE FOR STRIKES FOR 'MOVE CO-ORD'
C
C DISPLAY THE TRACKING CROSS
C
C CALL TRACK (IXI, IYI, 2, 2)
C
C WAIT FOR PUSH - BUTTONS STRIKE AND STORE NEW TRACKING CROSS POSITION
C IN TEMPORARIES
C
C IXI = IXI
C IYI = IYI
C
C TURN ON THE 'MOVE' LIGHT BUTTON (FLAG2 = 1)
C
C FLAG2 = 1

```

```

C RE - SEEK
C
C GOTO 100
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C ACTION ROUTINE FOR STRIKES FOR 'MOVE'
C
C NOW CALL THE DATA BASE ROUTINE 'CHANGE' TO CHANGE (MOVED) THE LINE
C INDICATED BY THE VALUE OF 'INUM', WHICH WAS PREVIOUSLY DEFINED BY
C A STRIKE ON A LINE IN THE PICTURE.
C
C CALL CHANGE (IXI, IYI, INUM)
C
C TURN THE 'MOVE CO-ORD' LIGHT BUTTON BACK ON
C
C FLAG2 = 0
C
C RE-SET THE STRIKE LINE NUMBER TO 0 AND RE-DISPLAY
C
C INUM = 0
C GOTO 100
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C ACTION ROUTINE FOR STRIKES FOR 'DELETE'
C
C NOW CALL THE DATA BASE ROUTINE 'DELETE' TO REMOVE THE LINE INDICATED
C BY 'INUM' FROM THE DATA BASE
C
C CALL DEL (INUM)
C
C RE-SET THE LINE NUMBER TO 0 AND RE-DISPLAY
C
C INUM = 0
C GOTO 100
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C ACTION ROUTINE FOR STRIKES OFF OF THE PICTURE (LINE DRAWING)
C
C INUM = MAXR-5
C
C RE - DISPLAY
C
C GOTO 100
C
C
C END

```

SUBROUTINE RECORD (XS,YS,YE,YE)

C ROUTINE TO ADD A VECTOR (LINE) TO THE DATA BASE

C INTEGER XS,YS,XE,YE

C COMMON IXS(100),IYS(100),IXE(100),IYE(100),IPNT

C IF (IPNT.GE.100) RETURN

C IPNT = IPNT + 1

C SAVE CO-ORDS OF NEW VECTOR IN DATA BASE

C IYS (IPNT) = YS

C IYS (IPNT) = YS

C IYE (IPNT) = YE

C IYE (IPNT) = YE

C RETURN

C END

-SUBROUTINE CHANGE (XS,YS,INUM)

C ROUTINE TO REPLACE A CHANGED VECTOR IN THE DATA BASE

C INTEGER XS,YS

C COMMON IXS(100),IYS(100),IXE(100),IYE(100),IPNT

C IDX = IXE (INUM) - IXS(INUM)

C IDY = IYE (INUM) - IYS (INUM)

C IXS(INUM) = XS

C IYS(INUM) = YS

C IXE(INUM) = XS + IDX

C IYE(INUM) = YS + IDY

C RETURN

C END

SUBROUTINE INITE

```

C INITIALIZE THE DATA BASE POINTS, SET POINTER TO 1
C
C COMMON IYS(100),IYS(100),IYS(100),IYS(100),IYS(100),IPNT
C
C IPNT = 1
C RETURN
C END

```

SUBROUTINE PLOTP

```

C PLOT THE LINES AS VECTORS
C
C INTEGER PLOTS(500)
C COMMON IYS(100),IYS(100),IYS(100),IYS(100),IYS(100),IPNT
C
C IF (IPNT.EQ.0) RETURN
C
C SCALE = 1; INT = 2; LPOW = 4; AREG = 120 --- THEREFORE IPARF = 135
C
C IPARF = 135
C
C INITIALIZE SUPERSTRUCTURE
C
C PLOTS(1)=0
C
C PLOT THE LINES BY COMPUTING THE DELTA Y AND DELTA X FOR EACH LINE
C
C DO 1 I = 1,IPNT
C   IF (AREG.I) GO TO 12
C   IDY1=IYS(1)-IYS(1+1)
C   IDY2=IYS(2)-IYS(2+1)
C   CALL LINE (IDY1,IDY1,2,PLOTS(1))
C   GO TO 15
C 12 IDY1=IYS(1)-500
C   IDY2=IYS(1)-500
C   GO TO 15
C 15 I1 = 1+6
C   IDY1 = IYS(1) - IYS(1)
C   IDY2 = IYS(1) - IYS(1)
C   CALL PAPER (IPARF,2,4,1,1,PLOTS(1))
C   CALL LINE (IDY1,IDY1,1,PLOTS(1))
C   CONTINUE
C
C LINK IT TO THE MAIN FILE
C
C CALL PLOT (2,1,PLOTS(1))
C RETURN
C END

```

DEL 503 PAGE 1 APRIL 32, 1974

SUBROUTINE DEL (INUM)

C ROUTINE TO DELETE THE 'INUM' IN TERM FROM THE DATA BASE AND
C TO COMPRESS THE DATA

C COMMON IXS(120),IYS(100),IYE(100),IYE(100),IPNT

C IYS(100) = IYS(IPNT)
C IYS(100) = IYS(IPNT)
C IYE(100) = IYE(IPNT)
C IYE(100) = IYE(IPNT)

C IPNT = IPNT - 1
C RETURN
C END

END

1910170

FIN