THE UNIVERSITY OF ALBERTA

A DATA STRUCTURE APPROACH TO INTERACTIVE GRAPHICS SOFTWARE

by

Ronald Curtis Enerson

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1972

ABSTRACT

The thesis documents an investigation aimed at determining a suitable basis for interactive graphics support software. The functional similarities of interactive graphical display programs have been considered. Based on these points of similarity, a design is presented for a stratified set of support software.

As a first step in the development of interactive graphics software, the author believes that a low level data structure system must be provided. The ALAS system is proposed to satisfy this need. The second step is the creation of a graphical interface. The design of the PRIG system is presented as an example of a graphical interface which makes use of the structure handling capabilities of ALAS. A possible extension of the ALAS/PRIG system is explored to illustrate one feasible path for the development of higher levels of support.

The most important step in the evolution of interactive graphics software is seen to be the development of low level software as a foundation upon which an entire family of support may be built. The author believes that a data structure system such as ALAS would provide a basis which is much superior to that provided by a standard assembly language.

ACKNOWLEDGEMENTS

I wish to express my appreciation to Professor J.P. Penny for his seemingly boundless efforts in advising and guiding me throughout the duration of this research. In particular, his patience with my incorrigible writing style required (I am sure) a truly Herculean effort. His clearcut, incisive editorial remarks did much to clear away the verbal fog.

I owe another debt of gratitude to Gordon Deecker, for endless hours of discussion and generous amounts of moral support. My wife, Leslie, has retyped the bulk of this thesis many times, and proofread the many resultant drafts, and I am extremely grateful for her labors in this area.

I also wish to thank Dr. Penny for his financial assistance during the printing of this thesis. Finally, I extend my gratitude to the Department of Computing Science, for without its financial support and facilities, none of this would be possible.

TABLE OF CONTENTS

PAGE

CHAPTER I - INTRODUCTION 1

CHAPTER II - INTERACTIVE GRAPHICAL DISPLAY PROGRAMS AN INTRODUCTION 5

2.1 Introduction 5

2.2 Graphical Communication 5

2.3 Interactive Graphical Display Programs 10

CHAPTER III - INTERACTIVE GRAPHICAL DISPLAY PROGRAMS FURTHER CONSIDERATIONS 14

3.1 Introduction 14

3.2 Concepts 14

3.3 Conclusions 18

CHAPTER IV - ON SOFTWARE SUPPORT FOR INTERACTIVE GRAPHICS 21

4.1 Introduction 21

4.2 Functional Organization of an Interactive Graphical Display Program 21

4.3 Programming Support 28

4.4 Levels of Graphics Support: An Introduction to ALAS and PRIG 32

CHAPTER V - ALAS - A LIST ASSEMBLER SYSTEM 36

5.1 Introduction 36

5.2 Storage Elements 39

5.2.1 Blocks 39

5.2.2 Fields 39

5.2.3 Templates 40

5.2.4 Block Types 41

PAGE

5.3 Storage Allocation 44

5.3.1 Static Allocation 44

5.3.2 Dynamic Allocation 44

5.3.3 Storage Elements and Their Allocatability 45

5.4 Addressing 45

5.4.1 Base Address Generation 50

5.4.2 Modifiers 51

5.4.3 Literals 54

5.4.4 Operand Length 54

5.4.5 Operand Alignment 57

5.5 Registers 60

5.5.1 Fixed Length Registers 60

5.5.2 Variable Length Registers 61

5.5.3 Register Stacks 62

5.6 Operations 62

5.6.1 Condition Code 62

5.6.2 Register and Operand Specification 63

5.6.3 Operation Descriptions 65

5.7 Machine Instructions 66

5.7.1 General Purpose Instructions 66

5.7.2 Arithmetic Instructions 70

5.7.3 Bit and Character Instructions 72

5.7.4 Addressing Instructions 80

5.7.5 Conversion Instructions 82

PAGE

5.7.6 Allocator Instructions 84

5.7.7 Control Instructions 88

5.8 Assembler Instructions 95

5.8.1 Listing Control Instructions 95

5.8.2 Allocator Instructions 97

5.8.3 Communication Instructions 104

5.8.4 Others 107

5.9 The Loader 107

5.10 Conclusions 108

CHAPTER VI - DISCUSSION AND EXAMPLE 109

6.1 Introduction 109

6.2 Address Processing 109

6.3 Storage Management 113

6.4 Variable Length Registers 114

6.5 A Programming Example 117

CHAPTER VII - IMPLEMENTATION 127

7.1 Introduction 127

7.2 The Hybrid Software Implementation 128

7.3 Hardware Implementations 131

CHAPTER VIII - A GRAPHICAL INTERFACE IN ALAS 133

8.1 Introduction 133

8.2 Picture Building 134

8.2.1 Structure Commands 135

8.2.2 Drawing Content Commands 137

PAGE

8.2.3 Status Sensing and Control Content Commands 139

8.2.4 Editing Commands 144

8.3 Display Control 149

8.4 Input Handling 154

8.5 Message Component Generation 158

CHAPTER IX - EXTENSIONS 170

9.1 Introduction 170

9.2 An Approach to Interactive Graphical Display Programming 170

9.3 Conclusion 177

CHAPTER X - CONCLUSIONS 179

BIBLIOGRAPHY 184

APPENDIX A - A FREEHAND INPUT PROGRAM 189

APPENDIX B - TABULATION OF MNEMONICS 198

TABLES

PAGE

Table 4.1 Levels of Graphical Support 35

5.1 Allocatability of Storage Elements 45

5.2 Grammar of Operand Addresses 47

5.3 Solidity of Address Processor Determined Lengths 58

5.4 Addressed Operand Lengths 59

5.5 Example of Operand Length Determination 59

5.6 Condition Code 63

5.7 Register Character Codes 64

5.8 LOAD and STORE Instructions 67

5.9 Initialization and Comparison Instructions 70

5.10 Arithmetic Instructions 71

5.11 Loading and Deleting Instructions 74

5.12 Variable Length Register Test Instructions 78

5.13 Bit Instructions 80

5.14 Addressing Instructions 81

5.15 Conversion Instructions 83

5.16 Data Format Conversions 83

5.17 Allocator Instructions 85

5.18 Grammar of Allocator Instruction Operand Lists 86

5.19 Internal Control Instructions 90

5.20 Input/Output Instructions 93

PAGE

5.21 Grammar of DFA Operands 102

5.22 Grammar of DFR Operands 104

8.1 Structure Commands 136

8.2 Drawing Content Commands 138

8.3 Status Sensing and Control Content Commands 140

8.4 Editing Commands 145

8.5 Parameters Retrieved by Query Commands 149

8.6 Parameters Altered by Set Commands 150

8.7 Display Control Commands 151

8.8 Control Value Interpretation 153

8.9 Action Identifiers and Their Actions 156

8.10 Significance of X-Y Composite 156

8.11 Contents Composite Contents vs. Component Type 157

8.12 Input Handling Commands 159

8.13 Information Accessed by Access Commands 160

A.1 Syntax of Input for Freehand Input Program 190

A.2 Freehand Input Program in PRIG 192

ILLUSTRATIONS

		PAGE
Figure 2.1	Man/Machine Loop	10
3.1	Two Models of a Line	17
4.1	Interactions of Fundamental Components	26
4.2	An Electrical Circuit Diagram and Its Model	27
5.1	A Block	42
5.2	Example of Fields	42
5.3	Block Header of a Vector	42
5.4	Block Associated With a Pattern	43
5.5	PBLOCK	43
5.6	ABLOCK	43
6.1	Accessing Example	111
6.2	A Graph and Its Representations	116
8.1	GRID Keyboard	162

To L.B.E., G.F.P.D., and P.O.E.(dec.),

"Faith, Hope, and Charity",

without whom this bock would

not have been created,

and to whom I am eternally grateful,

I humbly dedicate the proceedings.

r.c.e.

CHAPTER I

INTRODUCTION

The view is fairly well accepted that, despite its initial promise, interactive graphics has been less than spectacularly successful. The major obstacle to the wider use of interactive graphics is the inadequacy of most existing software support for programming any but the simpler applications. In this thesis, the author investigates the functional nature of interactive graphical display programs in order to develop a basis for support systems for such programming. In developing an overall plan for graphics support, the author perceives a need for a low level data structure system.

Interactive graphical display programs have two functional objectives in common:

a) to utilize effectively the graphical capabilities of the terminal, and

b) to effect a man-machine synergism oriented to the program's objectives.

There is obviously a functional similarity between such programs; in particular, their operation appears similar to that of an interpreter, a type of computer language processor which accepts strings and translates and executes them without generating intermediate object code, as is done by a compiler. Both Johnson (1970) and Deecker (1970)

describe 'console languages' which are processed interpretively by their programs. Also, the need for effective input/output software for communicating with the display is common. The use of a hierarchical structure to represent a displayed picture holds much potential for enhancing graphical input/output. This is particularly true where displayed entities represent instances of other quite different entities, that is, where an effort is made to create graphical symbolic communication, such as in Johnson's schematic display of a logic circuit.

In considering support for interactive graphics programming the author projects the following four levels of software:

a) Level 0 - this level is characterized by having no particular graphics support. Most programming languages exemplify this level.

b) Level 1 - this level is characterized by the presence of a display terminal interface capability. An example is GRIDSUB (Huen, 1970), which provides a block structure technique for describing displays.

c) Level 2 - this level is characterized by the presence of an automated input analyzer in addition to Level 1 support.

d) Level 3 - this level is characterized by the presence of Level 2 software coupled to an automatic semantics sequencer. Newman's system (Newman, 1968) is an

example of Level 3 software.

Examining the nature of such support has led to the following conclusions:

a) Support packages could be created which supply all levels of support in an integrated fashion, a technique having the advantages of compatibility of software written at various levels, and

b) A comprehensive data structure facility at level 0 should be used as the basis for construction of other support.

A number of data structure languages was investigated. None were found which satisfied the author's requirement for a wide range of capabilities coupled with a low level of language. The author then designed ALAS (A List Assembler System).

Having designed what he regards as an effective level 0 support vehicle, the author next considered a level 1 support system. The result is PRIG (Package for Remote Interactive Graphics). Being low level, PRIG is designed for a specific environment, the IBM 360/67 - CDC 160A GRID configuration at the University of Alberta. However, it embodies certain characteristics which the author considers generally applicable in the area of level 1 support. One principal characteristic is allowance for the hierarchical

structuring of pictures.

The development of higher levels of support requires the investigation and development cf a systematic approach to interactive graphical display programming. One such approach is to consider an interactive graphical display programming problem as a problem in language design and implementation. This opens many avenues of exploration for the support designer. The framework of support for programs written in this way suggests the creation of an interpreter-compiler as a level 3 support system. In view of some of the programming techniques used for systems at this level (Newman, 1968) and possible fragments of systems at this level, such as analyzers (Cohen and Gotlieb, 1970) and scanners, ALAS has a definite advantage as a level 0 support system.

ALAS represents, in the author's opinion, a significant first step in the development of interactive graphics support software. It is also intended to be usable more generally as a lcw level data structure language. Its facilities for string processing and linked list handling, abetted by versatile dynamic storage allocation capabilities, should provide an effective data structure system.

CHAPTER II

INTERACTIVE GRAPHICAL DISPLAY PROGRAMS

AN INTRODUCTION

2.1 INTRODUCTION

To provide a basis for further discussion, this chapter introduces a number of definitions and concepts. In particular, a brief discussion introduces an overview of interactive man-machine graphical communication, and the nature of an interactive graphical display program is explored.

2.2 GRAPHICAL CCMMUNICATION

As used within this thesis, the term 'graphical communication' implies the imparting, conveying, or exchange of ideas, knowledge, information, through the use of diagrams, linear figures, or symbolic curves. Techniques involving graphical communication are in evidence in many fields of endeavor. Some media for graphical communication are:

a) maps

b) mechanical drawings

c) schematic drawings

d) performance graphs

e) paintings

Although this concept of graphical communication is completely general, the author tends to direct this discussion to the use of graphical communication as a tool for the solution of technical problems.

Techniques of graphical communication are more powerful than techniques of string language communication, in that certain concepts may be more directly and conscisely expressed. In many areas, graphical communication is a more natural mode of communication for a human than string language, a fact borne out by the extensive use of graphical communication.

For example, an electrical schematic diagram is more concise and direct than a written description of the same circuit. The symbolic interconnections of components can be easily recognized, and have a simple, direct association with the physical circuit.

Because of its power, graphical communication has been extensively explored as a facet of the man-machine interface. Many hardware devices, such as plotters, CRT terminals and picture encoders, and many software systems to make use of them have been designed to provide graphical communication between man and computer. Much has been written on graphical communication, and some of the more significant software systems are listed below.

Sutherland's SKETCHPAD (Sutherland, 1963), one of the first interactive systems to use graphical communication, provided for a limited degree of picture input, display and manipulation. The facilities of SKETCHPAD were deemed (at the time) most useful in the following areas:

a) "for storing and updating drawings..."

b) "for gaining scientific or engineering understanding of operations that can be described graphically..."

c) "as a topological input for circuit simulators, etc."

d) "for highly repetetive drawings..."

Mezei's SPARTA (Mezei, 1968), a plotting package, was designed to simplify the use of the computer as a drawing device.

In "Graphical Communication and Control Languages", Roberts (1964) discusses the advancement of interactive graphics beyond the picture manipulation stage. He introduced such concepts as that "... the pictures are really abstractions used as labels for the external entities so that it is possible to create, interconnect, and rearrange the entities with a two-dimensional language rather than the normal one-dimensional text stream". He relates the concepts described to the current research

efforts at M.I.T., such as the development and use of CORAL (W. R. Sutherland, 1966).

Kulsrud (1968) discusses the requirements for a general purpose graphic language and indicates some of the inherent problems. He describes a system for constructing and testing graphics languages, and presents a model language which fulfills a large number of the requirements.

Newman (1968) describes a system which is oriented about the interactive nature of a graphical display program. The system allows the individual to describe a program as a state diagram or network. The description may then be encoded into a "Network Definition Language", which can be processed to form the desired graphical display program.

Thomas (1967) provides yet another system description, based in part on the design philosophy represented in SKETCHPAD. His GRASP system provides the programmer-user with the ability to create and manipulate drawings as well as alphanumeric information.

Van Dam and Evans (1967), in their paper "A Compact Data Structure for Storing, Retrieving and Manipulating Line Drawings", describe PENCIL, (Picture ENCodIng Language), and the data structures created by their system for drawings

represented in PENCIL.

Denil (1966) provides yet another language, DL1, which uses an interactive graphical environment to provide a facility for the design of three-dimensional structures.

Cameron, Ewing and Liveright (1967) describe "DIALOG, a conversational system with a graphical orientation", which is a system for on-line use of an algebraic language.

Corbin and Frank (1967) describe a system which is oriented to on-line graphics (DOCUS). This system is a programming environment, designed to provide for high level language programming (FORTRAN, COBOL) and executive language programming (either General Operating Language or Procedure Implementation Language).

This brief resume is far from exhaustive, but is sufficient to illustrate that there have been a variety of points of view on interactive graphical display programming.

As with many fields of computing, the terminology used is in a state of confusion and definitions of basic concepts have not yet been agreed upon. Therefore, a discussion of facets of interactive graphical man-machine communication software systems is difficult, unless the working definitions to be used are presented. In the next section,

certain terms in common use are discussed.

2.3 INTERACTIVE GRAPHICAL DISPLAY PROGRAMS

As used within this thesis, the term "interactive graphical display program" implies a program which makes use of an interactive graphical man-machine interface to further in some substantial manner the objective of the program. The user of the program is involved in a control loop with the program, as shown in Figure 2.1.

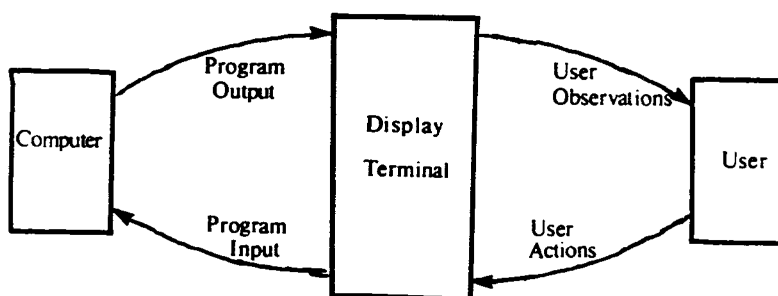


Figure 2.1 Man/machine Loop

One may think of interactive graphical display programs in a manner akin to the approach of Newman (1968), that is, in terms of possible user actions and program reactions or responses to such actions. The interaction allows the user and/or the program to direct the course of the man-machine system.

An example of such a program is CALD (Johnson, 1969), which provides the user with an interactive logic circuit

design facility. The user can "construct" a logic circuit design by drawing a logic circuit schematic on the terminal. He may then specify input values for the circuit, and have the computer determine the circuit's output(s). The user may edit the design and recheck it until the circuit design is satisfactory.

CALD is a computer aided design system, whose operation is as follows. The "construction" of a logic circuit design diagram on the display is reflected within the program by the construction of a data structure "model" of the circuit. Editing of the circuit diagram is reflected as editing of the model. Analysis of the circuit to determine output(s) is performed by the program on the model. As in most other programs for computer aided design, the techniques for encoding the model in CALD were tailored to the application in order to facilitate easy and rapid analysis.

The distinction between interactive graphical display _programs_, such as CALD, and interactive graphical display _support systems_ such as PENCIL, Newman's system, and GRIDSUB (Huen, 1968), should be noted. The former are programs designed for a particular area of application, while the latter are designed to assist the creation of programs of the former sort.

While each interactive graphical display program has individual characteristics peculiar to the problem it is capable of solving, it is the author's contention that a functional division of such programs may be observed, and that this division is common to most programs. This contention is borne out by the design of some of the software systems. For example, Thomas (1967) distinguishes four areas in GRASP:

a) The model, and routines to handle the model. The 'model' in GRASP is "the internal or computer representation of a drawing".

b) The console commands, and command processing routines. With commands the user may request the manipulation of graphical data, or call on the display and system functions.

c) Display routines, which process the model according to user determined parameters to generate a display.

d) System service routines which aid in building or changing the system, and in the control of application processing. Such facilities as saving and restoring part or all of the model, printing out the model (for diagnostic purposes) and initializing and starting application programs are provided.

The idea that interactive graphical programs have functional components in common will be developed further in

Chapter 3 to obtain insight into the implementation of graphical display support systems.

CHAPTER III

INTERACTIVE GRAPHICAL DISPLAY PROGRAMS

FURTHER CONSIDERATIONS

3.1 INTRODUCTION

In this chapter, the author describes concepts which influence the structure of interactive graphical display programs, to bring out what the author sees as the fundamental nature of these programs.

3.2 CONCEPTS

The following are characteristics of most interactive graphical display programs:

a) Interactivity

b) Modelling

c) Graphical Communication

a) Interactivity

An interactive graphical display program is one which alternately accepts user input and reacts to it. The user's input is based on his consideration of the state of the program, and the interaction loop is therefore complete: man acts on machine, machine acts on itself, change in machine state influences man, man acts on machine, machine acts on itself, and so on.

b) Modelling

A computer model of some problem, object or system is the computer representation of that problem, object or system. A model consists of a data structure and associated procedures for meaningfully handling the data structure. Quite often the data structure is considered to be the model, and the procedures a part of the environment. For example, Johnson's CALD (Johnson, 1969) creates a model of a logic circuit, a data structure formed in a way which facilitates analysis of the circuit. Another example is the simulation of a computer, which may be considered to be the construction and use of a model of the computer.

The structure and significance of a computer model depend upon the application for which the model is intended. As an example of how one particular item may be modelled in a number of ways, consider the following models of a straight line displayed on a graphical display unit:

i) The line may indicate a connection between two objects. In this case, assuming the connected objects to be described as blocks, the line may be represented by pointers within the blocks. (Figure 3.1a)

ii) The line may indicate a connection between two objects (such as a wire connecting two electrical elements) and also represent the connecting medium. In this case, the line may be represented as a block containing pointers to the blocks representing the connected objects, which in turn

contain pointers to the block representing the connecting medium. (Figure 3.1b)

iii) The line may have only graphical significance and may be represented as a set of two coordinate pairs, each coordinate pair specifying an end point of a line.

iv) The line may have significance only in that it is part of another object. In this case, there may be no part of the model which directly represents the line. However, as the object containing the line is modelled, the line is implicitly modelled.

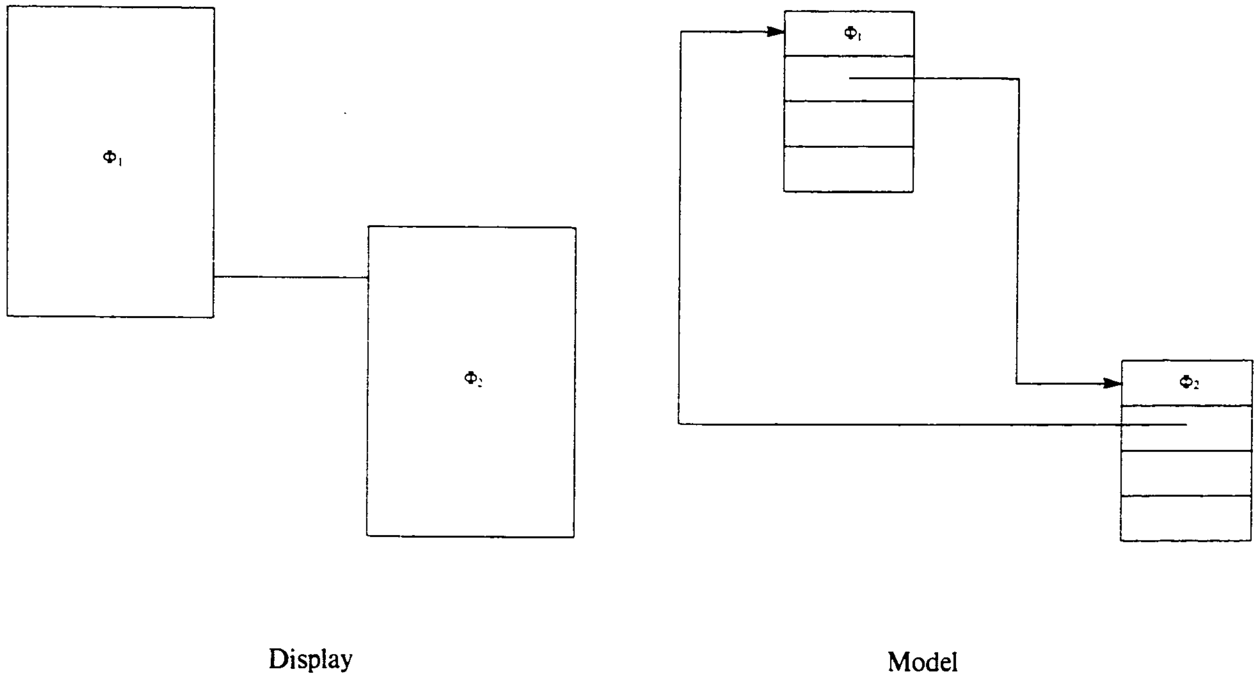
c) Graphical Interface

The form of graphical interface suitable to interactive graphical display programs can be considered in two parts:

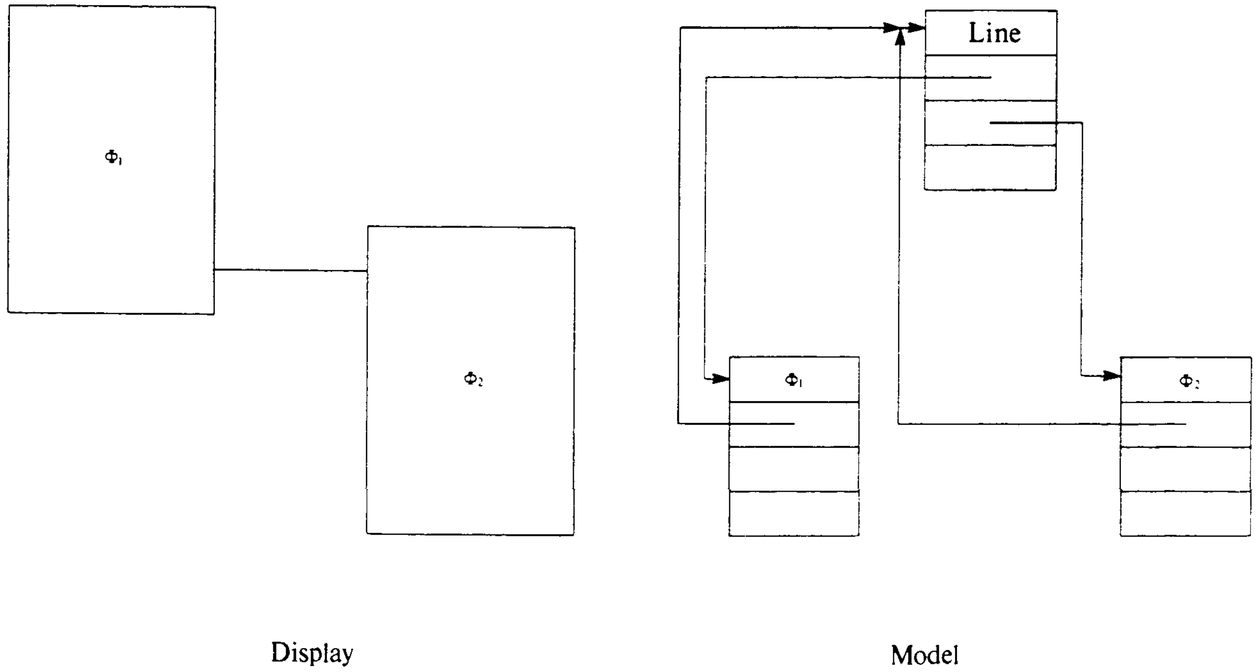
i) facilities for the specification and generation of displays, and

ii) facilities for the decoding of input from the terminal.

The facilities for coping with interactivity, modelling, and graphical interface are the tools with which the programmer may implement an effective form of graphical man-machine communication through the display terminal.



(a) Line Represented by Pointers



(b) Line Represented by a Block

Figure 3.1 Two Models of a Line

3.3 CONCLUSIONS

Being interactive, graphical display programs act like interpreters in that the program interprets a 'language' comprised of actions taken by the console operator. This inspires some to approach interactive graphical display programming as an exercise in language definition and interpreter construction. For example, Deecker's implementation of the SIEVE process (Deecker, 1970) involved the definition of a 'command language', which is executed interpretively. Another example is DL1 (Denil, 1966), which specifies a language that allows the user to use the computer as a three dimensional drafting machine. Such languages are usually formed of linear strings of input actions such as pressing alpha keys and light pen picks of displayed entities.

The use of models by interactive graphical display programs is a logical outcome of attempting to use symbolic graphical communication, that is, graphical communication in which the displayed entities symbolize the system being considered. For example, the pictorial representation of a three dimensional tic-tac-toe cube on a CRT enables the user to visualize the game. However, this representation is symbolic, as a program to play tic-tac-toe would (in all likelihood) use a computer model which is quite different from the visual representation. The model would be designed

to facilitate the program's play and analysis of the game. (A typical model for this program would be an array.)

Another possible use of modelling in interactive graphical display is in the area of graphical interface. In describing a picture, one could be creating a model of the picture which can be used to create the display, and to provide context for input from the terminal. A principal advantage of such an approach is that picture editing is simplified. Such a model and its associated processes could form part of the software support for interactive graphical display programming. To distinguish between a model used for this purpose and a model used to describe a problem system, the author will refer to the former as a display model, and the latter as a problem model.

The characteristics discussed above are fundamental to interactive graphical display programs. As such, they bear examination to yield insight into such programs, and into interactive graphical display support systems.

This approach to understanding the nature of interactive graphical display programs may be contrasted with the "top down" approach utilized by others, such as Ross and Rodriguez (1963). By limiting the area of concern to one particular field, computer aided design, they are able to consider an overall objective of all programs within

this field, and develop their system in the light of this objective - effectively working from the "top down". The chief merit of the approach used in this thesis is the lack of restrictions on the nature of the programs discussed, as the objectives of interactive graphical display programs vary widely.

The next chapter discusses the implications of the characteristics mentioned above on the design of interactive graphical display programming support, in an effort to develop a logical basis for the design of such support.

CHAPTER IV

ON SOFTWARE SUPPORT FOR INTERACTIVE GRAPHICS

4.1 INTRODUCTION

Through developing the concepts of Chapter 3 further, we show in this chapter a functional organization which is typical of interactive graphical display programs. An examination of this organization reveals the elements needed in interactive graphical display programming support. Levels of support are discussed, and the author's ALAS and PRIG are introduced.

4.2 FUNCTIONAL CRGANIZATION OF AN INTERACTIVE GRAPHICAL DISPLAY PROGRAM

Since interactive graphical display programs attempt to exploit interactive graphical man-machine communication, they have certain components in common:

a) Display terminal software, which includes:

i) Software to generate the hardware display commands, and

ii) Software to process input from the terminal.

b) A component which decodes input from the terminal, provides error responses when there is invalid input, and directs control to routines which generate the

computer's responses, and

c) A problem component, which consists of routines which generate the computer's responses and the data on which these responses are based. The problem component may be conceptualized as a model or set of models, which describe the system of interest, and a group of program modules which operate on the model as directed by terminal input. 'Model' here means the problem model discussed previously, although in certain cases a problem model may not exist explicitly, or may be coincident with the display model (see Chapter 3). The author maintains that the problem component of an interactive graphical display program may be considered as

i) a model, whether explicit as a data structure, or implicit within the program, and

ii) a component for model synthesis and analysis. Figure 4.1 illustrates the interaction of the three common components discussed above.

(The reader should note that the display model, a highly specialized type of model, is seen by the author to be part of the display terminal component.)

To illustrate further the functional components of an interactive graphical display program, let us consider the example of a hypothetical program ZOT, which allows the user to design passive electrical networks and to compute their

transfer functions. ZOT has two basic modes:

a) The Synthesis Mode, which provides the facility for entering and altering network designs, and

b) The Analysis Mode, which provides the facility for analyzing a network design.

Through the use of these two modes, the user can complete the design of a passive electrical network. In either mode, information is entered via the terminal. The graphical capabilities allow the user to present the design to the computer in the schematic language used for drawing electrical circuits. This "language" is extremely concise, and the user can express the topology of the circuit and component types and values with relatively few operator actions. For example, placing a resistor into a circuit alters its topology, and is symbolized by the inclusion of a resistor symbol in the schematic. In terms of operator actions, such alterations to the design are brought about by:

a) pressing a key to indicate a request to add to the circuit,

b) picking, with the light pen, a 'prototype' resistor,

c) picking the nodes to which the resistor is to be connected, and

d) entering values for the parameters of the resistor via the keyboard.

The results of analysis of a circuit are presented to the user as performance graphs, for example, time domain or frequency domain response curves, impedance curves, and so on.

The _display terminal component_ of ZOT is responsible for generating displays of circuit diagrams in a way which allows unique identification of the circuit elements. It is responsible for returning to the program various user inputs, such as light pen picks, in the context of the displayed entities. For example, if the operator picks resistor R3, the display terminal software returns this information. The display terminal component is also responsible for generating displays of the results of circuit analysis.

The _input interpretation component_ of ZOT is responsible for generating error messages for invalid input (such as character input when a light pen pick is expected), and for determining which portion of the problem component must be executed. For example, the input sequence described above may be forwarded to the input interpretation component as 'ADD,R,NODE5,NODE6,560 OHMS,0.5 WATT,10%'. On the basis of this, the input interpretation component would call the 'ADD' module of the problem component, with the string, 'R,N5,N6,560,0.5,0.1' as a parameter list.

ZOT uses an explicit model of the circuit being examined, and the distinction between model and program can be easily seen in the problem component of ZOT. The <u>problem component</u> of ZOT is responsible for creating, maintaining and analyzing the circuit model. A typical block and pointer model may be vizualized as having blocks representing circuit elements and multiconnection nodes, and pointers indicating connections to other components. These blocks contain attribute information concerning the element represented. An example of a circuit and its corresponding block-pointer model are shown in Figure 4.2.

An example of a program module which is part of the problem component of ZOT would be the 'ADD' module referred to above. The module would:

a) create the block representing the new resistor,

b) initialize the block, assigning the resistor a name (say R6),

c) locate the blocks for NODE5 and NODE6, and add to them pointers to the block R6,

d) set the pointers in R6 to point to the blocks NODE5 and NODE6,

e) adjust the display model by adding the display of a resistor which is shown as connected to NODE5 and NODE6, and

f) revert to display.

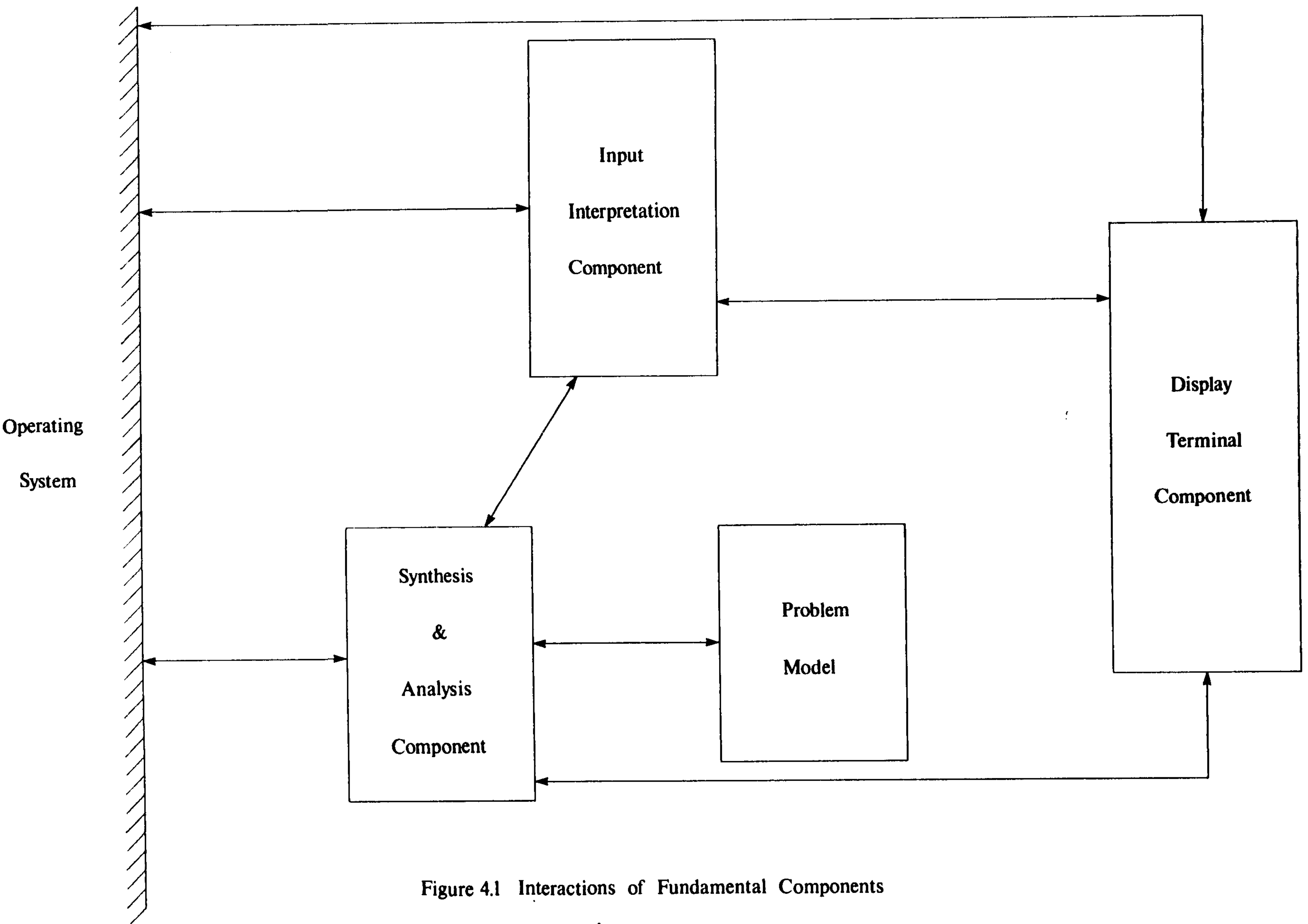
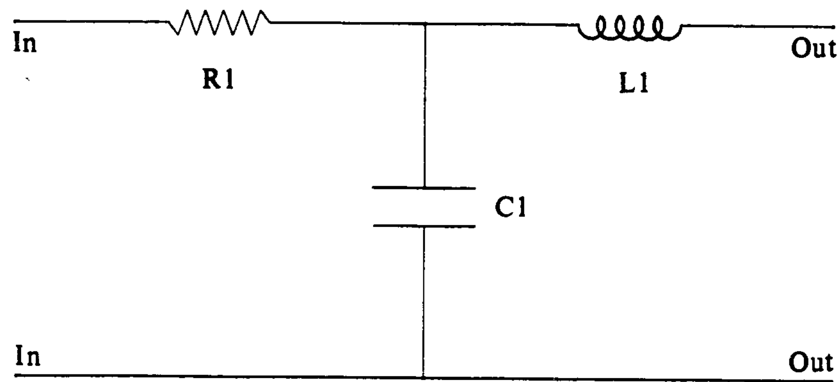
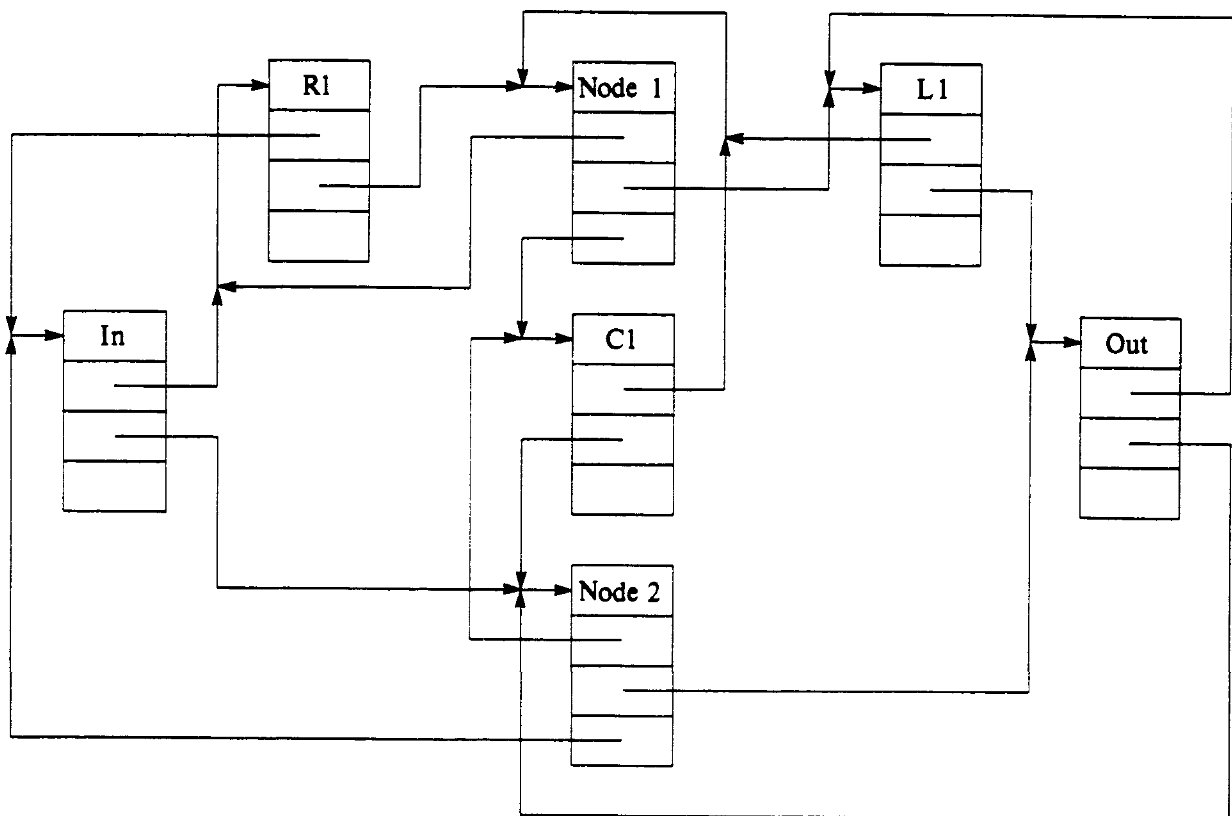


Figure 4.1 Interactions of Fundamental Components



(a) The Circuit



(b) The Model

Figure 4.2 An Electrical Circuit Diagram and Its Model

In this example, virtually all of the actions performed by the module deal with altering the problem model. This is not always the case.

4.3 PROGRAMMING SUPPORT

Support software for a particular class of programming problems provides for programming in that class. For example, the CALCCMP plotter subroutines are designed to make use of the plotter available to the FORTRAN programmer.

Based on the functional description given in Section 4.2, one may project some of the requirements of such support software. Let us consider the components of an interactive graphical display program in terms of their support requirements:

a) <u>Display Terminal Software</u> as discussed in Section 4.2, may be considered in two parts, only one of them program dependent. Certain facilities, such as the controls to display a line or a string of characters, or returning from the terminal a string of characters, are program independent. On the other hand, the facility to determine the significance of a picked line or typed character is program dependent, as these inputs are only significant in the context of a particular program.

Display terminal support software should aid the programmer in building the program dependent portion of the display terminal software. Consider, for example, the block type of organization utilized by GRIDSUB for organizing displays. GRIDSUB allows the programmer to identify displayed elements by numbering "blocks" of display elements. In addition, the programmer can identify different displayed instances of the same block through the use of an ID function. The block number and ID are returned by GRIDSUB upon the detection of a light pen pick.

The program dependence of the display terminal software can be minimized through providing a general hierarchical scheme for creating and describing pictures. A graphical interface based on these concepts should be an integeral portion of interactive graphical display program support.

b) The Input Interpretation Component is almost entirely program dependent. Usually a language is provided with which a programmer may build his own input interpretation component.

There is, however, some merit to considering a table- or structure-driven approach to input interpretation. Newman's system (Newman, 1968) approaches the problem of interactive graphical display programming as a problem in description of the program in terms of actions and

reactions; that is, input actions and program "reactions". Once the description has been created, it may be encoded in the two languages provided:

i) a control-oriented language to encode the state description of the program, and

ii) a procedure-oriented language to encode the procedures required by the program in transferring states.

Newman calls the control-oriented language the Network Definition Language. The interesting thing is that the control portion of the interactive graphical display program is written in NDL, and compiled into a ring structure which is used to drive an interpreter called the Reaction Handler. The Reaction Handler is responsible for the tasks outlined above as being required of the input interpretation component. Thus, table- or structure-driven processing is a possible approach to input interpretation.

c) <u>The Problem Component</u>: The requirements of interactive graphical display programs in the area of problem component support are extremely varied. A large proportion of interactive graphical display programs will require some medium in which they can create and analyze problem models. Facilities adequate for the modelling needed for some problems are available in some general purpose languages, but this is not true for all problems.

Thus, one part of comprehensive support must be a data structure system, which would insure that a programmer could create extensive problem models, and handle them effectively.

For that portion of the problem component which does not handle the model, any of a number of general purpose languages is suitable. However, there is the difficulty of interfacing the modelling system and the general purpose language. One has three possible alternatives:

i) to augment an existing high level language to provide a modelling facility. For example, SLIP (Wiezenbaum, 1963) is a list processor embedded in FORTRAN.

ii) to design an entire general purpose language, which provides the necessary modelling facility as well as the required graphical orientation. Kulsrud (1968) proposes an approach similar to this alternative.

iii) to select a general purpose language and a modelling system, and resolve the interfacing problem.

As alternative iii) requires the programmer to understand more than one language, the author considers it undesirable. Those advocating alternative ii) point out that, in the case of augmented general purpose language, any augmentation will reflect the original language's good and bad features, and the design of the program may therefore be "restricted" by the lack of various facilities in the

original language. People advocating an augmented language point out that programmers who already use the original language need not learn a new language, but merely additions to the old one.

Regardless of which alternative is chosen, the emphasis should be on enhancing the versatility and efficiency of the support software system. The approach the author advocates is alternative ii) with overtones of alternative iii), that is, a general purpose language with adequate interface to allow the use of other languages as desired.

4.4 LEVELS OF GRAPHICS SUPPORT: AN INTRODUCTION TO ALAS AND PRIG

As there is a wide diversity of applications which may make use of interactive graphics, any approach to providing support of such programming has to be flexible. The author believes that the solution to this problem lies in providing a solid foundation upon which support systems of various scopes and levels may be constructed.

As was mentioned previously, interactive graphical display programs have in common a need for a graphical interface, and there are definite advantages to handling this area through the use of a data structure. Also, many graphical applications need a modelling system to facilitate

analysis of problem systems. Therefore, it is obvious that a solid foundation for interactive graphical support software would be a low level, data structure language. While a number of data structure languages are available, their facilities are not always available to the programmer at a low enough level to be used as a basis for support. In Chapter 5, a description is presented of a language which would fulfill the requirement. The language, ALAS (A List Assembler System), is designed specifically to give the programmer an assembler level language with data structure capabilities. Typically, programs in ALAS are assembled to produce code which is a mix of directly executable code and calls to an address translator, storage allocator, and supervisor which make up the ALAS interpreter.

ALAS does not contain a graphical interface facility such as that discussed in Section 4.3. The programmer may create the interface of his choice. Thus, ALAS is the lowest level of graphics support.

The flexibility of ALAS in handling structures can be used to provide a picture modelling and handling system. In Chapter 8, a brief description of such a system is presented. PRIG (Package for Remote Interactive Graphics) is proposed to be implemented in ALAS in the form of a system of macro instructions. It provides the ALAS programmer with a particularly simple picture modelling

capability, coupled with mechanisms to evaluate input in terms of the current picture. PRIG, and other systems at this level, are at the next lowest level of support.

The next level of support would involve a greater degree of automatic graphical input translation. An example would be a system which provides a table driven analyzer which works on graphical input. Implicitly, this involves the creation and use of graphical entities as terminal symbols. The program would have available in some form the parsed string to use as input, thus removing some of the burden of programming at a lower level.

At the highest level of support, the author sees systems such as an entirely table driven system, which localizes the programmer's task to that of specifying his application program as a problem definition in some formalized manner, such as a language specification or a series of decision tables.

The author believes that the construction of systems at each of these levels would benefit from the structure handling capabilities of ALAS. (Table 4.1)

Table 4.1 Levels of Graphical Support

Level	Characteristics	Example
0	no especially graphically oriented support	Assembler, ALAS
1	picture modelling	PRIG in ALAS
2	picture modelling and automated input analysis	table driven syntax analyzer
3	picture modelling, automated program generation and processing	a metacompiler or compiler-compiler system such as Kulsrud's (1968)

CHAPTER V

ALAS - A LIST ASSEMBLER SYSTEM

5.1 INTRODUCTION

In this chapter, a brief description of ALAS is presented. The principle objective of the design is to provide a data structure system in an assembler environment. In specifying an assembler system, the author intends to leave sufficient latitude in the assembly process and machine configuration to allow implementers to create efficient emulations of the ALAS language.

In Chapter 6, some of the more advantageous features of ALAS are discussed and illustrated with a programming example. Chapter 7 discusses some possible methods of implementing ALAS.

In describing ALAS in this chapter, we follow the example of Knowlton's specification of L^6 (Knowlton, 1966) and present the system in a form appropriate for a potential user. Because of the length and degree of detail, we also follow Knowlton by providing an index to the chapter:

INDEX TO CHAPTER V

5.1 INTRODUCTION

5.2 STORAGE ELEMENTS

- 5.2.1 Blocks
- 5.2.2 Fields
- 5.2.3 Templates
- 5.2.4 Block Types

5.3 STORAGE ALLOCATION

- 5.3.1 Static Allocation
- 5.3.2 Dynamic Allocation
- 5.3.3 Storage Elements and Their Allocatability

5.4 ADDRESSING

- 5.4.1 Base Address Generation
- 5.4.2 Modifiers
- 5.4.3 Literals
- 5.4.4 Operand Length
- 5.4.5 Operand Alignment

5.5 REGISTERS

- 5.5.1 Fixed Length Registers
- 5.5.2 Variable Length Registers
- 5.5.3 Register Stacks

5.6 OPERATIONS

- 5.6.1 Condition Code
- 5.6.2 Register and Operand Specification
- 5.6.3 Operation Descriptions

5.7 MACHINE INSTRUCTIONS

5.7.1 General Purpose Instructions

5.7.2 Arithmetic Instructions

5.7.3 Bit and Character Instructions

5.7.4 Addressing Instructions

5.7.5 Conversion Instructions

5.7.6 Allocator Instructions

5.7.7 Control Instructions

5.8 ASSEMBLER INSTRUCTIONS

5.8.1 Listing Control Instructions

5.8.2 Allocator Instructions

5.8.3 Communication Instructions

5.8.4 Others

5.9 THE LOADER

5.10 CONCLUSIONS

5.2 STORAGE ELEMENTS

ALAS has two principal type of storage elements:

a) Blocks, and

b) Fields.

5.2.1 Blocks

A _block_ is a group of consecutive memory locations. The _block address_ is the address of the first location of the group. There is a group of locations at the beginning of each block which provide information concerning the length of the block, and its structure. A block is either _amorphous_ or _prestructured_. An amorphous block has no predefined structure. A prestructured block is one which has had a specification made in its declaration which 'attaches' a template to the block. This template determines the block length, and may be used in referring to the block (Section 5.2.3 and Figure 5.1).

5.2.2 Fields

A field is a memory word of programmer specified length. A field is either

a) a relative field, or

b) an absolute field.

A _relative field_ is not bound to a fixed location in

memory. It is an addressing mechanism used to refer to memory locations in terms of displacement on another address. This will be explained further in Section 5.4. An absolute field is an actual memory area (Figure 5.2).

5.2.3 Templates

To facilitate addressing within an ALAS program, two extended addressing mechanisms are provided under the general heading of Templates. Templates are provided for the construction of and accessing of blocks. The two types of templates are vectors and patterns.

A vector specification is a field containing a field length specification and a quantity. A vector specification is incorporated into the block header of a block defined as a vector. A block so defined can then be accessed by subscripting, as well as by other storage accessing techniques (Figure 5.3).

A pattern specification is a vector (and hence a block) of relative field specifications. The primary value of templates is the capability of organizing groups of relative field definitions for symbolic reference in a structured manner. The field definitions within a pattern may only be referred to in conjunction with the pattern name. A block defined using a pattern has that pattern's address retained

in the header (Figure 5.4).

5.2.4 Block Types

There are four types of blocks:

a) Program block (PBLOCK): - A program block is an amorphous block containing ALAS program code, and is initialized by the assembler on the basis of ALAS instruction records appearing in the block declaration (Figure 5.5).

b) Address block (ABLOCK): - An address block is a vector of addresses, and is initialized by ALAS on the basis of the address constants in the block declaration (Figure 5.6).

c) Static Data block (SBLOCK): - A static data block is a block specified by the programmer which will exist for the duration of the job.

d) Dynamic Data block (DBLOCK): - A dynamic data block is specified by the programmer to be brought into being during execution, and may be discarded under the same conditions.

Certain static data blocks have default definitions. The programmer may define them as he sees fit. (A complete description and discussion of them will not be presented here in the interests of brevity). These are:

a) Interrupt Control Block, a vector of control

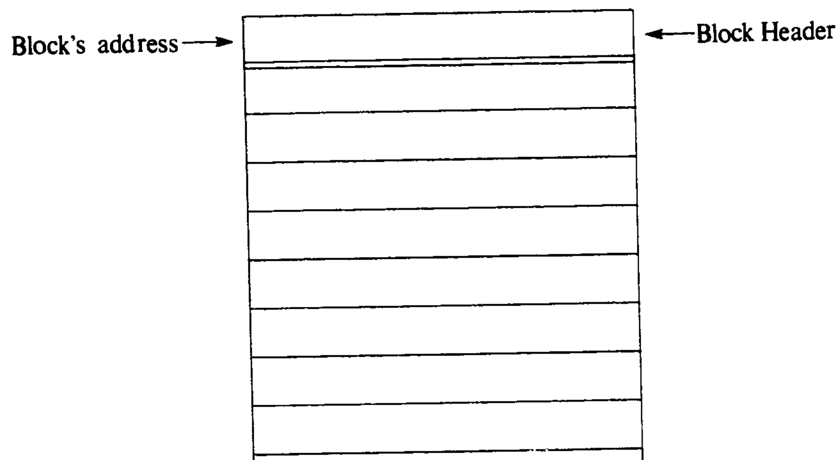


Figure 5.1 A Block

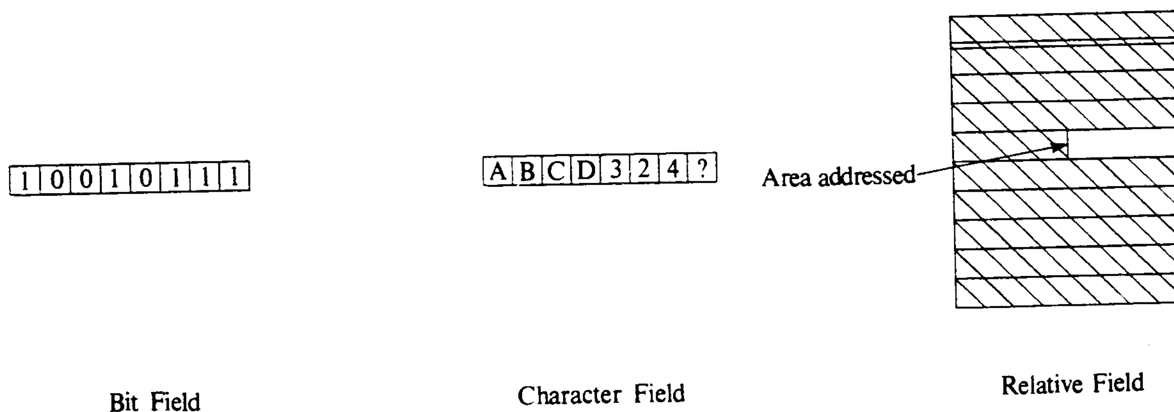


Figure 5.2 Examples of Fields

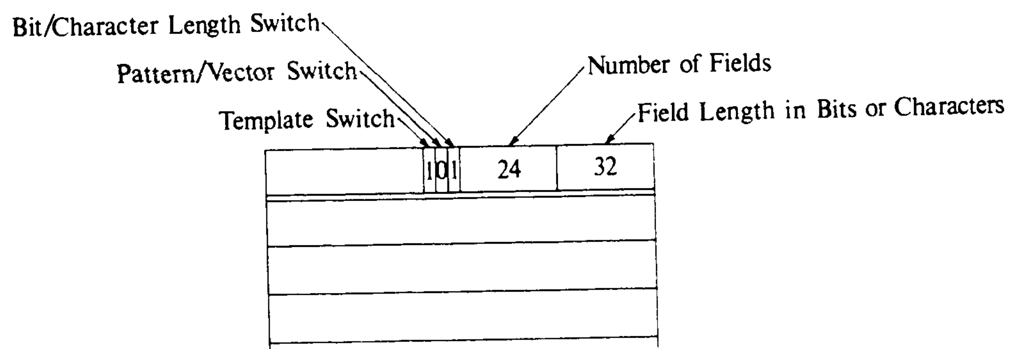


Figure 5.3 Block Header of a Vector

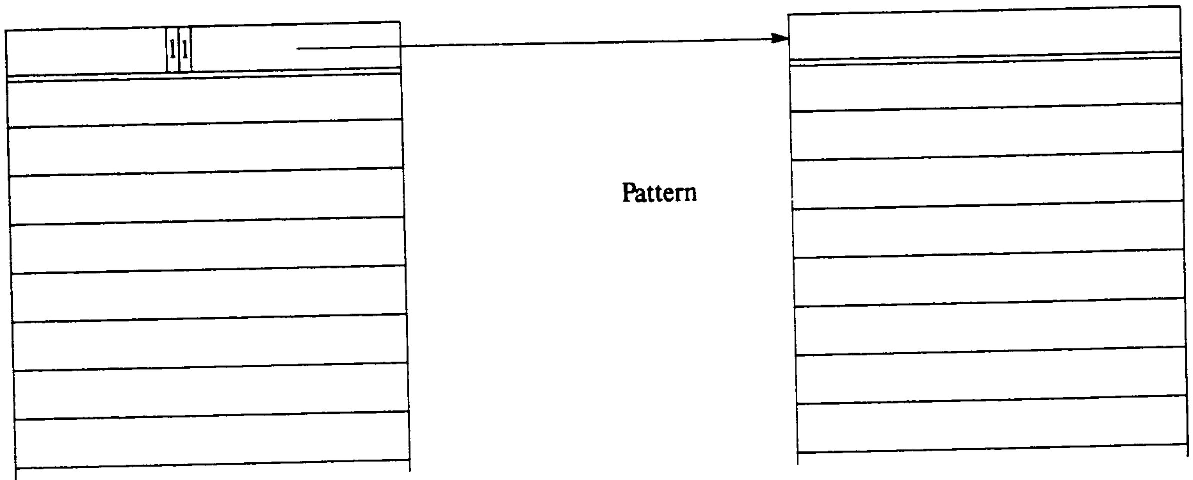


Figure 5.4 Block Associated With a Pattern

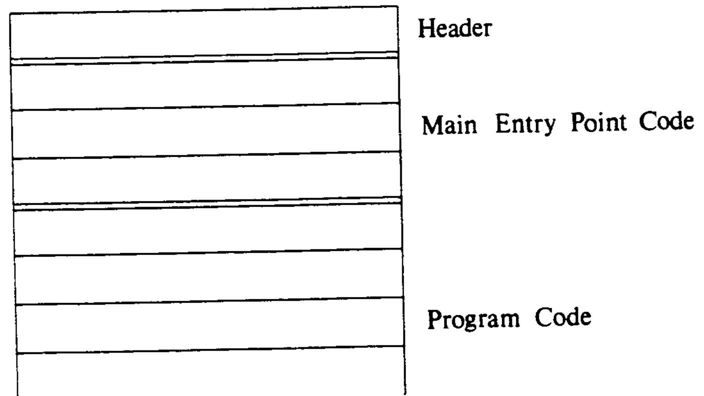


Figure 5.5 PBLOCK

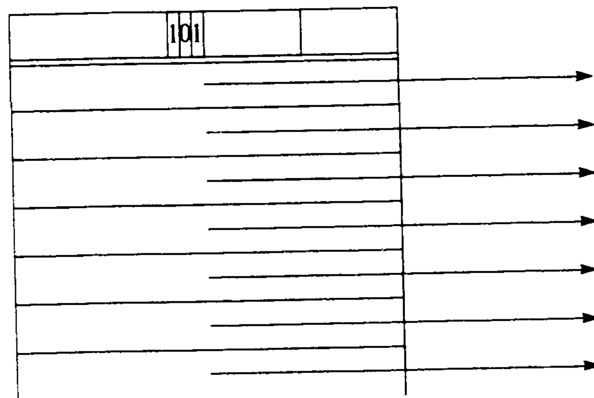


Figure 5.6 ABLOCK

words utilized for directing control upon receipt of interrupts,

b) Transput Control Blocks, vectors of control words for handling I/O devices, and the

c) Allocator Control Block, a vector of control words utilized by the allocator.

5.3 STORAGE ALLOCATION

AIAS has two modes of storage allocation:

a) Static, and

b) Dynamic.

5.3.1 Static Allocation

Static allocation is the assignment of storage prior to program execution. Statically allocated storage cannot be detached during program execution, and has a stable location during execution. References to statically allocated storage can therefore be resolved by the assembler and the loader.

5.3.2 Dynamic Allocation

Dynamic allocation is the process through which storage can be obtained and released under program control. This enables more efficient use of storage, in that a storage

area can be reused by the program in varying configurations.

5.3.3 Storage Elements and Their Allocatability

Table 5.1 illustrates the allocatability status of the various types of storage elements previously described. It should be noted that a statically allocated element cannot be reallocated within the same program; however, a program block overlay feature is available.

Table 5.1 Allocatability of Storage Elements

Element	Static	Dynamic
Program Block	x	
Address Block	x	
Data Block	x	x
Absolute Field	x	x
Relative Field	x	x
Vector	x	
Pattern	x	
Control Blocks	x	

5.4 ADDRESSING

A flexible addressing scheme allows the programmer to intermix references to statically and dynamically allocated

storage and immediate data to create an address. For example, the operand address

AT@B_<I3>+2

where AT refers to a dynamically allocated block,

B is a statically defined relative field,

I3 is an index register, and

2 is immediate data,

is valid, and generates an address as follows:

a) The address of AT is determined, and becomes the base address.

b) The displacement and length of the relative field B is determined, and the base address AT is incremented by the displacement.

c) The indirector (underscore) causes the base address to become the contents of the field addressed by AT@B.

d) The subscript expression <I3> causes the address processor to fetch the header of the block addressed by AT@B_ and to set the base address to the address of the element subscripted by the contents of the register I3.

e) The value 2 is added to the base address, and the result is taken as the actual address of the operand.

The syntax of operand addresses is given in Table 5.2. An operand address string can be used wherever a storage reference is required as an operand.

Table 5.2 Grammar of Operand Addresses

a) Grammar

1. [operand address]::=[base address generator][modifier list option]
2. [base address generator]::=[symbol]•[absolute expression]([address register])
3. [modifier list option]::=[modifier list]•[empty]
4. [modifier list]::=[subfield specifier]•[modifier]•[modifier][modifier list]
5. [subfield specifier]::=:([subtyp],[integer operand option],[integer operand])
6. [subtyp]::=B•C
7. [modifier]::=[absolute expression modifier]•[indexer]•[indirector]•[relative field modifier]•[templater]
8. [absolute expression modifier]::=[sign][absolute expression]
9. [sign]::=+•-
10. [indexer]::=([index operand])
11. [indirector]::=_
12. [relative field modifier]::=@[symbol]
13. [templater]::=[subscriptor]•[pattern reference]
14. [subscriptor]::=<[index operand]>
15. [pattern reference]::=|[symbol]@[symbol]
16. [integer operand option]::=[integer operand]•[empty]
17. [integer operand]::=[integer register]•[index operand]
18. [index operand]::=[index register]•[self defining term]
19. [address register]::=A[hexq]
20. [index register]::=I[hexq]

Table 5.2 continued

21. [integer register]::=F[quartq]

22. [symbol]::=[letter]•[symbol][char]

23. [char]::=[decq]•[letter]

24. [letter]::=A•B•C•...•X•Y•Z

25. [hexq]::=[decq]•10•11•12•13•14•15

26. [decq]::=[quartq]•4•5•6•7•8•9

27. [quartq]::=[binq]•2•3

28. [binq]::=0•1

b) Notation

The notation used is a homomorphism of B.N.F.:

where [ ] are the metalinguistic brackets,

::= is the 'defined as' sign, and

• is the alternation symbol.

c) Examples

Rule   Examples (separated by two blanks)

1   AT@B_<I3>+2

2   AT 72(A3)

3   @B_

4   :(B,,I2) @B @B_:(b,,I2)

5   :(C,3,F2) :(B,,I2)

6   B C

7   +32 (I2) _ @B <I4> |ALPHA@D

8   -75 +2

9   + -

Table 5.2 continued

10	(I5) (12)
11	_
12	@B @DOG
13	<3> |ALPHA@DOG
14	<2> <I3>
15	|ALPHA@DIRT |FMT1@B
16	38
17	F2 I3 62
18	I3 62
19	A12
20	I10
21	F3
22	AT B9 X12P3
23	A 3
24	D X T
25	1 12
26	1 8
27	1 3
28	1 0

Certain elements of the grammar are not completely defined in the table. These elements will be discussed in conjunction with the constructs of which they are part.

An address is created by the processor as follows:

a) The [base address generator] is interpreted to give a machine address and a length specification, and this information is given to the operations processor and execution proceeds.

b) If there are modifiers, they are applied in the order given, using the address and length determined by the preceding modifier, or the base address generator in the case of the first modifier, and they in turn generate new addresses and lengths.

5.4.1 Base Address Generation

A [base address generator] is either a [symbol], (a group of letters and digits, the first of which must be a letter), or a displacement and register specification. To be used as a base address generator, a symbol must be defined in the same PBLOCK as associated with a storage location. This can be done by placing the symbol in the name field of an instruction, by placing the symbol as the first operand of a DBLOCK, DFRD or DFAD instruction, or by placing the symbol in the operand list of an EXTRN or EXTRND instruction.

The displacement and register specification of an address is assembled into a form from which an address may be created by summing the contents of the register and the value of the displacement. The displacement is given as an [absolute expression], which is an individual term which has an absolute arithmetic value, or an arithmetic combination of terms that will yield an absolute arithmetic value and can be processed to yield this value by the assembler.

5.4.2 Modifiers

Modifiers are the working handles which the programmer has on the extended addressing mechanisms. They are:

a) [absolute expression modifier]

b) [indexer]

c) [indirector]

d) [relative field modifier]

e) [templater]

f) [subfield specifier]

An [absolute expression modifier] is a plus or minus sign followed by an absolute expression, the value of which is added to or subtracted from the previous address. The length specification remains unchanged. As an example, the modifier "+32" adds the quantity 32 to the address.

An [indexer] is a bracketed index register specification or self defining term. A self defining term is an unsigned decimal integer or a qualifier followed by a quoted expression in terms specified by the qualifier. The value of the term or the value contained in the register is used to increment the previous address. The length specification remains unchanged. For example, the indexer "(I6)" increments the address by the contents of the register I6.

An [indirector] is an underscore. The [indirector] causes the address to be replaced by the contents of memory currently addressed. The length specification remains unchanged. For example, if the current address refers to a field containing the address of B, modification of the current address by the indirector would cause it to become the address of B.

A [relative field modifier] is the symbol @ followed by a [symbol]. The [symbol] must have appeared in association with a relative field definition. The relative field definition provides the information needed to adjust the address and length information. If B is defined to be the relative field with a displacement of 10 and a length of 8 characters, the modifier @B would increment the current address by 10, and set the length to 8 characters.

A [templater] may be used to provide subscripting or pattern referencing. A subscriptor is the symbol < followed by either a self defining term or an index register reference followed by the symbol >. The value of the expression or the content of the register is used as a zero origin subscript into the vector assumed addressed by the current address. For example, if the current address points to a block defined as a vector, the modifier <5> would set the current address to point to the sixth element of that vector, and set the length to be that of the vector element. A pattern reference is the symbol | followed by a [symbol] followed by a [relative field modifier] which refers to a relative field described in the pattern definition associated with the [symbol]. The [relative field specification] so specified is used to adjust the address and length information appropriately. For example, if the relative field D is defined in the pattern ALPHA as a displacement of three characters and a length of seven characters, the modifier |ALPHA@D would increment the current address by three and set the length to seven characters.

A [subfield specifier] is the final modifier when it appears, and is defined as follows:

[subfield specifier]::=:([subtyp],[integer operand option],
[integer operand])

[subtyp] indicates whether bit (B) or character (C) field

trimming is to be performed, and the second [integer operand] indicates the length of the desired result. The first [integer operand] is a zero origin index indicating the starting point in the field. If the first [integer operand] is omitted, a value of zero is assumed. The address and length specification are adjusted appropriately. For example, the subfield specifier :(B,3,5) would access bits 3 to 7 inclusive of the field currently addressed.

5.4.3 Literals

Literals are operands specified directly in lieu of addressed operands. The assembler builds a table of these items, for which addresses are generated and substituted into code. This table, the literal pool, is created anew for each PBLOCK, and is added to the end of the assembled form of the PBLOCK.

5.4.4 Storage Operand Length Considerations

Depending upon the operation and the form of the operand list, certain storage operands may be required to be of a specific length. The case where a specific length of operand is required is called a <u>hard</u> requirement for a storage operand. The case where a specific length of operand is not required is called a <u>soft</u> requirement.

In the case of soft requirements, the operand length is taken to be that determined by the addressing operation. An example of an operation which has a soft requirement for a storage operand is the load operation applied to a variable length register.

L B3,A

would load the field referenced by A into register B3, regardless of the length of A, and the length of B3 after the operation would coincide with the length of A.

In the case of hard requirements, one of two possibilities occurs:

a) the required length of the operand is equal to the length determined during address calculation. For example, if the block D is defined as a vector of elements of the same length as the integer registers, the operation A which has a hard requirement for a storage operand would encounter this condition in the case of the instruction

A F3,D<2> .

b) the required length of the operand is not equal to the length determined during address processing, in which case, there are two possible courses of action:

i) the length determined during address processing is ignored, and the length of the operand is assumed to be the required length, and the operation proceeds. For example, given block D and operation A which has a hard requirement, the instruction

A F3,D(3)

would use as its operand a field of length required by the operation A, starting at the address D(3).

ii) the length determined during address processing is used to access the operand, which is then altered in length and justification to be of the style required by the operation, and the operation proceeds using the altered form of the operand. For example, given block D as a vector whose elements are not the same length as the integer registers, the instruction

A F3,D<3>

would select the fourth element of the vector D, and adjust its length to match that of the register, and then perform the operation.

The course of action selected in the case of unequal hard required length and generated length is determined by the addressing process. The length specification determined during addressing is either soft (in which case, subcase b) i) is used), or hard (in which case, subcase b) ii) is used). The elements of an address sequence set the solidity of the length specification as shown in Table 5.3. To summarize, the length of an addressed operand which is used in an operation may be determined from Table 5.4.

As an example to illustrate the determination of operand lengths, consider the load operation "L". Depending

upon the type of register selected as a target for the load, the operation has either a soft or a hard length requirement. The block D is defined as a vector of elements whose lengths are not the same as that of the integer registers. The example is given in Table 5.5.

5.4.5 Storage Operand Alignment Considerations

In certain cases of hard length requirements, the operations processor requires alignment of the operand address, i.e. the address is required to be a multiple of a certain quantity. If the addressed length is hard, this requirement (alignment) is assumed satisfied. If the addressed length is soft, the address must be aligned, or alignment is not satisfed. The sensitivity to alignment of storage operands is implementation dependent, and therefore will not be discussed further in this description.

Table 5.3 Solidity of Address Processor Determined Lengths

Base Address Generators		Solidity
[absolute expression]([address register])		soft
[symbol]	if the symbol is a field name	hard
	if the symbol is a block name	soft

Modifiers	Solidity
[absolute expression modifier]	soft
[indexer]	soft
[indirector]	soft
[relative field modifier]	hard
[templater]	hard
[subfield specifier]	hard

Table 5.4 Addressed Operand Lengths

Solidity of Required Operand Length \ Solidity of Addressed Operand Length	Soft	Hard
Soft	Length determined by address processor	Length determined by address processor
Hard	Length required by operation, no adjustment beforehand	Operand is adjusted to fit the required length

Table 5.5 Example of Operand Length Determination

Solidity of Required Length	Solidity of Addressed Length	Instruction		Length Accessed	Length Used
Soft	Soft	L	B3,D+5	Default length of register	Default length of register
Soft	Hard	L	B3,D<5>	Length of vector element	Length of vector element
Hard	Soft	L	F3,D+5	Length of register	Length of register
Hard	Hard	L	F3,D<5>	Length of vector element	Length of register

5.5 REGISTERS

ALAS has a set of registers which are used mainly for holding operands and results of operations. The registers can be divided into two groups, _fixed length_ registers, and _variable length_ registers. Operations performed upon a _fixed length_ register usually involve the entire register. The _variable length_ registers are actually long fixed length registers, which have a length register attached to each of them. The amount of a variable register which is active at any one time is designated by its length register, and may vary under operations performed using that register.

5.5.1 Fixed Length Registers

Fixed length registers are of four basic types:

a) Address registers, of which there are 16, designated A0,A1,...,A15

b) Index registers, of which there are 16, designated I0,I1,...,I15

c) Fixed point registers, of which there are 4, designated F0,F1,F2,F3

d) Floating point registers, of which there are 4, designated E0,E1,E2,E3

Depending upon the machine of implementation, the numeric registers may be extended to provide other

precisions. For example, on IBM 360 machines, in addition to the above, one may find halfword fixed point and doubleword floating point registers. Such things would be considered extensions of ALAS.

5.5.2 Variable Length Registers

Variable length registers are of two types:

a) Bit registers, of which there are 4, designated B0,B1,E2,B3

b) Character registers, of which there are 4, designated C0,C1,C2,C3

The maximum length of a variable length register is 255 elements. Variable length operands may in some cases be trimmed as addressed operands were trimmed. The trimming specifier has the syntactic form

:([integer operand],[integer operand])

where the value of the first integer is taken as the zero origin index of the start of the subregister specified, and the value of the second integer is the length of the subregister specified. For example,

C1:(2,3) specifies characters 2,3 and 4 of register C1; and

B3:(5,9) specifies bits 5 to 13 inclusive of register B3.

Either of the operands may be left empty, in which case certain defaults determined by the operation are assumed. If the second operand is left out, the comma preceding it may also be omitted.

5.5.3 Register Stacks

ALAS registers may be stacked. The stacks are pointer chain constructs stemming from pointer registers associated with the data registers into the dynamic data area. The operations PSH and POP are provided for handling the register stacks.

5.6 OPERATIONS

5.6.1 Condition Code

The operations processor upon completion of some operations sets a four state <u>condition code</u>, which can be tested using the branch or the execute operation.

In Section 5.7, the machine operations are tabulated, and the condition codes they set are described as follows:

a) The codes that can be set by an instruction are listed, and

b) The appropriate row of Table 5.6 is selected. The factor which is tested to set the condition code is

Table 5.6 Condition Codes

Condition Code Set	0	1	2	3
a	zero	negative	positive	empty, or overflow
b	all zero bits	all one bits	mixed zeros and ones	empty
c	1st = 2nd	1st < 2nd	1st > 2nd	empty, or overflow
d	normal	device error	interface error	device not available
e	non-empty	-	-	empty

mentioned in the operation description.

5.6.2 Register and Operand Description

In describing ALAS operations below, a letter selected according to Table 5.7 is used to refer to a register of a particular type. If a register may be one of a number of types, the letters will be enclosed in braces, and separated by commas.

For example,

{F,I}

indicates either a fixed point or an index register. A storage operand is referred to by the letter S. Immediate

Table 5.7 Register Character Codes

Letter	Register Type
A	Address
B	Bit
C	Character
E	Floating point
F	Fixed point
I	Index

operands, which are data operands placed directly into the ALAS machine code instruction, are referred to by the letter D.

A digit may be suffixed to an operand type specification. This indicates that there is a hard length requirement involved, and by duplication (of the suffix digit) within the operand list, the other operands (if any) which are associated with the requirement. The first instance of a suffix digit in an operand list description is considered the binding instance from the point of view of the hard length requirement. For example, the operand list description

{A0,E0,F0,I0},S0

describes an operand list containing two elements, the first of which may be an address register, a floating point register, a fixed point register or an index register, and

the second is an addressed operand with a hard length requirement determined by the first operand.

In some cases (notably with immediate operands) there is a constraint cn an operand which is not directly related to the other operands within the list. In these cases, a digit suffix is used which is not duplicated within the operand list description. For example,

D0,S

describes a two cperand list, in which the first operand is an immediate operand with some particular restriction on it, and the second operand is an addressed operand, without any restriction related to the first operand, (since it is not suffixed by 0), and without any restrictions (since it does not have any numeric suffix).

5.6.3 Operation Descriptions

The following paragraphs describe the set of operations. The mnemonic, the operand lists allowed, and a brief description of the semantics cf the operation are given. The operations are split into two major categories:

a) Machine Instructions, described in Section 5.7, and

b) Assembler Instructions, described in Section 5.8.

5.7 MACHINE INSTRUCTIONS

Machine instructions are organized into seven groups:

a) General purpose

b) Arithmetic

c) Bit and Character

d) Address

e) Conversion

f) Allocator

g) Control

5.7.1 General Purpose Instructions

General purpose instructions perform basic functions which are usually applicable to all of the register types. They will be considered in three subgroups:

a) LOAD Instructions

b) STORE Instructions

c) Initialization, Comparison, and Content Control Instructions

a) LOAD Instructions

Function: The load instructions (Table 5.8) load memory or register contents into the register(s) specified as the first operand(s).

Notes: i) Load and Test, Load and Test Register (LT, LTR) These instructions test the second operand as it is being

Table 5.8 LOAD and STORE Instructions

Instruction Name	Opcode	Operand List	Condition Row	Codes
Load	L	{A0,E0,F0,I0},S0 or {B,C},S	unchanged	
Load Register	LR	{A,B,C,E,F,I},{A,B, C,E,F,I}	unchanged	
Load and Test	LT	{A0,E0,F0,I0},S0 or B,S or C,S	a b e	0,1,2 0,1,2,3 0,3
Load and Test Register	LTR	{A,E,F,I},{A,B,C,E,F,I} or B,{A,B,C,E,F,I} or C,{A,B,C,E,F,I}	a b e	0,1,2 0,1,2,3 0,3
Load Multiple	LM	{A,B,C,E,F,I},{A,B, C,E,F,I},S	unchanged	
Load Specified	LS	{B,C},S	unchanged	
Store	ST	{A0,B0,C0,E0,F0,I0},S0	unchanged	
Store Justified	STJ	{A0,B0,C0,E0,F0,I0},S0	unchanged	
Store Multiple	STM	{A,B,C,E,F,I},{A,B, C,E,F,I},S	unchanged	
Store Specified	STS	{B,C},S	unchanged	

loaded, and set the condition code.

ii) Load Multiple (LM) - This instruction loads a range of registers designated by the first two operands from storage starting at the address given by the third operand. For register ranges extending across register type group boundaries, the following ordering of register type groups is in effect:

A,I,F,E,B,C.

The loading of B or C type registers is performed as for the Load Specified instruction.

iii) Load Specified (LS) - This instruction loads a variable length register with a field stored with a preceding tag which indicates the field's length.

b) STORE Instructions

Function: The store instructions (Table 5.8) store register contents into memory. Store instructions reverse the priorities on operand length determination. The length of the register is always considered hard, and the addressed length is either hard or soft as mentioned previously. The length of the stored data is either the addressed length if the addressed length is hard (which may involve justification and length alteration) or the register length if the addressed length is soft.

Notes: i) Store Justified (STJ) reverses nominal justification of stored data. If, for example, register C1 contains 'ABCD', and field A is only three characters long,

ST C1,A

would result in field A containing 'ABC' (left justification is standard for variable length registers) while,

STJ C1,A

would result in field A containing 'BCD'.

ii) Store Multiple (STM) is the mirror image of Load Multiple (LM). The same considerations apply, except

that data is transfered to memory, instead of from it.

iii) Store Specified (STS) is used to store a variable length register in the form suitable for loading with the Load Specified (LS) instruction.

c) Initialization, Comparison, and Content Control Instructions

Function: These instructions (Table 5.9) are universally applicable, but varied in function. The details of their operation are given in the notes below.

Notes: i) Zero or Empty (ZE) zeros fixed length and empties variable length registers.

ii) Push (PSH) and Pop (POP) are the register stacking instructions. Push stacks the register(s) contents, and Pop unstacks them. The condition code is set by Pop on the basis of the stack size prior to the operation.

iii) Swap, Swap Register (SW, SWR) interchange the contents of their first and second operand addresses. The condition code is set on the basis of the lengths of the operands prior to the operation. Condition code three indicates an overflow caused by one of the operands being larger than its recieving location. The operation is still completed, the offending operand being truncated.

iv) Compare, Compare Register (C, CR) compare the first and second operands. The comparison is logical if the first operand is A, B or C, and arithmetic if the first

Table 5.9 Initialization and Comparison Instructions

Instruction Name	Opcode	Operand List	Condition Row	Codes
Zero or Empty	ZE	{A,B,C,E,F,I} #	unchanged	
Push	PSH	{A,B,C,E,F,I} #	unchanged	
Pop	POP	{A,B,C,E,F,I} #	e	0,3
Swap	SW	{A0,E0,F0,I0},S0 or {B,C},S	c	0,1,2,3
Swap Register	SWR	{A,B,C,E,F,I},{A,B, C,E,F,I}	c	0,1,2,3
Compare	C	{A0,E0,F0,I0},S0 or {B,C} S	c c	0,1,2 0,1,2,3
Compare Register	CR	{A,B,C,E,F,I},{A,B, C,E,F,I}	c	0,1,2,3

# indicates that the operand list may be repeated, separated by commas

operand is E, F or I. The condition code is set based upon the result of the comparison. If a variable length register is the first operand, the comparison is made on a field length that is the shorter of the two operand lengths.

5.7.2 Arithmetic Instructions

Function: The operations associated with most of these instructions (Table 5.10) are evident from the instruction names. Cases where there may be doubt are explained below. The condition code is set according to the result of the operation. The result always replaces the first operand.

Table 5.10 Arithmetic Instructions

Instruction Name	Opcode	Operand List	Condition Row	Codes
Add	A	{E0,F0,I0},S0	a	0,1,2,3
Add Register	AR	{F,I},{F,I} or E,E	a	0,1,2,3
Subtract	S	{E0,F0,I0},S0	a	0,1,2,3
Subtract Register	SR	{F,I},{F,I} or E,E	a	0,1,2,3
Multiply	M	{E0,F0},S0	a	0,1,2,3
Multiply Register	MR	E,E or F,{F,I}	a	0,1,2,3
Full Multiply	FM	F0,S0	a	0,1,2
Full Multiply Register	FMR	F,{F,I}	a	0,1,2
Divide	D	{E0,F0,I0},S0	a	0,1,2,3
Divide Register	DR	E,E or F,{F,I}	a	0,1,2,3
Full Divide	FD	F0,S0	a	0,1,2,3
Full Divide Register	FDR	F,{F,I}	a	0,1,2,3
Negate Register	NR	{E,F,I} or B	a b	0,1,2 0,1,2,3
Load Positive	LP	{E0,F0,I0},S0	a	0,2,3
Load Positive Register	LPR	{F,I},{F,I} or E,E	a	0,2,3
Load Negative	LN	{E0,F0,I0},S0	a	0,1
Load Negative Register	LNR	{F,I},{F,I} or E,E	a	0,1

<u>Notes</u>: i) Full Multiply, Full Multiply Register (FM, FMR) The multiply instructions (M, MR), when applied to a fixed point register of length n, generate only the right hand n bits of the result. If the left hand n-1 bits contain significant information, condition code three is set. Full Multiply generates the entire product, which requires two registers, and stores it in order in the first operand register, and the register immediately following it.

ii) Full Divide, Full Divide Register (FD, FDR) In doing a fixed point divide with the Divide instructions (D, DR), the remainder is discarded. A Full Divide retains the remainder, and places it in the register immediately following the first operand register (where the quotient is placed).

iii) Negate Register (NR) arithmetically negates E, F or I operands, and logically negates B operands.

iv) Load Positive, Load Positive Register (LP, LPR) ensure that the result is positive, negating it if necessary.

v) Load Negative, Load Negative Register (LN, LNR) ensure that the result is negative, negating it if necessary.

5.7.3 Bit and Character Instructions

These instructions operate primarily on variable length registers. They will be considered in three subgroups:

a) Loading and Deleting Instructions

b) Test Instructions

c) Bit Instructions

a) Loading and Deleting Instructions

Function: These instructions (Table 5.11) provide flexible mechanisms for loading and deleting information in variable length registers. The result always replaces the first operand.

Notes: i) The Build instructions (BUL, BULR, BUR, BURR) concatenate the second operand to the first operand. The use of a displaced variable length register as a first operand causes widening of the register at the indicated place. It is widened by the length of the second operand, which is then inserted.

ii) Widen (WD) acts as an immediate data form of the Build Left instruction. The field concatenated is described by the two immediate operands, D1 and D0. D0 must be one element of the same type as the first operand. D1 is an immediate integer specifying the length of the field, in terms of replications of D0.

iii) The Overlay instructions (OV, OVR) load the second operand over top of register contents without altering the remainder of the register. This operation differs from Load in that it does not empty the register before loading, and it differs from Build and Widen in that it does not move register contents before loading.

Table 5.11 Loading and Deleting Instructions

Instruction Name	Opcode	Operand List	Condition Row	Codes
Build Left	BUL	{B,C},S	unchanged	
Build Left Register	BULR	B,B or C,C	unchanged	
Build Right	BUR	{B,C},S	unchanged	
Build Right Register	BURR	B,B or C,C	unchanged	
Widen	WD	{B0,C0},D1,D0	unchanged	
Overlay	OV	{B,C},S	unchanged	
Overlay Register	OVR	B,B or C,C	unchanged	
Eliminate	EL	{B,C},S	a	0,2,3
Eliminate Register	EIR	B,B or C,C	a	0,2,3
Shift	SH	{F,I},S0 or B,S0 or C,S0	a b e	0,1,2,3 0,1,2,3 0,3
Shift Register	SHR	{F,I},{F,I} or B,{F,I} or C,{F,I}	a b e	0,1,2,3 0,1,2,3 0,3
Rotate	RT	{F,I},S0 or B,S0 or C,S0	a b e	0,1,2 0,1,2,3 0,3
Rotate Register	RTR	{F,I},{F,I} or B,{F,I} or C,{F,I}	a b e	0,1,2 0,1,2,3 0,3
Translate	TR	C,S0	unchanged	
Expand	EXP	{B0,C0},B0	unchanged	
Compress	CMP	{B0,C0},B0	unchanged	
Merge	MRG	{B0,C0},S0,B0	unchanged	
Merge Register	MRGR	C0,C0,B0 or B0,B0,B0	unchanged	

iv) The Eliminate instructions (EL, ELR) eliminate all instances of the second operand from the first operand, shortening it appropriately.

v) The Shift and Rotate instructions (SH, SHR, RT, RTR) alter register contents in the following fashion. The second operand is used as a signed integer which specifies the number of positions to be shifted or rotated to the left. Rotation moves elements within a register circularly; i.e. elements moved out of one end of the register are inserted at the other end. Shifting discards elements moved out of the end of a register. The elements moved are bits in arithmetic and bit registers, and characters in character registers. Shifting applied to a variable length register alters the register length.

vi) The Translate instruction (TR) replaces the characters of the first operand by characters selected by using the characters of the first operand as zero origin indices into a table of characters addressed by the second operand.

vii) The Expand, Compress and Merge instructions (EXP, CMP, MRG, MRGR) provide bit field control over the contents of variable length registers. In the case of Expand, the first operand is expanded to be of the same length as the second operand, and contains the base element (i.e. 0 in the case of bit registers, or the character which when evaluated as an integer has the value zero in the case of character registers) at positions corresponding to 0 bit

positions in the second operand. The elements of the first operand are placed in the result at the positions corresponding to 1's in the second operand. Compress is the inverse instruction to Expand. The elements of the first operand selected by corresponding in position to 1's in the second operand are packed, to yield the result. The Merge instructions blend the first and second operands under the control of the third operand by selecting consecutive elements from the first operand for positions in the result corresponding to 0's in the third operand, and consecutive elements from the second operand for positions in the result corresponding to 1's in the third operand.

b) Test Instructions

Function: These instructions (Table 5.12) are used for testing variable length registers. Those instructions marked with an asterisk (*) must have an L or an R concatenated to the opcode which indicates in which direction the scanning is to proceed.

Notes: i) The Size instruction (SIZE) places the length of the first operand in the second operand.

ii) The Count instructions (CNT, CNTI, CNTR) count the number of instances of the second operand in the first operand, and place the quantity in the third operand. In the case of Count Immediate, the immediate data is one element corresponding in type to the first operand. The condition code is set based on the quantity.

iii) The Index instructions (IEF*, IIEF*, IEU*, IIEU*, ISF*) scan the first operand until a particular criterion is satisfied, or the operand is exhausted. Immediate data must be one element corresponding in type to the first operand. If the scanning criterion is satisfied, the index into the register of the first element or the first of a group of elements satisfying the criterion is placed into the third operand. The criteria are as follows:

Mnemonic	Criteria
IEF*, IIEF*	Any element of the second operand is found in the first operand
IEU*, IIEU*	First element found that is not in the second operand
ISF*	The second operand is a substring in the first operand

iv) The Compare Length instructions (CL, CLR, CLIR) compare the length of the first operand with either the length of the second operand (CI, CLR) or the value contained in an integer register (CLIR). The condition code is set by the comparison.

v) The Translate and Test instructions (TRT*) scan by performing a table look-up operation as for the Translate instruction. The scan continues until the first operand is exhausted, or the character accessed from the table is not

Table 5.12 Variable Length Register Test Instructions

Instruction Name	Opcode	Operand List	Condition Row	Condition Codes
Size	SIZE	{B,C},{F,I}	a	0,2
Count	CNT	{B,C},S,{F,I}	a	0,2,3
Count Immediate	CNTI	{B0,C0},D0,{F,I}	a	0,2
Count Register	CNTR	B,B,{F,I} or C,C,{F,I}	a	0,2,3
Index Element Found	IEF*	B,B,{F,I} or C,C,{F,I}	a	0,2,3
Index Immediate Element Found	IIEF*	{B0,C0},D0,{F,I}	a	0,2
Index Element Unfound	IEU*	B,B,{F,I} or C,C,{F,I}	a	0,2,3
Index Immediate Element Unfound	IIEU*	{B0,C0},D0,{F,I}	a	0,2
Index Substring Found	ISF*	B,B,{F,I} or C,C,{F,I}	a	0,2,3
Compare Length	CL	{B,C},S	c	0,1,2
Compare Length Register	CLR	{B,C},{B,C}	c	0,1,2
Compare Length to Int. Reg.	CLIR	{B,C},{F,I}	c	0,1,2
Translate and Test	TRT*	C,S0,{F,I},{F,I}	a	0,2
Membership	MEMR	C,C,B	b	0,1,2,3

* must be replaced by L or R to indicate direction of scan

the base character. In the event of the latter happening, the third operand is set to the index of that character within the first operand, and the fourth operand is set to the numeric value of the table entry.

vi) The Membership instruction (MEMR) sets the third operand to contain non-zero bits where elements of the second operand are found in the first operand, and zero bits elsewhere. The condition code is set on the basis of the number found.

c) Bit Instructions

Function: The following instructions (Table 5.13) affect only bit registers. The operations associated with these instructions are evident from the instruction names. The length of the result is always the same as the length of the first operand. The second operand is always taken as left justified, and is truncated or extended as necessary. If extended, it is extended with bits selected such that the operation on the extended portion does not affect the values of the first operand in this area. The result replaces the first operand.

Notes: i) The And instructions (ND, NDR) perform the logical product operation upon the two operands.

ii) The Or instructions (OR, ORR) perform the bitwise disjoint union of the two operands.

iii) The Exclusive Or instructions (XO, XOR) perform the bitwise disjoint logical sum operation on the

Table 5.13 Bit Instructions

Instruction Name	Opcode	Operand List	Condition Row	Codes
And	ND	B,S	b	0,1,2,3
And Register	NDR	B,S	b	0,1,2,3
Or	OR	B,S	b	0,1,2,3
Or Register	ORR	B,B	b	0,1,2,3
Exclusive Or	XR	B,S	b	0,1,2,3
Exclusive Or Register	XRR	B,B	b	0,1,2,3

two operands.

5.7.4 Addressing Instructions

Function: These instructions (Table 5.14) are used to modify the contents of address registers.

Notes: i) The Point instruction (PNT) loads the address generated for the second operand into the first operand.

ii) The Indirect instruction (IND) causes a standard sized field addressed by the operand to be loaded into the operand.

iii) The Push and Indirect instruction (PIND) does a Push operation followed by an Indirect operation.

iv) The Scan instructions (SCAN, SCANL) scan linked lists. The first operand addresses the list element under

Table 5.14 Addressing Instructions

Instruction Name	Opcode	Operand List	Condition Row Codes
Point	PNT	A,S	unchanged
Indirect	IND	A	unchanged
Push & Indirect	PIND	A	unchanged
Scan	SCAN	A,S0,{B,C},S1,{F,I}	see below
Scan Limited	SCANL	A,S0,{B,C},S1,{F,I},{F,I}	see below

consideration. The second operand is a relative field reference, and in conjunction with the first operand refers to the link field of the list element. The third operand is a variable length register which is used as one of the comparands in the scanning operation. The fourth operand is a relative field reference, and in conjunction with the first operand specifies the field to be compared to the third operand. The fifth operand is a register which contains the count of the list element currently being examined. In the case of SCANL, the sixth operand contains a value which is the upper limit for the fifth operand. The operation proceeds as follows:

a) The fifth operand is zeroed.

b) The second operand is compared with the field addressed by the first and fourth operands.

c) If equality exists, the operation stops

with a condition code of zero. Otherwise, continue.

d) The field addressed by the first and second operands is accessed. If it is zero, the operation stops with a condition code of one. If it is an invalid address, the operation stops with a condition code of three. Otherwise, continue.

e) The fifth operand is incremented. In the case of SCANL, if the fifth operand is now greater than the sixth operand, the operation stops with a condition code of two. Otherwise, continue.

f) The field accessed in step d) is loaded into the first operand, and the process continues at step b).

5.7.5 Conversion Instructions

Function: The Conversion instructions (Table 5.15) are used for converting data from one internal format to another. Data is internally represented in the following formats:

a) Character

b) Bit

c) Floating point (numeric)

d) Fixed point (numeric)

Table 5.16 summarizes the conversions and the applicable instructions.

Table 5.15 Conversion Instructions

Instruction Name	Opcode	Operand List	Condition Row	Codes
Convert Numeric	VNN	{F,I},E or E,{F,I}	a	0,1,2,3
Convert Char's to Numeric	VCN	{E,F,I},C	a	0,1,2,3
Convert Char's to Bits	VCB	B,C	b	0,1,2,3
Convert Floating to Char's: Floating Format	VEC	C,E	unchanged	
Convert Floating to Char's: Exp. Format	VECE	C,E	unchanged	
Convert Fixed to Char's	VFC	C,{F,I}	unchanged	
Convert Bits to Char's	VBC	C,B	unchanged	

Table 5.16 Data Format Conversions

From \ To	Character	Bit	Floating Point	Fixed Point
Character	-	VCB	VCN	VCN
Bit	VBC	-	-	-
Floating Point	VEC, VECE	-	-	VNN
Fixed Point	VFC	-	VNN	-

5.7.6 Allocator Instructions

Function: The Allocator Instructions (Table 5.17) provide control over dynamic storage allocation.

Notes: i) Set Dynamic Space (SDS) and Add Dynamic Space (ADS) do as their names imply. In a program which uses dynamic allocation, there must be one SDS instruction to initialize the allocator. The operand is taken as a positive integer indicating in terms of characters the amount of space to be available for dynamic allocation. ADS alters the amount of dynamic space, its operand indicating the number of characters to be added. SDS and ADS set condition code three to indicate that there is not enough storage available to satisfy their requirements. ADS sets condition code two to indicate that the quantity indicated cannot be deleted, either because the amount of free space is not sufficient, or the amount of dynamic space is not sufficient. Successful completion of these operations sets condition code zero.

ii) Query Free Space (QFS) and Query Dynamic Space (QDS) load the amount of free space and dynamic space respectively into the operand.

iii) The Length of Block instruction (LBL) determines the length of the block addressed by the second operand (in terms of vector elements if it is a vector, in terms of characters otherwise), and loads this quantity into the first operand.

Table 5.17 Allocator Instructions

Instruction Name	Opcode	Operand List	Condition Row Codes
Set Dynamic Space	SDS	{F,I,DO}	see below
Add Dynamic Space	ADS	{F,I,DO}	see below
Query Free Space	QFS	{F,I}	unchanged
Query Dynamic Space	QDS	{F,I}	unchanged
Length of Block	LBL	{F,I},SO	unchanged
Get Dynamic Block	DBLOCK	[dblock oplist] Table 5.18	see below
Get Dynamic Abs. Field	DFAD	[dfad oplist] Table 5.18	see below
Get Dynamic Rel. Field	DFRD	[dfrd oplist] Table 5.18	see below
Drop Block	XBL	[xbl oplist] Table 5.18	unchanged
Drop Field	XFI	[symbol] Table 5.2	unchanged

iv) The block and field allocation instructions (DBLOCK, DFAD, DFRD, XBL, XFI) have a special operand list structure given by Table 5.18.

v) The DBLOCK instruction allocates a block from dynamic storage and associates it with the [symbol] which is the instruction's first operand. The second operand determines the length of the block allocated. In the case

Table 5.18 Grammar of Allocator Instruction Operand Lists

```
[dblock oplist]::=[symbol],[length][repopt]

[dfad oplist]::=[symbol],[field length generator]

[dfrd oplist]::=[symbol],[disp],[field length generator]

[xbl oplist]::=[symbol][repopt]

[repopt]::=[empty]•,[operand address]•,[modifier list]

[length]::=[integer operand]•T[template reference]

[template reference]::='[symbol]'•([integer operand],[field
                        length generator])

[field length generator]::=F'[symbol]'•[bc],[integer
                        operand]

[bc]::=B•C

[disp]::=[integer operand]•([integer operand],[integer
                        operand])
```

Types used from Table 5.2 are:

```
[integer operand]
[symbol]
[empty]
[operand address]
[modifier list]
```

of the length being an [integer operand], the length is the value of the operand in terms of characters. In the case of the length being a [template reference], the length of the template is used. The template may be specified either as a symbol or as a dynamic vector generation. If non-empty, the third operand specifies the location where the address previously associated with the first operand is to be

stored. If this address is specified as a [modifier list], the address of the newly allocated block is used as a base address.

vi) The DFAD instruction allocates an absolute field from dynamic storage and associates it with the [symbol] which is the first operand. The length of the field allocated is determined by the second operand, which either specifies another field or indicates whether the third operand expresses the number of bits or characters.

vii) The DFRD instruction associates a relative field definition with the [symbol] which is the first operand. The second portion of the operand field determines the displacement of the field, and the third portion determines the field length as for the DFAD instruction. The displacement is specified either as a number of characters, or as a number of characters and a number of bits.

viii) The XBL instruction deletes the block currently associated with the [symbol] which is the first operand. If the second operand is non-empty, it addresses a location from which an address may be retrieved to become associated with the first operand. If the second operand is specified as a [modifier list], the address of the block to be deleted is used as a base address.

ix) The XFL instruction deletes the field or field definition associated with the [symbol] which is the operand.

5.7.7 Control Instructions

The following instructions deal with transfers and monitoring of control within the program. As ALAS is oriented to operating under a monitor system which handles input/output, the control instructions include the input/output instructions. These instructions will be considered in twc subgroups:

a) Internal Program Control Instructions

b) Input/Output Instructions

a) Internal Program Control Instructions

Function: For the most part, these instructions (Table 5.19) handle the mechanisms of transfer of control. Transfers of control within a program are performed in two ways:

i) Branching, and

ii) Executing.

Branching involves a direct transfer of control. The address in the operand field is taken as the address of the next executable instruction. Executing involves a transfer of control in a manner which allows control to be returned later to the instruction immediately following the execute instruction. The actual mechanism of an execute instruction involves stacking the current instruction address prior to the transfer. The instruction which reverses the effect of the Execute is the Revert instruction.

<u>Notes</u>: i) The Conditional Branch and Execute instructions (BC, BCR, EC, ECR) test the condition code with a four bit mask, one bit for each possible condition. This mask, the first operand, is specified as immediate data having a decimal value of from zero to fifteen inclusive. The test is considered positive if the current condition code is selected by the mask. For example,

Condition Code 2 : 0 0 1 0

Mask 12 : 1 1 0 0

The test is negative.

Condition Code 3 : 0 0 0 1

Mask 11 : 1 0 1 1

The test is positive.

If the test is positive, the branch or execute is performed, using the address specified by the second operand.

ii) The Zero or Empty Branch and Execute instructions (ZEB, ZEE, ZEBR, ZEER) branch or execute to the second operand address if the first operand is zero or empty.

iii) The Translate and Branch or Execute instructions (TRB, TRE) branch or execute on the basis of the following test. The second operand is a table of characters as for the Translate and Test instruction. The third operand is a table of addresses. The first operand is scanned, its characters being used to look up characters in the second operand. As long as the characters selected are base characters, and the first operand is not exhausted, the

Table 5.19 Internal Control Instructions

Instruction Name	Opcode	Operand List
Branch Conditional	BC	D0,S
Branch Conditional Register	BCR	D0,A
Execute Conditional	EC	D0,S
Execute Conditional Register	ECR	D0,A
Zero or Empty Branch	ZEB	{B,C,E,F,I},S
Zero or Empty Branch Register	ZEBR	{B,C,E,F,I},A
Zero or Empty Execute	ZEE	{B,C,E,F,I},S
Zero or Empty Execute Register	ZEER	{B,C,E,F,I},A
Translate and Branch	TRB	C,S0,S1,{F,I}
Translate and Execute	TRE	C,S0,S1,{F,I}
Revert	REV	
Query Depth of Execute	QDE	{F,I}
Set Condition Code	SCC	{B,F,I,D0}
Loop	LOOP	I,{F,I,D},{F,I,D},{F,I,D}
Loop End	LEND	
Pop Loop Stack	PLOOP	
Query Depth of Loop Stack	QDL	{F,I}

The only instruction in the above table which sets the condition code is SCC.

scan continues. If the first operand is exhausted, processing continues with the next instruction. If a non-base character is looked up, it is used as a zero origin index into the third operand to retrieve an address which is the target of the branch or execute. The index within the first operand of the character which selected the target is placed in the fourth operand.

iv) Revert (REV) pops the execution stack, returning control to the point of the last currently active Execute instruction.

v) Query Depth of Execute (QDE) retrieves a numerical value indicating the current depth of the execute stack.

vi) Set Condition Code (SCC) sets the condition code to the numerical value of the last two bits of the operand if it is arithmetic, or the first two bits of a bit operand.

vii) The instructions LOOP, LEND, PLOOP, and QDL are concerned with the constructing of loops. LOOP initializes the loop in the following way:

a) The first operand is pushed down on its own stack, and is initialized to the value of the second operand.

b) The first operand, the address of the instruction following the Loop instruction (the 'range top address'), and the values of the second and third operands are formed into a composite, and pushed down on the loop

stack.

The LEND instruction examines the top entry on the loop stack, using it in the following way:

a) The first operand is incremented by the value of the fourth operand, and tested against the value of the third operand.

b) If it is less than or equal to the third operand, a branch to the 'range top address' is taken.

c) Otherwise, the first operand is popped, and the loop stack is popped, and processing continues with the next instruction. PLOOP pops an entry from the loop stack, and may be used in the case of abnormal exits from loops. QDL may be used to check the depth of the loop stack.

b) Input/Output Instructions

Function: The following instructions (Table 5.20) handle input/output and calls to programs written in languages other than ALAS.

Notes: i) Input/Output may be executed in-line or concurrent with other programming. The in-line I/O instructions (INP, OUT) have the effect of the out-of-line instructions (BGINP, BGOUT) followed immediately by WAIT instructions. The first operand of the I/O instructions (INP, OUT, BGINP, BGOUT) indicates the device to be used. The second operand is the address of a control string to be used by the device interface. The third operand is the address of a memory buffer into or from which data is to be

Table 5.20 Input/Output Instructions

Instruction Name	Opcode	Operand List	Condition Row	Condition Codes
Input	INP	D0,S1,S1	d	0,1,2,3
Output	OUT	D0,S1,S2	d	0,1,2,3
Start Input	BGINP	D0,S1,S2	d	0,1,3
Start Output	BGOUT	D0,S1,S2	d	0,1,3
Halt I/O	KIO	D0	unchanged	
Device Busy Branch	DBB	D0,S	unchanged	
Device Busy Branch Register	DBBR	D0,A	unchanged	
Device Busy Execute	DBE	D0,S	unchanged	
Device Busy Execute Reg.	DBER	D0,A	unchanged	
Wait	WAIT	or D0,D1,D1,...	d	0,1,2,3
Set Mask	SM	D0,D1	unchanged	
Set Status	SS	D0,D1,D1,...	unchanged	
Call	CALL	D0,S1,S2	unchanged	

transferred.

ii) KIO halts the operation of the device specified as its operand.

iii) WAIT is used to await completion interrupts on I/O devices. An empty operand list indicates that the program is to resume operation after the next completion, or immediately if no devices are active. The non-empty operand

list specifies the number of devices to be watched, and which ones they are. Completion on all specified devices is required before processing continues. The condition code is set to the highest code due to a completion within the group. The control strings for I/O devices contain status information updated by the interfaces. The program can therefore determine the completion status of individual devices even though they may be grouped in the operand list of one WAIT instruction. As control strings are machine and device dependent, they will not be discussed further at this point.

iv) The Device Busy branch and execute commands (DBB, DBBR, DBE, DBER) test the device indicated by the first operand, and branch or execute if it is busy.

v) Set Mask (SM) and Set Status (SS) control the program trapping of interrupts as follows. Set Mask sets a mask used to control which interrupts will be acknowledged. Set Status is used to set or reset interrupt lines and other status controls. These instructions in conjunction with the error and interrupt control blocks mentioned previously afford the programmer a considerable degree of control over interrupt handling. Default masks are set at the outset of the program.

vi) CALL is a macro instruction which is conditionally assembled to provide standard addressing linkages to programs compiled from languages other than ALAS. The first operand indicates the source language of

the called module. The second operand is the address of the module, and the third operand is the address of the parameter address list. The proper formatting of data is the programmer's responsibility.

5.8 ASSEMBLER INSTRUCTIONS

The following section includes instructions which control the assembler and, in some cases, generate machine code. The instructions presented are grouped into four classes:

a) Listing Control

b) Allocation

c) Communication, and

d) Others

5.8.1 Listing Control

The following five instructions provide program control over the assembly listing:

a) Name : Listing off

Mnemonic : LOFF

Operand List : nil

Effect : Listing of assembler output is suspended

b) Name : Listing On

Mnemonic : LON

Operand List : nil

Effect: Listing of assembler output is started

c) Name : Space

Mnemonic : SPACE

Operand List : DO

Effect : The value of the operand determines the number of lines skipped at this point in the listing

d) Name : Eject Page

Mnemonic : EJECT

Operand List : nil

Effect : The listing is continued at the top of the next page

e) Name : Title

Mnemonic : TITLE

Operand List : DO

Effect : The operand must be a character string enclosed in quotes. It replaces the current title. The title appears at the top of each page of the listing. The effect of the Title instruction includes the effect of an Eject instruction.

5.8.2 Allocation Instructions

The following instructions control static allocation.

a) Name : Program Block

Mnemonic : PBLOCK

Operand List : nil

Effect : This instruction generates the main entry point code and initializes the assembler to take ensuing instructions until an END instruction is encountered as an ALAS program module. A name in the label field of a PBLOCK instruction becomes associated with the module as its main entry point, and as the module's name.

b) Name : Static Block

Mnemonic : SBLOCK

Operand List : [empty] or [symbol] or DO

Effect : This instruction causes a static allocation of a block of memory. The name in the label field of an SBLOCK becomes associated with the address of the block. If the operand list of an SBLOCK instruction is not empty, it must either specify a template, whose length determines the length of the block, or specify a number of characters which is to be the size of the block. If the operand list is empty, the size of the block is determined as large enough to enclose the ensuing Define Field instructions, terminated with an END instruction. Define

Field instructions appearing within a static block description perform a special function. Although their form is that of an absolute field definition, the name (if any) on such a DFA instruction becomes associated with a relative field description describing the position within the block of that field. The initialization within the field definition initializes the block.

c) Name : Address Block

Mnemonic : ABLOCK

Operand List : nil

Effect : The ABLOCK instruction initializes the assembler to accept the ensuing instructions to the next END instruction as defining a series of addresses. The name in the label field of an ABLOCK instruction is associated with the block address. The body of the block description consists of DFA instructions whose label fields are ignored, and whose operand fields must contain address references which can be resolved by the assembler or loader.

d) Name : Interrupt Control Block

Mnemonic : INTCB

Operand List : nil

Effect : This instruction is the header of a static block declaration. The block so declared will be attached to the supervisor, and used to control the trapping of interrupts. The content is initialized by ensuing DFA

instructions, whose label fields are ignored. The block has a standard length; however, terminating the declaration with an END instruction is permitted, in which case the undeclared portion is initialized by the assembler to contain the appropriate portion of the default Interrupt Control Block.

e) Name : Allocator Control Block

Mnemonic : ALLCB

Operand List : nil

Effect : This instruction is the header of a static block declaration. The block so declared will be attached to the allocator, and used to control its operation during the execution of the program. The content is initialized by ensuing DFA instructions, whose label fields are ignored. The block has a standard length; however, premature termination of the declaration with an END instruction is permitted, in which case the undeclared portion is initialized by the assembler to contain the appropriate portion of the default Allocator Control Block.

f) Name : Input Output Device Control Block

Mnemonic : IODCB

Operand List : nil

Effect : This instruction is the header of a static block declaration. The block so declared will be attached to the program, and used to assist in performing

input/output operations. The block contains a description of an I/O device, and is initialized by following the IODCB instruction with a series of DFA instructions, terminated by an END instruction. The IODCB instruction must be labelled. IODCB labels are used in DIODE instructions (see below).

g) Name : Define Pattern

Mnemonic : DP

Operand List : nil

Effect : Define Pattern initiates a pattern description. A name must be present in the label field of a DP instruction, and it becomes associated with the pattern. The instructions following DP up to the next END instruction are the body of the pattern description, and must be DFR instructions describing the various fields within the pattern. The associations of names and relative fields within a pattern are local to that pattern and do not affect other instances of those names outside the pattern description.

h) Name : Define Vector

Mnemonic : DV

Operand List : DO,[field length generator]* Table 5.18

Effect : Define Vector defines a vector description and associates it with the name which is present

in the label field of the DV instruction. The operand list consists of an immediate element whose value determines the number of fields in the vector, and a special version of the field length generator as described in Section 5.7.6. The modification consists of prohibiting the use of registers in the specification of the field length.

i) Name : Define Field Absolute

Mnemonic : DFA

Operand List : conforms to Table 5.21

Effect : The DFA instruction may be used in many different areas in an ALAS program. Except as noted, a name appearing in the label field of a DFA instruction becomes associated with the address of the beginning of the field. The operand list of the DFA instruction is processed in the following way:

i) The [repetition factor] (if present) determines the number of copies of the ensuing definition to be generated contiguously within memory. If ommitted, it is assumed to be one.

ii) The [type] indicator indicates the type of terms to be used in initializing if the initialization option is non-empty, and the length of each of the terms in the initialization list if the length option is empty.

iii) The [length option] (if non-empty) specifies the size of each of the terms in the

Table 5.21 Grammar of DFA Operands

```
[dfa oplist]::=[repetition option][type][length option]
                    [initialization option]

[repetition option]::=[empty]•[decimal integer]

[type]::=A•B•C•E•I•X

[length cption]::=[empty]•L[decimal integer]•LB[decimal
                    integer]

[initialization option]::=[empty]•[initialization list]

[initialization list]::='[char string]'•[initialization
                    list[,'[char string]'
```

initialization cption (if it is non-empty). The length is in terms of characters in the case of L, and in terms of bits in the case cf LB. If omitted, the length is assumed to be that specified by the type indicator.

iv) The initialization option (if non-empty) specifies one or more values to be used for initializing memory. The required format of the character string is determined by the type indicator. If more than one value is present, a field of the size determined previously is used for each of the terms. If the initialization option is empty, no initializing is performed. The duration of the association of a name and a field depends upon the location of the DFA instruction. If it is within a PBLOCK declaration, the association is valid within that PBLOCK only, unless the name is mentioned in a GLBL instruction (see Section 5.8.3). If the DFA

instruction is external to any block declaration, the name-field association is considered global, i. e. the name is permanently bound to the location for the purpose of loading, use by program modules, etc. If the DFA instruction is used within an SBLOCK declaration, its name becomes associated with a global relative field definition.

j) Name : Define Field Relative

Mnemonic : DFR

Operand List : conforms to Table 5.22

Effect : The DFR instruction associates the relative field definition presented as the operand list with the name in the label field of the instruction. The first half of the operand list specifies the displacement of the field, in terms of either characters or characters and bits. The second half of the operand list specifies the field length, as for the DV instruction. As with DFA, the position of a DFR instruction determines the duration of the association of the name and field definition.

k) Name : End

Mnemonic : END

Operand List : nil

Effect : END is used to terminate all static allocations which have a declaration requiring more than one instruction.

Table 5.22 Grammar of DFR Operands

[dfr oplist]::=[bc][decimal integer],[decimal integer option][bc]

[decimal integer option]::=[empty]•[decimal integer]

[bc]::=B•C

1) Name : Declare I/O Device

Mnemonic : DIODE

Operand List : DO,[symbol]

Effect : This instruction informs the assembler that a particular device (indicated by DO) will be attached to the processor during program execution. The [symbol] is a label of an IODCB block. If a name is present in the label field, it becomes associated with the device, and may be used in specifying the device in lieu of DO.

5.8.3 Communication Instructions

The following instructions provide for the communication of address information between program modules. The assembler and loader resolve references created by these instructions.

a) Name : Entry

Mnemonic : ENTRY

Operand List : a group of one or more [symbol]s, separated by commas

Effect : This instruction denotes certain symbols within a PBLOCK as entry points, i.e. points which may be used in other modules in Execute or Branch instructions. Such symbols become global by virtue of being in the operand list of an Entry instruction. The instruction also causes the necessary multiple entry point code to be generated.

b) Name : Global

Mnemonic : GLBL

Operand List : a group of one or more [symbol]s, separated by commas

Effect : Name-address associations within a PBLOCK are implicitly localized to the PBLOCK. This instruction provides a mechanism for overriding the implicit localization. Symbols listed in the operand list of a GLBL instruction are marked by the assembler for consideration by the loader at load time. Symbols in the operand list of a GLBL instruction must be associated with addresses of statically allocated entities.

c) Name : Global Dynamic

Mnemonic : GLBLD

Operand List : a group of one or more [symbol]s, separated by commas

Effect : As with statically allocated name-address associations, dynamically allocated name-address associations are normally implicitly restricted to access within the PBLOCK containing the allocation. GLBLD is used to override this phenomenon. A similar process to that referred to for resolving statically allocated name-address associations during loading is used to construct a dynamic dictionary. The dynamic dictionary has two distinct portions, a public portion, and a private portion. The public portion contains all global dynamically allocated name-address references; and the private portion contains all locally dynamically allocated name-address references. The presence of a dynamically allocated symbol in a GLBLD operand list promotes the name-address reference from the private portion of the dynamic dictionary to the public portion.

d) Name : External

Mnemonic : EXTRN

Operand List : a group of one or more [symbol]s, separated by commas

Effect : The EXTRN instruction is used to identify symbols used within a PBLOCK that are static global

symbols. The localization of symbols within a PBLOCK also has the effect that symbols defined outside a PBLOCK cannot be accessed within that block. The EXTRN instruction overrides this function. (Note: There is one exception to the exclusion phenomenon mentioned herein, and that is the case of symbols used as targets in branch and execute instructions. In that case, the symbols are implicitly declared external if they cannot be resolved internally.)

e) Name : External Dynamic

Mnemonic : EXTRND

Operand List : a group of one or more [symbol]s, separated by commas

Effect : This instruction has the same effect as EXTRN, except that the symbols involved must be dynamically defined.

5.8.4 Others

Other instructions available in ALAS present a conditional assembly and a macro facility. These must be omitted at this time in the interests of brevity.

5.9 THE LOADER

The loader creates a core image of a program by combining the various outputs from the assembler, other

language processors, and libraries according to control information which may or may not be provided by the programmer. It loads the necessary system support and compiles the dynamic dictionary. It resolves intra-modular references and provides error information as necessary. An additional feature of the loader, visualized, but not completely formalized at this point, is an overlay feature which would allow program controlled overlaying of statically allocated storage. The loader control input format will not be detailed at this time.

5.10 CONCLUSION

ALAS's principal operational points have been presented above. The format of ALAS is primarily assembler level in nature. The author feels that the intrinsic tedium of the medium is more than outweighed by the attendant flexibility.

In the next chapter, some of the more important advantages of ALAS are discussed, and a programming example is presented. Chapter 7 discusses three techniques of implementing ALAS.

CHAPTER VI

DISCUSSION AND EXAMPLE

6.1 INTRODUCTION

In this chapter, the author discusses what he considers to be the most important characteristics of ALAS:

a) Address processing mechanism,

b) Static and dynamic storage allocation, and

c) Variable length registers.

The value of a programming language can only be determined by its use, and an example has therefore been given in Section 6.5.

Throughout this chapter, illustrative comparisons between ALAS and other languages, notably L^6, are presented. The choice of L^6 for purposes of comparison was made primarily because L^6 is, like ALAS, a low level language, designed to be almost universal in its applications.

6.2 ADDRESS PROCESSING

In an assembler language which is not data structure oriented, data structure programming can involve complicated address manipulation, a direct consequence of the extensive

use of pointers. To provide for easier handling of address manipulation, ALAS has an address processing feature which allows for extensive processing of various components within an address specification used as an operand. The address processor affords the following facilities:

a) Direct addressing,

b) Indirect addressing,

c) Subscripting (use of vectors),

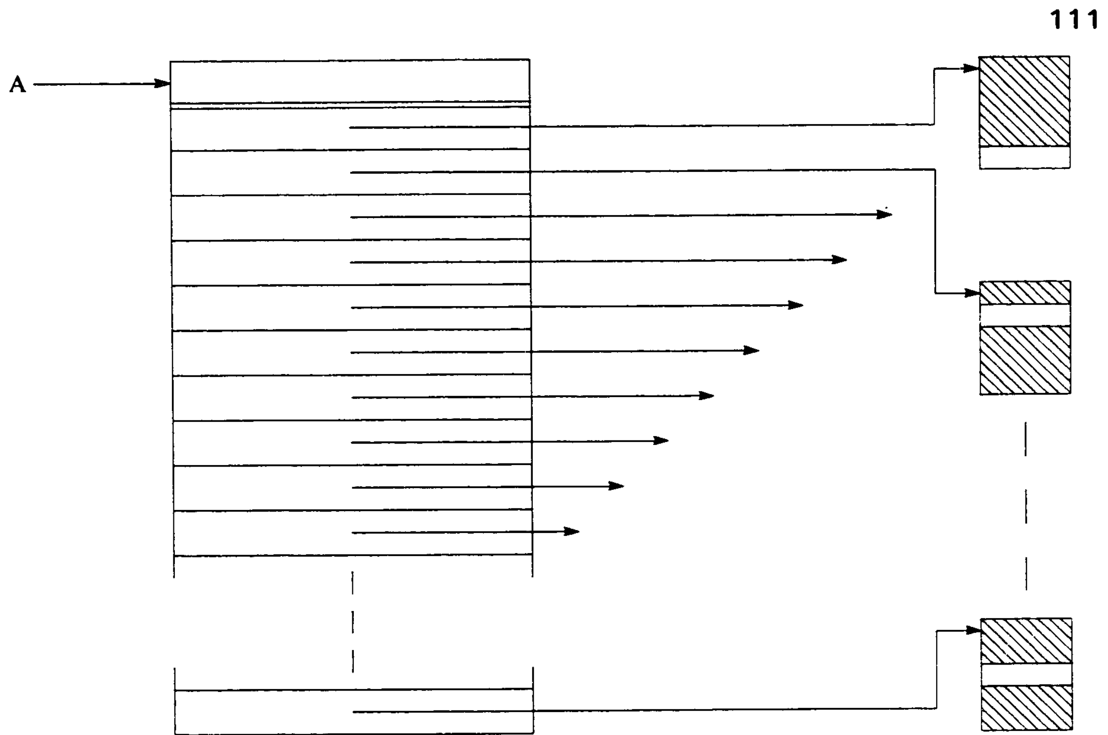
d) Symbolic displacement (the use of relative field definitions),

e) Indexing, and

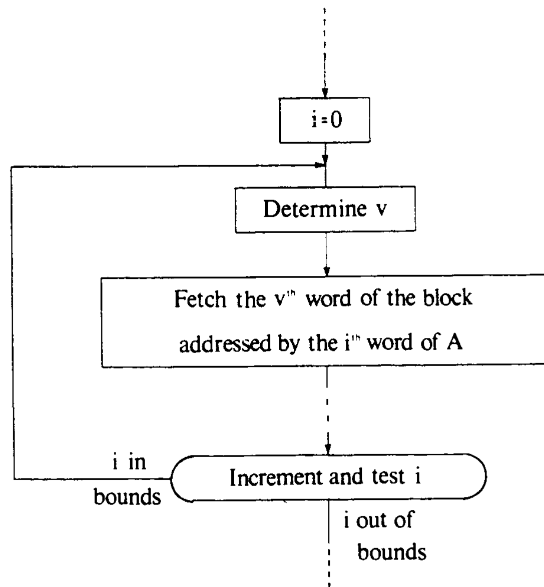
f) Arithmetic expression displacement.

In comparison, consider L^6, another low level data structure language. In particular, its addressing feature allows no direct addressing of storage, but only indirect addressing. It allows for symbolic displacement, but no indexing, subscripting, or arithmetic expression displacement.

To illustrate the significance of these differences, suppose that a block of memory is set aside to contain a series of fields, all of the same length. These fields each contain the address of a block elsewhere in memory (Figure 6.1a). The block of addresses is to be used to access, within a loop, words in other blocks whose position within the block depends on the value of a variable (Figure 6.1b).



(a) Structure to be Accessed as in (b)



(b) Flowchart of the Loop Containing the Access

Figure 6.1 Accessing Example

In L^6, it is necessary to redefine relative fields within the loop, as the relative field mechanism is the main displaced addressing facility. In ALAS, however, the access may be performed using subscripting and indexing (as well as indirect addressing), all within the operand specification.

a) <u>L^6 code</u>: Assume that the variable displacement is in bug V, the loop index is in bug I, and the address of the block of addresses is in bug A. To get the result into a bug, three instructions are required:

```
(V,DV,16,31),(I,DI,0,31),(X,E,AIV)
```

b) <u>ALAS code</u>: Assume that the variable displacement is in register I1, the loop index is in register I0, and the address of the block of addresses is associated with the name A. To load the same field into a register, only one instruction is required:

```
L          I2,A<I0>_(I1):(B,16,16)
```

Another indication of the difference in power between the two systems' addressing features is the relative ease with which the ALAS address mechanisms can duplicate those of L^6, while the converse is quite complicated. For example, assume that the relative fields are defined to be the same in both systems, and bug A (in L^6) points to the location with which the name A (in ALAS) is associated. Then the L^6 address ABCD can be duplicated in ALAS as A@B_@C_@D. However, the subscripting mechanism in ALAS (to consider just one example) can only be duplicated in L^6

through computation: Consider an ALAS vector of fields 24 bits long. The ith element of the vector can be accessed by the address A<I0>, if register I0 contains the subscript i. In L^6, the program must determine the word or words which contain the field or portions of it, and define the appropriate field or fields, and perform the access by merging, if necessary, the results of the two relative field accesses.

The addressing mechanisms of ALAS are not solely oriented toward data structure processing, which is an advantage in that programmers are not restricted to a particular technique of accessing memory. In LISP, for example, the programmer has an associative addressing feature which, combined with the operations CAR and CDR, forms the basis of all memory accessing. While a technique of this simplicity is aesthetically attractive, it is inconvenient when dealing with data that cannot be easily formed into binary trees. The absolute field of ALAS could be used to handle trees in a similar way to LISP, while other mechanisms in ALAS are more suited to the data that cannot be handled conveniently in the form of a binary tree.

6.3 STORAGE MANAGEMENT

In ALAS, storage allocation can be performed either statically, or dynamically. In the case of statically

allocated storage, the assembler can resolve more of the addressing, and the resultant code can be more efficient, than if the storage is allocated dynamically. By contrast, LISP and L^6 do not allow static allocation.

ALAS storage is block and field oriented. In some ways, this is similar to the L^6 system, but ALAS provides also for absolute field allocation. An absolute field is a memory word of programmer defined length. The distinction between a block and an absolute field lies in the existence of the "header" field at the beginning of a block. A block header contains information which is used by the address processor and by the allocator, and is part of the price paid for the extended addressing mechanisms present in ALAS. An absolute field does not have a header, and the use of extra storage for block headers can be avoided.

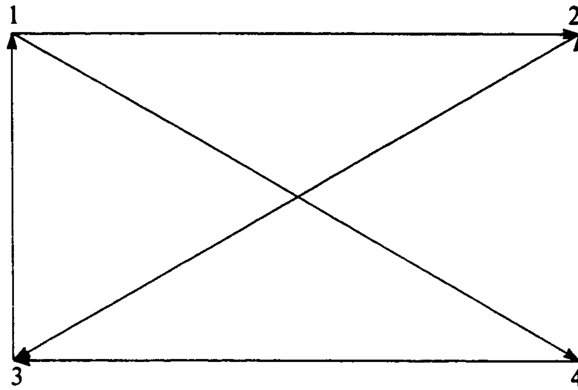
6.4 VARIABLE LENGTH REGISTERS

The variable length registers of ALAS represent a departure from commonly accepted machine architecture, and complement the use of variable length fields in memory. The registers are intended for character and bit string processing, which is important for data structure programming as character or bit strings are often used.

As a simple example, consider the representation of a graph by its incidence matrix, a Boolean matrix A whose elements a_{ij} are true if there is an arc from node i to node j, and false otherwise. While no low level language handles this representation in a particularly convenient manner, ALAS's variable length bit registers and fields are more suitable in this situation than those of, for example, L6. A typical representation of such a matrix in ALAS would be a block, a vector of length n, with fields in the vector being n bits long. The block would contain a direct transcription of the matrix; that is, the ith element of the would be an n-bit field which represents the ith row of the incidence matrix (Figure 6.2).

The variable length registers and string operations of ALAS would also be useful in synthesizing the facilities of certain higher level languages (such as SNOBOL) for string processing. Using the ALAS operations, the programmer may create string processing packages which are tailored to particular applications, thus gaining efficiency over similar systems written in a higher level language environment.

For example, consider the interpretive processing of FORTRAN format statements. Part of the processing is concerned with the recognition of the various format codes and transfer of control based on the code recognized (See



(a) The Graph

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

(b) Its Incidence Matrix

0	1	0	1
0	0	1	0
1	0	0	0
0	1	1	0

(c) The ALAS Vector

Figure 6.2 A Graph and Its Representations

Lee (1967), Figure 6.1). In SNOBOL, branching is limited to one binary choice of exit per test. In ALAS, there are the instructions TRB and TRE, which allow the choice of up to 255 exits on one test. To branch according to the format code at the current pointer position would require as many SNOBOL statements as format codes, statements which perform a linear search of the list of codes. In ALAS, however, a single instruction could perform the same job.

6.5 A PROGRAMMING EXAMPLE

As a programming example, consider the use of the graph representation shown in Figure 6.2 in decomposing a graph into its strongly connected components. The strongly connected component of a vertex i of the graph $G = (N,\Gamma)$ is the graph $G_i = (N_i,\Gamma_i)$, where N_i is the set consisting of i and of those vertices $j \in N$ for which there exists a circuit in G passing through i and j, while $\Gamma_i j$ is defined for $j \in N$ as being $\Gamma_i j = \Gamma j \cap N_i$, i. e. the set of those $k \in \Gamma j$ which belong also to N_i (Hammer and Rudeanu, 1967, Chapter 9). The object of the decomposition is the generation of m pairwise disjoint subsets $N_1,N_2,\ldots,N_m$ of the set of vertices N.

The decomposition may be performed as follows:

a) Given the incidence matrix A of the graph, determine the incidence matrix $\hat{A}$ of the transitive closure

of the graph. This may be performed by determining the incidence matrix $\check{A}$ of the algebraic closure of the graph, and using the identity $\hat{A} = E \cup \check{A}$; where E is the square Boolean matrix with 1's on the diagonal, and 0's elsewhere. $\check{A}$ may be determined through the successive application of simple transformations to the incidence matrix A. The transformation T_i (i=1,...,n) is defined as follows. If $A = \{a_{hk}\}$, $T_i A = \{\alpha_{hk}\}$, where $\alpha_{hk} = a_{hk} \vee a_{hi} a_{ik}$. In order to compute row h of $T_i A$, one proceeds as follows. If $a_{hi} = 0$, then the row h of $T_i A$ coincides with row h of A: $\alpha_{hk} = a_{hk}$ (k=1,...,n). If $a_{hi} = 1$, then row h of $T_i A$ is the disjunction of rows i and h of A: $\alpha_{hk} = a_{hk} \vee a_{hi}$ (k=1,...,n). It can be shown that $\check{A} = T_n T_{n-1} \cdots T_1 A$.

The following PBLOCK accepts a square Boolean matrix A pointed to by the register A0, and generates the incidence matrix $\hat{A}$ of the transitive closure of the graph whose matrix is A.

```
MBCLOSE      PBLOCK
             PSH       I0,I1,I2,B3
             LBL       I0,0(A0)                GET N
             DBLOCK    AT,T(I0,B,I0)           GET AT
             S         I0,ONE
*   THIS LOOP COPIES A INTO AT
             LOOP      I1,0,I0,1
             L         B3,0(A0)<I1>
             ST        B3,AT<I1>
             LEND
             LOOP      I1,0,I0,1               LOOP THRU T(I)
             LOOP      I2,0,I0,1               LOOP FOR T(I)
             LT        B3,AT<I2>:(B,I1,1)      IS A(H,I) = 0?
             BC        8,ENDLP                 IF SO, GET OUT
             L         B3,AT<I2>               ELSE ROW H =
             OR        B3,AT<I1>               ROW H OR ROW I
```

```
                ST        B3,AT<I2>
ENDLP           LEND
                LEND
* AT THIS POINT AT IS THE ALGEBRAIC CLOSURE
                L         B3,=B'1'
* THIS LOOP MAKES IT THE TRANSITIVE CLOSURE (SEE TEXT)
                LOOP      I1,0,I0,1
                ST        B3,AT<I1>:(B,I1,1)
                LEND
                PNT       A1,AT                 LOCATE AT
                POP       I0,I1,I2,B3           CLEAN UP
                REV                             RETURN
ONE             DFA       I'1'
                END
```

b) Given the transitive closure of the graph, the determination of the strongly connected components of a graph, and its reduced graph, is simply the determination of the following two matrices:

i) the decomposition matrix $D = \{d_{ij}\}$ having n rows and m columns, where

$$d_{ij} = \begin{cases} 0 \text{ if } i \notin N_j, \\ 1 \text{ if } i \in N_j; \end{cases}$$

ii) the incidence matrix A^R of the reduced graph G^R.

These matrices may be determined through the use of two simple transformations $A \rightarrow A^V$ and $A \rightarrow A^W$, defined for square (say, n by n) Boolean matrices A:

i) The matrix A^V is obtained in m $(m < n)$ steps, each one of them consisting of the deletion of certain columns. More precisely, in step p $(p=1,\dots,m)$ we consider the first column h which remained undeleted after the previous steps, and delete all of those columns j for

which $j > h$ and $a_{jh} = 1$.

ii) A^W is the matrix which coincides with the matrix A except on the diagonal, where it has only zeros.

It can be shown that $D = [\hat{A} \times (\hat{A}^T)]^V$, and $A^R = (D^T \times A \times D)^W$. Given the transitive closure of the graph, represented by $\hat{A}$, the determination of the decomposition is accomplished as follows:

i) Form $\hat{A} \times (\hat{A}^T)$

ii) Transform $[\hat{A} \times (\hat{A}^T)]$ to give D

iii) Form $D^T \times A \times D$

iv) Transform $D^T \times A \times D$ to give A^R

To perform part b) of the decomposition, the author programmed the following routines:

i) A matrix transposition routine

ii) A matrix multiplication routine

iii) A V transform routine

iv) A W transform routine

The matrix transposition routine and the matrix product routine are complicated somewhat by the vagaries of columnar access in the representation utilized in this example. However, the operations which may be performed simultaneously upon a complete row of a matrix in this representation more than repay accessing difficulties. For example, the fragment

```
LR        B2,B3
ND        B2,0(A0)<I5>
```

```
                BC        6,ONEL
```

performs the expression $\bigcup_{j=1}^{n} a_{ij} b_{jk}$, exiting to 'ONEL' if the result is 1, dropping through if the result is 0. Without ALAS's bit registers and operations, programming this portion of the matrix multiplication would require a loop, possibly with two double subscript accesses within it.

The following modules perform the operations necessary for part b) of the decomposition described above:

i) Matrix Transpose - This PBLOCK accepts a Boolean matrix represented as above, and creates its transpose. Upon entering the routine, it is assumed that register A0 points to the matrix, and upon returning, register A1 points to the transpose.

```
MBTRANS        PBLOCK
               PSH       I0,I1,I2,I3,B3
* DETERMINE SIZE OF MATRIX - I1 BY I0
               L         B3,0(A0)<0>
               SIZE      B3,I0
               LBL       I1,0(A0)
* GET BLOCK FOR TRANSPOSE
               DBLOCK    AT,T(I0,B,I1)
* DECREMENT INDICES TO ALLOW FOR ZERO ORIGIN
               S         I0,ONE
               S         I1,ONE
* LOOP THROUGH THE ROWS OF A
               LOOP      I2,0,I1,1
* ACCESS I2TH ROW OF A
               L         B3,0(A0)<I2>
* LOOP THROUGH THE COLUMNS OF A
               LOOP      I3,0,I0,1
* SEED A(I2,I3) AT AT(I3,I2)
               ST        B3:(I3,1),AT<I3>:(B,I2,1)
               LEND
               LEND
               PNT       A1,AT                  LOCATE AT
               POP       I0,I1,I2,I3,B3         CLEAN UP
               REV                              RETURN
ONE            DFA       I'1'
```

END

ii) Matrix Product - This PBLOCK accepts two matrices, pointed to by A0 and A1, and generates the matrix product defined as follows:

If $A = \{a_{ij}\}$: $i=1,\ldots,m$; $j=1,\ldots,p$, and

$B = \{b_{jk}\}$: $j=1,\ldots,p$; $k=1,\ldots,n$, then

$A \times B = C = \{c_{ik}\}$: $i=1,\ldots,m$; $k=1,\ldots,n$,

where $c_{ik} = \bigcup_{j=1}^{p} a_{ij} b_{jk}$.

```
MBPRCD          PBLOCK
                PSH       I0,I1,I2,I3,I4,I5,B1,B2,B3
* DETERMINE SIZE OF MATRIX A - I0 BY I1
                LBL       I0,0(A0)
                L         B3,0(A0)<0>
                SIZE      B3,I1
* DETERMINE SIZE OF MATRIX B - I1 BY I2
                L         B3,0(A1)<0>
                SIZE      B3,I2
* GET THE BLOCK FOR THE PRODUCT
                DBLOCK    C,T(I0,B,I2)
* DECREMENT INDICES FOR ZERO ORIGIN INDEXING
                S         I0,ONE
                S         I1,ONE
                S         I2,ONE
                L         B1,=B'01'
* LOOP THROUGH THE COLUMNS OF THE PRODUCT
                LOOP      I4,0,I2,1
                ZE        B3
* THIS LOOP BUILDS THE I4TH COLUMN OF B IN B3
                LOOP      I3,0,I1,1
                BUR       B3,0(A1)<I3>:(B,I4,1)
                LEND
* THIS LOOP PRODUCES A COLUMN OF THE PRODUCT
                LOOP      I5,0,I0,1
                LR        B2,B3
                ND        B2,0(A0)<I5>
                BC        6,ONEL
                ST        B1:(0,1),C<I5>:(B,I4,1)
                B         MLOOP
ONEL            ST        B1:(1,1),C<I5>:(B,I4,1)
MLOOP           LEND
                LEND
                PNT       A2,C                    LOCATE C
                POP       I0,I1,I2,I3,I4,I5,B1,B2,B3 CLEAN UP
                REV                               RETURN
```

```
ONE             DFA       I'1'
                END
```

iii) V Transform - This PBLOCK accepts a square Boolean matrix addressed by A0 and stored as described above, and performs the V transform upon it. The matrix created is in the same form as the original, and is addressed by register A1 at the completion of the procedure. It should be noted that the procedure below performs the V transform on any square Boolean matrix, while in the case of transforming $A\ x(A^T)$, further simplification of the program can be performed due to the fact that $A\ x(A^T)$ is symmetrical. In particular, the statements labelled X to Y inclusive may be replaced by the statement

```
                OR        B3:(I2,I3),0(A0)<I2>:(B,I2,I3)
```

In full, the module is as follows:

```
MBVTR           PBLOCK
                PSH I0,I1,I2,I3,B2,B3
* DETERMINE SIZE OF MATRIX - I0 BY I0
                L         B3,0(A0)<0>
                SIZE      B3,I0
* SET UP LOOP COUNTERS I0 = N-1, I1 = N
                LR        I1,I0
                S         I0,ONE
* INITIALIZE THE ELIMINATION STRING
                XOR       B3,B3
* THE STATEMENTS FOLLOWING DOWN TO "OUT" GENERATE A
* BIT FIELD WHICH MARKS THE COLUMNS TO BE ELIMINATED
                ZE        I2
ELIM            LR        I3,I1
                SR        I3,I2
X               ZE        B2
                LOOP      I3,I2,I0,1
                BUR       B2,0(A0)<I3>:(B,I2,1)
                LEND
Y               ORR       B3:(I2,I3),B2
                L         B3:(I2,1),=B'0'
SHELIM          A         I2,ONE
                CR        I2,I0
```

```
              BC        2,OUT
              LTR       B3:(I2,1),B3:(I2,1)
              BC        8,ELIM
              B         SHELIM
* THIS SECTION BUILDS THE COMPRESSED MATRIX, USING
* THE ORIGINAL MATRIX A, AND THE ELIMINATION STRING
OUT           NR        B3
              CNTI      B3,B'1',I3
              DBLOCK    D,T(I1,B,I3)
              LOOP      I2,0,I0,1
              L         B2,0(A0)<I2>
              CMP       B2,B3
              ST        B2,D<I2>
              LEND
              PNT       A1,D                LOCATE D
              POP       I0,I1,I2,I3,B2,B3   CLEAN UP
              REV                           RETURN
ONE           DFA       I'1'
              END
```

iv) W Transform - This PBLOCK accepts a square Boolean matrix A pointed to by register A0, and forms a second matrix BW = $\bar{E} \cap A$, which is effectively A with its diagonal elements all zero.

```
MBWTR         PBLOCK
              PSH       I0,I1,B3
* DETERMINE SIZE OF THE MATRIX - I0 BY I0
              LBL       I0,0(A0)
              DBLOCK    BW,T(I0,B,I0)
              S         I0,ONE
              LOOP      I1,0,I0,1
              L         B3,0(A0)<I1>
              L         B3:(I1,1),=B'0'
              ST        B3,BW<I1>
              LEND
              PNT       A1,BW
              POP       I0,I1,B3
              REV
ONE           DFA       I'1'
              END
```

The following PBLOCK calls on the modules given above to perform the overall operation of decomposing a graph into its strongly connected components. Given the incidence

matrix of the graph, pointed to by register A0, it returns the decomposition matrix D, and the incidence matrix of the reduced graph, A^R, pointed to by registers A1 and A2 respectively.

```
DECOMP      PBLOCK
            PSH       A0         REMEMBER A
            E         MBCLOSE    GET TRANSITIVE CLOSURE
            LR        A0,A1
            E         MBTRANS    GET ITS TRANSPOSE
            E         MBPROD     MULTIPLY THEM
            XBL       0(A0)      DROP THE CLOSURE
            XBL       0(A1)      AND ITS TRANSPOSE
            LR        A0,A2
            E         MBVTR      FORM D
            XBL       0(A0)      DISPOSE OF PRODUCT
            PSH       A1         REMEMBER D
            LR        A0,A1

            E         MBTRANS    GET D TRANSPOSE
            POP       A0         RETRIEVE ADDRESS OF
            PSH       A0         A THEN REVERSE ARGUMENTS
            SWR       A0,A1      FOR MBPROD
            E         MBPROD     FORM D TRANS X A
            XBL       0(A0)      DROP D TRANS
            LR        A0,A2
            POP       A1         RETRIEVE ADDRESS OF
            PSH       A1         D
            E         MBPROD     FORM (D TR X A) X D
            XBL       0(A0)      DROP (D TR X A)
            LR        A0,A2
            E         MBWTR      FORM AR
            XBL       0(A0)      DROP (D TR X A) X D
            LR        A2,A1
            POP       A0,A1
            REV
            END
```

To program the above example in a language which did not have the variable length operations and bit addressing of ALAS would probably require more data storage and more intermediate programming concerned with data access. In particular, the lack of a vector and subscripting facility in L^6 makes creation of and accessing a Boolean matrix of

indeterminant size more complex than it is in ALAS. To illustrate, consider that locating the ith word in an L^6 block requires either the addition of the quantity i to a bug containing the address of the block, or the definition of a relative field of displacement i, while in ALAS, the location of the ith element of an ALAS vector involves the simple use of the subscript operation. Consider also that the elements in an ALAS vector are not necessarily word length. The amount of L^6 programming effort increases again if one attempts to duplicate the storage appearance of the ALAS vector - in that the ith element may not lie in the ith word, it may lie in the jth or j and $j+1^{th}$ words, where $i > j = f(i, l_e, l_w)$; l_e = element length, and l_w = word length.

CHAPTER VII

IMPLEMENTATION

7.1 INTRODUCTION

At the beginning of this thesis, we pointed out that attempts at designing general purpose software for interactive graphics have not been outstandingly successful. Rather than simply using the available hardware and software tools to design yet another software support system, we have chosen to return to first principles in developing an essentially new approach to the design of graphics software.

The operations required for interactive graphics software involve a great deal of string processing and model building which fall under the general heading of data structure manipulation. The crux of the problem is that machines are not yet well designed for this type of work. The author has therefore designed a low level data structure system which, in his opinion, is very suitable for the purpose.

In the long term, the author forsees the gradual evolution of machines designed more explicitly for data structure manipulation. In the meantime, systems such as the one described in this thesis should be implemented for

experimentation, using those hardware techniques at present available. Possible methods of implementation are discussed in the remainder of this chapter.

7.2 THE HYBRID SOFTWARE IMPLEMENTATION

The author considers that the most convenient method for implementing ALAS is a hybrid technique, part assembly and part interpretation. The software is split into two components: a preprocessing component, which includes the assembler and loader, and an in-core component, which includes the storage allocator, the interpreter for extended operations, and the address processor.

The assembler translates ALAS programs into a mixture of machine code and calls to the in-core component. The loader attaches the required external routines to the ALAS program, and creates a core image of the static part of the program. It then requests storage for the program, loads it, and initiates execution. As the ALAS program is partly in-line code and partly subroutine calls, the author considers the translation process a hybrid of assembly and interpretation.

The decision on which operations are to be interpreted and which operations translated as in-line code depends upon the characteristics of the machine.

As an example, consider the evaluation of operand addresses. Each evaluation may be handled by generated machine code, or through a call to a special routine. The choice would be made by weighing the possible saving in space using a call against the saving in time using in-line code. For an implementation on the 360, the author would choose:

Addressing Operation	Technique
Base Address Generation	In-Line Code
Absolute Expression Modifier	In-Line Code
Index Modifier	In-Line Code
Indirect Modifier	Call
Relative Field Modifier	Call
Subscript Modifier	Call
Pattern Reference	Call
Subfield Specifier Bits	Call
Subfield Specifier Characters	In-Line Code

In this case, the address string

AT@B_<I3>+2

where AT refers to a dynamic block,

B is a statically defined relative field,

I3 is an index register, and

2 is immediate data,

would translate into the following:

a) In-line code to get the address of block AT,

b) A call to the relative field modifier routine,

c) A call to get the indirect address,

d) A call to the subscripting routine, and

e) In-line code to add 2.

An example of translation into 360 Assembler would be as follows:

```
* BASE ADDRESS GENERATION - PART A)
        LM        4,5,AAT          PICK UP ADDRESS OF AT
        LTR       4,4              IS IT VALID?
        BC        13,R1ME3         IF NOT, ERROR TIME
* RELATIVE FIELD CALL - PART B)
        LA        6,B              GET ADDRESS OF B
        BAL       14,S1F           CALL FIELD PROCESSOR
* INDIRECT ADDRESSING COMPOUND - PART C)
        LA        6,TOLFT          GET ADDRESS OF AREA 0
        BAL       14,MOVER         MOVE FIELD INTO AREA 0
        BAL       14,CONAD         CHANGE AREA 0 TO ADDRESS
        L         4,TORGT-4        ADDRESS TO REG 4
        LA        5,32             DEFAULT LENGTH
* SUBSCRIPTOR CALL - PART D)
        L         6,I3             GET SUBSCRIPT INTO 6
        BAL       14,SOS           CALL SUBSCRIPTOR
* ABSOLUTE EXPRESSION CODE - PART E)
        LA        6,2              EVALUATE EXPRESSION
        AR        4,6              AND ADD IT
```

Notes: i) AAT contains the address of the block currently associated with AT

ii) ALAS registers are in core and associated with names identical to their specifiers.

iii) ALAS address words are one full word, used as follows:

Bit	Contents
0	on if address invalid
1	on if address word aligned
2	on if address byte aligned
3-4	unused, always off
5-7	bit remainder of address
8-31	360 (byte) address

A similar process may be applied to the operation code set. Machines with more powerful hardware will of course have more in-line code and fewer subroutine calls. The hybrid technique has the advantage that it is realizable with present machinery.

7.3 HARDWARE IMPLEMENTATIONS

Since ALAS is at the assembler level, its design may be said to reflect the architecture of a hypothetical machine. All ALAS operations are realizable in modern hardware design, and storage management could be handled with techniques already used by some assemblers. The major obstacle is the high expected cost of building an ALAS processor. The author feels that a period of experimentation with an assembler/interpreter implementation is necessary.

A second stage in experimentation might make use of microprogramming. This technique involves a processor which has a number of simple, high speed operations (micro-operations) which are combined into programs (microprograms) retained usually (though not always) in read-only storage, and executed as unit instructions. The selection of the appropriate microprogram is based on the "instruction code" fetched from memory. Thus, when the machine appears to be executing a machine code instruction, it is actually

performing a program of micro-code instructions. This technique is used commercially in most machines in the IBM 36C series.

The author feels that implementation of ALAS on a machine with variable microprogram storage would be worthwhile. A microprogrammed ALAS machine would be faster than a system using the hybrid software technique. However, micro-computers themselves are not well designed for handling many of the functions required in ALAS, in particular the use of variable length operands. One can foresee eventually the design of micro-computers oriented specifically towards the implementation of this type of system.

CHAPTER VIII

A GRAPHICAL INTERFACE IN ALAS

8.1 INTRODUCTION

ALAS itself does not have any particular graphical facilities. What it does have are standard instructions for sending strings of data to any output device (including a graphics terminal), and accepting strings from any input device. However, ALAS has been designed to be particularly suitable for building a graphics package to aid the programmer.

The following package (PRIG: Package for Remote Interactive Graphics) is intended for implementation in ALAS for the IBM 360/67 - CDC GRID System at the University of Alberta, to be made available as part of the ALAS library. As ALAS is at the assembler level, the characteristics of the terminal will affect the interface. In particular, the design of PRIG is affected by the fact that the display terminal controller is a computer, and is therefore capable of performing some processing of data transmitted to and from the ALAS program in the 360.

Using PRIG, output to the display terminal is brought about as follows:

a) The ALAS program uses PRIG routines to create a

picture description.

b) PRIG processes the picture description to create appropriate code for the terminal, forwards the code to the terminal, and initiates display of the picture described.

Using PRIG, input from the display terminal is organized into groups of user actions, which are called 'messages'.

PRIG is used to provide functions of three types:

a) Picture Building

b) Display Control

c) Input Processing

A brief description of PRIG follows in the next three subsections. The techniques used to describe ALAS (Chapter 5) will be used in this chapter. An example, a program which provides a simple sketching facility, is given in Appendix A.

8.2 PICTURE BUILDING

The PRIG picture model is a plexlike structure created from blocks called nodes which contain drawing information, status control information, and references to other nodes.

The PRIG commands for picture modelling fall into four categories, dealing with:

a) Structure,

b) Drawing Content,

c) Status Sensing and Control Content, and

d) Editing.

8.2.1 Structure Commands

Function: The Structure Commands (Table 8.1) are concerned with the creation and interrelationships of nodes.

Notes: i) NODE and ENDNODE delimit a node definition. NODE initiates node definition mode, and associates its operand, as a name, with the node created by the remainder of the definition. ENDNODE terminates node definition mode. The body of the node definition consists of those PRIG node content commands which occur between a NODE command and an ENDNODE command. Use of these node content commands inserts code into the node.

ii) SHOW inserts code (into the current node definition) which shows another node, named by its first operand. This code includes a pointer to the shown node (the 'target' node), and control parameters given by the remainder of the SHOW command operand list. Inserting a reference to a node via a SHOW command is considered to subjugate the target node to the current node for the instance of the SHOW command. The hierarchical relationships so defined can be used in editing picture descriptions and appraising input strings. Examples of these uses are to be found in Appendix A. The second and

Table 8.1 Structure Commands

Command Name	Opcode	Operand List
Node	NODE	[symbcl]
Endnode	ENDNODE	
Show	SHOW	[symbcl],S0,S1,S2,S3,S4,S5,S6, S7
Repeat Node	RNODE	[symbol],[symbol]
Delete Node	XNODE	[symbol]

third operands indicate the position of the origin of the target node (relative, of course to the origin of the current node). The fourth and fifth operands are values to be used in scaling the target node. The sixth and seventh operands indicate the time range during which the target node is to be displayed. The eighth operand is a bit operand indicating whether or not the target node is to be blinking in this instance. The ninth operand is a bit operand indicating whether the target node is to be blanked in this instance. Any or all of the control parameters (operands two to nine) may be omitted, but the commas must be included. The omission of a parameter indicates that a default action is desired. The defaults are:

a) Operand Two - No X displacement

b) Operand Three - No Y displacement

c) Operand Four - No X scaling

d) Operand Five - No Y scaling

e) Cperand Six - Start Time = 0

f) Operand Seven - Stop Time = 2047

g) Operand Eight - Not Blinking (0)

h) Operand Nine - Not Blanked (0)

iii) The Repeat Node command (RNODE) creates a copy of the node associated with its first operand, and associates the copy with its second operand.

iv) Delete Node (XNODE) causes the deletion of the node associated with its operand.

8.2.2 Drawing Content Commands

Function: The Drawing Content commands (Table 8.2) insert drawing code into the current node definition.

Notes: i) The Vector commands (VCT, VCTC) insert code which results in the drawing of a series of connected straight line segments. The first operand indicates the number of line segments to be drawn. The second and third operands are ALAS vectors which contain in corresponding positions the x and y coordinates (respectively) of the series of points used consecutively as end points of the line segments. VCT draws the series of connected straight line segments starting at the first point through the points consecutively to the last point. VCTC draws the same series as VCT, and an additional line from the last coordinate pair used by the preceding drawing content command to the first point in the series. The fourth operand is a bit operand

Table 8.2 Drawing Content Commands

Command Name	Opcode	Operand List
Vector	VCT	S0,S1,S2,S3
Vector Continued	VCTC	S0,S1,S2,S3
Point	POINT	S0,S1,S2
Symbol	SYMBOL	S0,S1,S2,S3
Text	TEXT	S0,S1,S2,S3,S4,S5
Text Continued	TEXTC	S0,S1,S2,S3
Text Register	TEXTR	C,S0,S1,S2,S3
Text Register Continued	TEXTRC	C,S0,S1

indicating whether or not the line is to be dashed.

ii) The POINT command inserts code which shows a series of points. The first operand indicates the number of points to be shown, and the second and third operands indicate their positions, as in the case of the Vector commands.

iii) The SYMBOL command operates in the same manner as the POINT command, except that instead of a point, the character addressed by the fourth operand is shown at the points indicated. The first, second and third operands are identical in type and function to those of the POINT command.

iv) The Text commands (TXT, TXTC, TXTR, TXTRC) insert code which shows strings of characters. In the case of TXT and TXTC, the strings are in memory, and their first operands indicate the lengths of the strings, and their second operands address the strings. TXTR and TXTRC obtain their strings from character registers, specified as their first operands. The third and fourth operands of the TXT command and the second and third operands of the TXTR command indicate the x and y coordinates of the start position of the string. TXTC and TXTRC start their strings at the next character position following the last coordinate pair used by the preceding drawing content command. The last two operands of all four instructions are identical in function and style. The second last operand is a bit operand indicating whether the characters are to be large or small. The last operand is a bit operand indicating whether or not the character string is to be horizontal (left to right), or vertical (bottom to top). An example of the use of both TEXT and TEXTR is to be found in Appendix A.

8.2.3 Status Sensing and Control Content Commands

Function: These nine commands (Table 8.3) are concerned with the insertion of code which makes the picture sensitive to certain environmental considerations, such as the value of the timer, the key code of the last function key depressed, light pen interrupts, and so on.

Table 8.3 Status Sensing and Control Content Commands

Command Name	Opcode	Operand List
Identify	ID	S0
Disable Interrupts	DISINT	S0,S1
Enable Interrupts	ENINT	S0,S1
Branch Timer Low	TMBL	S0,[symbol]
Branch Timer Not Low	TMBNL	S0,[symbol]
Branch Function Key Latch Equal	FKBE	S0,[symbol]
Branch Function Key Latch Not Equal	FKBNE	S0,[symbol]
Timer Control	TMC	S0,S1
Request Next Frame	NXFR	

Notes: i) ID - As mentioned in connection with the SHOW command, there is a hierarchy in a PRIG picture structure. For the purposes of discussion, a node body will be considered one lateral level; i. e. all code within a given node is at a common lateral level. SHOW commands define vertical connections. A node containing a SHOW command is considered to be, in that instance, at a level above the node shown by that command. Thus, PRIG picture models may be thought of as having levels. The levels are considered in terms of the number of SHOW commands between the top level of the picture and the node under consideration.

Level is a structural characteristic, not a characteristic of a node, as a node may be utilized at many different levels within one picture model. Every node has an identity character. This character can be changed within the node through the use of the ID command. The default identity character is the base character.

Identity characters are used by the PRIG picture generator in the following way. During display, the display controller software maintains an ID stack. Whenever a SHOW command code is encountered, the ID stack is pushed down, and the top of the stack is set to the identity character of the node shown. Whenever the end of a node is encountered, the ID stack is popped up. Whenever an ID command code is encountered, the contents of the top element of the ID stack are changed to the character specified by the command. This is done to provide a basis for the identification of interrupts. When an interrupt occurs, such as the light pen pick of a displayed entity, the current ID stack is copied as part of the action description.

The identity character is also used in the editing process. The locating of SHOW command code for the purposes of alteration can be directed through the use of an ID chain. The edit functions identically to the display controller software in handling identity characters. For example, the ID chain associated with a light pen pick will,

when utilized by the editor, locate the code generating the display of the element picked.

ii) The Interrupt Control commands (DISINT, ENINT) insert code which respectively disables or enables interrupts. The operands are two binary values, the first of which is associated with light pen interrupts, and the second with alphanumeric keyboard interrupts. If an operand is true (1), the enable/disable status is altered for the corresponding interrupt type; otherwise, there is no effect.

iii) The Timer Branch commands (TMBL, TMBNL) insert code into the current node which will alter the order of execution of code within the node on the following basis. The GRID display terminal displays a picture by looping through display code, as the terminal does not use a storage CRT. To prevent possible burning of the phosphor in the CRT, the loop must not be executed more than fifty times a second. The GRID instruction set includes an instruction which, when present in the display loop, ensures that the maximum refresh rate (i.e. frequency cf loop execution) is not exceeded. This instruction effectively provides a time base, as long as the display loop takes no longer than twenty milliseconds to execute. PRIG's timer is a counter updated by the display controller software each time the display loop is executed. The timer can be zeroed at the initiation of a frame (Section 8.3), and may be altered by code generated by the Timer Control command (see below).

The timer has two modes of operation, _fine_ and _coarse_. In _fine_ mode, one timer unit corresponds to one display loop execution (twenty milliseconds under optimal conditions). In _coarse_ mode, one timer unit corresponds to sixty-four executions of the display loop (1.28 seconds under optimal conditions). The Timer Branch commands insert code which checks their first operands against the timer, and branches if the timer is either low (TMBL), or not low (TMBNL). The addresses to which branches are taken are generated by the picture generator, and are specified as [symbol]s which have occurred in the label fields of PRIG node content commands within the same node definition as the branches. The timer commands provide the user with time dependent display capabilities.

iv) The Function Key Branch commands (FKBE, FKBNE) insert code which will alter the order of execution of code within a node in the following manner. On the keyboard of the GRID terminal, there are latching keys called Status keys, and special keys called Function keys (Section 8.5). Pressing Status keys causes them to latch; if the key were on (in) it switches off (out), and vise versa. Status keys do not cause interrupts; however, on every interrupt they are read. Depressing Function keys causes interrupts. Except when none of the Status keys are depressed (Status 0), Function key interrupts cause the display controller software to update a control word which contains a composite of the Status and the Function key number, referred to as

the Status/Function Key Latch. The code inserted by the Function Key Branch commands checks the Status/Function key composites which are their first operands against the Status/Function Key Latch. FKBE branches if they are equal; FKBNE branches if they are not equal. The addresses for branches are determined from their second operands as in the case of the Timer Branch commands.

v) The Timer Control command (TMC) inserts code to alter the operation of the timer. It has two binary operands. The first operand affects the mode of operation of the timer. If it is true (1), the timer is altered to, or remains in, coarse mode. If it is false (0), the timer is altered to, or remains in, fine mode. If it is omitted, the mode of operation of the timer is unaltered. The second operand, if true (1), requests that the timer be restarted.

vi) The Next Frame command (NXFR) inserts code that requests the display controller software to display the next frame.

8.2.4 Editing Commands

Function: PRIG picture editing allows for the following operations:

a) Deletion of nodes (see XNODE, above),

b) Chaining of nodes, and

c) Altering the parameters of SHOW commands.

The commands are shown in Table 8.4.

Table 8.4 Editing Commands

Command Name	Opcode	Operand List
Chain	CHAIN	[symbol],[symbol]
Locate Show	LOSH	[symbol],D0,[symbol],S1,S2
Locate Next Show	NXSH	
Query Origin	QOR	{F,I},{F,I}
Query Scale	QSC	{F,I},{F,I}
Query Timer Limits	QTI	{F,I},{F,I}
Query Blink	QBLI	B
Query Blank	QBLA	B
Set Node	SETND	[symbol]
Set Origin	SETOR	{F,I},{F,I}
Set Scale	SETSC	{F,I},{F,I}
Set Timer Limits	SETTI	{F,I},{F,I}
Set Blinking On	BLINK	
Set Blinking Off	UNBLINK	
Set Blanking On	BLANK	
Set Blanking Off	UNBLANK	
Copy Node	CNODE	[symbcl]

Notes: i) CHAIN attaches a copy of the body of the node named by its second operand to the node named by its first operand.

ii) The Locate Show commands (LOSH, NXSH) scan the picture model for a SHOW command which satisfies a particular set of criteria. LOSH initiates the scan. Its first operand identifies the top node of the picture (or picture fragment) which is to be scanned. Its second operand indicates the combination of criteria to be used to locate the SHOW command code. The third, fourth and fifth operands of the LOSH command specify the scanning criteria. If a particular criterion is not required by the combination in use, the corresponding operand may be omitted. NXSH continues the scan, using the same criteria, and starting at a point in the picture model just after the SHOW located by the previous LOSH or NXSH command. If a SHOW satisfying the criteria is located, condition code zero is set. Otherwise, condition code three is set. The three criteria that are used to make up the criteria combinations are

a) A Node Name - operand three

b) A Level Indicator - operand four

c) An ID Chain - operand five

The seven combinations and their interpretations are as follows:

a) Operand Two : 1

Operand Active : five

Effect : The SHOW command sought is in code with an ID chain equal to operand five.

b) Operand Two : 2

Operand Active : four

Effect : The SHOW command sought is on the level indicated.

c) Operand Two : 3

Operands Active : four,five

Effect : The SHOW command sought is on the level indicated. It must also either show a node (either directly or indirectly) which contains code with an ID chain equal to operand five, have an ID chain equal to operand five, or have an ID chain whose leftmost portion is equal to operand five.

d) Operand Two : 4

Operand Active : three

Effect : The SHOW command sought directly shows the node named by operand three.

e) Operand Two : 5

Operands Active : three,five

Effect : The SHOW command sought directly shows the node named by operand three. It must also either show a node (either directly or indirectly) which contains code with an ID chain equal to operand five, have an ID chain equal to operand five, or have an ID chain whose leftmost portion is equal to operand five.

f) Operand Two : 6

Operands Active : three,four

Effect : The SHOW command sought directly shows the node named by operand three on the level indicated by operand four.

g) Operand Two : 7

Operands Active : three, four, five

Effect : The SHOW command sought directly shows the node named by operand three at the level indicated by operand four. It must also either show a node (either directly or indirectly) which contains code with an ID chain equal to operand five, have an ID chain equal to operand five, or have an ID chain whose leftmost portion is equal to operand five.

iii) The Query commands (QOB, QSC, QTL, QTU, QBLI, QBLA) retrieve the current control parameters associated with the SHOW command most recently found by the Locate Show commands. The mnemonics and the parameters accessed are as in Table 8.5.

iv) The Set commands (SETND, SETOR,SETSC, SETTI, BLINK, UNBLINK, BLANK, UNBLANK) alter the control parameters associated with the SHOW command most recently found by the Locate Show commands. The mnemonics and the parameters altered are as in Table 8.6.

v) Omission of parameters from the Query and Set commands is permitted, and the omitted operands are either not accessed (Query commands) or not altered (Set commands).

vi) The Copy Node command (CNODE) generates a copy of the node containing the SHOW command most recently found by the Locate Show commands. This copy replaces the instance of the node that contains the selected SHOW command. The copy may be named (optional) by the operand of

Table 8.5 Parameters Retrieved by Query Commands

Mnemonic	Parameters Accessed
QOR	The displacement of the origin of the target node relative to the origin of the node containing the SHOW.
QSC	The scale parameters applied by the SHOW
QTI	The timer limits applied by the SHOW
QBLI	The blink control value (true (1) if blinking)
QBLA	The blank control value (true (1) if blanked)

the CNCDE command. Ensuing modifications to the selected SHOW command affect the copy, which now in effect contains the selected SHOW command. (Note that, without copying, alterations to a node in the form of alterations to SHOW commands contained within that node affect all displayed instances of that node, as there is nominally only one copy of any node.)

8.3 DISPLAY CONTROL

Structures created by the picture building commands are processed by the PRIG picture generator to prepare display controller code. This code is forwarded to the display

Table 8.6 Parameters Altered by Set Commands

Mnemonic	Parameters Altered
SETND	The node shown becomes that named by the SETND operand
SETOR	The displacement of the target node origin relative to the origin of the node containing the SHOW is set to the SETOR operands
SETSC	The scale parameters applied by the SHOW are set to the SETSC operands
SETTI	The timer limits applied by the SHOW are set to the SETTI operands
BLINK	The blink control value is set to true (blinking)
UNBLINK	The blink control value is set to false (not blinking)
BLANK	The blank control value is set to true (blanked)
UNBLANK	The blank control value is set to false (unblanked)

controller, which then creates displays on the CRT. The Display Control commands (Table 8.7) provide program control over these processes.

Notes: i) The Initialize Display command (IDISP) initiates communication with the display controller, initializing the display controller software. If specified, the operand

Table 8.7 Display Control Commands

Command Name	Opcode	Operand List
Initialize Display	IDISP	S0
Reinitialize Display	REDISP	S0
Close Display	XDISP	
Frame	FRAME	[symbol],{F,I}
Add Node	ADNODE	[symbol],{F,I}
Send	SEND	{F,I},{F,I}
Frame Status	FSTST	B

indicates the number of frames which can exist simultaneously in the display controller storage. A frame is one complete display loop and, therefore, only one frame can be displayed at a time. The capability of having more than one frame in the display controller storage at any one time is made available to allow the overlapping of frame transfer time and user interaction with the current displayed image. The frame being processed to create the current displayed image will be referred to as the active frame, the process of initiating the display of a frame being activation of the frame. The default is one frame.

ii) The Reinitialize Display command (REDISP) resets the display controller software, discarding any frames in display controller storage, and initializing the

display controller software to provide for the number of frames indicated by the operand, as in the case of IDISP above. In the case of REDISP, however, the default number of frames is the number associated with the previous IDISP or REDISP command.

iii) The Close Display command (XDISP) terminates communication with the terminal.

iv) The Frame command (FRAME) processes the picture structure whose initial ('topmost') node is named by its first operand. It creates the display controller code which will display the picture modelled. The second operand is optional, and indicates a frame index, a numerical identifier used in the case of the multiple framing process mentioned above. The PRIG picture generator maintains a number of staging areas equal to the number of frames which can exist simultaneously in the display controller storage. The second operand of FRAME specifies which staging area the display controller code is to occupy. If the frame index is not specified, the next available frame staging area is utilized. For the purposes of this decision, the frame staging area which is occupied by the active frame is considered available, however, it is always used only if no other frame staging areas are available.

v) ADNODE processes the picture structure whose initial ('topmost') node is named by its first operand. The code created is added to a previously generated frame. If the second operand is specified, it indicates the frame

index of the frame to which the code is to be attached. Otherwise, the code is attached to the most recently created frame. (Note - This code is not automatically forwarded to the terminal, even if the augmented frame is active.)

vi) The Send command (SEND) controls output to the terminal controller. Its first operand is a control value, expressed as a decimal integer in the range 0 to 31 inclusive. The control value is interpreted as five control bits, whose controls are as follows:

Table 8.8 Control Value Interpretation

Bit	Control
0	Send frame
1	Send added node
2	Activate Frame
3	Reset timer
4	Set timer coarse mode

The bit indices in the table are from left to right in the control value. For example, the control value 22 ((Bit 0 = 16) + (Bit 2 = 4) + (Bit 3 = 2) = 22) indicates that a frame is to be sent and activated, with the timer reset to 0 and operating in fine mode. The second operand of SEND is an optional frame index specification. If a frame index is required by the command, but not supplied, the index of the

most recently created frame is used.

vii) The Frame Status command (FSTST) retrieves information concerning the status of frames. The information is returned in groups of four bits, one group per frame to a maximum of eight frames. The groups are ordered with respect to frame indices, and combined into a field which is loaded into the operand (a bit register). The four bits are interpreted as follows:

a) Bit 0 - Frame Possible - If a staging area for this frame exists, this bit is on.

b) Bit 1 - Frame Active - If this frame is currently being displayed, this bit is on.

c) Bit 2 - Frame Sent - If the most recently created copy of this frame has been sent to the display controller, this bit is on.

d) Bit 3 - Frame Amended - If the frame in the staging area and the frame in the display controller storage differ only in that the frame in the staging area is an augmented version of the frame in the display controller storage, this bit is on.

8.4 INPUT HANDLING

A message (input record from the terminal) is returned to the ALAS program as a structure. The facilities of PRIG include an 'await input' command, and commands for accessing and decoding the structure. Input may be decoded through

the use of regular ALAS instructions as well. Input from the terminal that occurs prior to the program encountering an 'await input' command is held until such a command is encountered.

Prior to considering the actual commands, a brief discourse on the input structure is in order.

The input message is structured as an ALAS vector, with pointers within it which point to varying length portions of two other blocks. Each field of the vector has five subfields:

i) Action Identifier - This subfield contains a numerical value which indicates the type of message component this vector element describes. Table 8.9 describes the actions and their corresponding action identifiers.

ii) Timer/Active Frame/Status Composite - This subfield contains the setting of the Status keys, the active frame index, and the value and mode of operation of the timer at the time of the interrupt (the first interrupt in the case of a multiple interrupt component) which created this message component.

iii) X-Y Composite - This subfield contains the values of X and Y to be associated with this message component. The significance of these values depends upon the type of the message component. Table 8.10 indicates the significance of the X and Y values in terms of the Action

Table 8.9 Action Identifiers and Their Actions

Action ID	Action
1	Light pen hit
2	Alpha key hit
3	Function key hit
4	Special key hit
5	Alpha key string
6	Point string
7	Vector
8	New Frame interaction
9	Timer hit maximum
10	End of message

Table 8.10 Significance of X-Y Composite

Action ID	Significance of X-Y Composite
1	contents of X register and Y register of the display controller at the time of the interrupt, which effectively indicate the position of the element hit
2,3,4	the position of the cursor at the time of interrupt
5	the cursor position at the beginning of the string
6	the position of the first point
7	starting position of the vector
8,9,10	no significance

Identifiers.

iv) ID - This subfield contains the length of the ID chain, and if that value is less than three, the ID chain itself. As in the case of the Timer/Active Frame/Status composite, if the message component is a multiple interrupt type, the ID chain is the ID chain that was current at the time of the first interrupt of the series. If the length of the ID chain is greater than three, this subfield contains a pointer into the ID heap, a block which contains ID chains.

v) Contents Composite - This subfield contains variable information, dependent on message component type as well as interrupt information, as in Table 8.11.

Table 8.11 Contents Composite Contents vs Component Type

Action ID	Contents
1,9,10	Not relevant
2,3,4	Key code
5	Count of characters and pointer to character string
6,7	Count of, and pointer to a contiguous group of X-Y composites
8	A value which indicate what interactions have taken place concerning frame status

Character strings, and vectors of X-Y composites are held in a block called the Data Heap. ID chains longer than three characters are held in a block called the ID Heap. The PRIG input handling commands are listed in Table 8.12, and detailed in the notes below:

Notes: i) Await Graphical Input (AWGINP) suspends ALAS program operation until the next message from the terminal arrives.

ii) The Accessing commands (ACTION, ASTFRTI, ASTAT, AFRAM, ATIME, AXY, AX, AY, AID, ASTR, AKCD, AVECT, ACONT) each access portions of the current input message. The first operand is an integer register which contains the index of the message component being examined, and the remainder of the operand list specifies the areas to which the desired information is transferred. Table 8.13 details the information accessed.

8.5 MESSAGE COMPONENT GENERATION

This section describes the user actions required to generate the message components discussed in Section 8.4. In the course of this discussion, the consolidation functions of the display controller software become evident.

Message components may be the following:

a) Light pen hit

Table 8.12 Input Handling Commands

Command Name	Opcode	Operand List
Await Graphical Input	AWGINP	
Access Action ID	ACTION	I,{F,I}
Access Status/Frame /Time Composite	ASTFRTI	I,{F,I}
Access Status	ASTAT	I,{F,I}
Access Frame Index	AFRAM	I,{F,I}
Access Timer Value	ATIME	I,{F,I}
Access X-Y Composite	AXY	I,{F,I}
Access X Value	AX	I,{F,I}
Access Y Value	AY	I,{F,I}
Access ID Chain	AID	I,C
Access Character String	ASTR	I,C
Access Key Code	AKCD	I,{B,C,F,I}
Access Vector Addresses	AVECT	I,{F,I},A,A
Access Contents Subfield	ACONT	I,{F,I},A

Table 8.13 Information Accessed by Access Commands

Mnemonic	Information Accessed
ACTION	The Action Identifier as in Table 8.9
ASTFRTI	The Timer/Active Frame/Status subfield
ASTAT	The Status
AFRAM	The active frame index
ATIME	The timer value
AXY	The X-Y Composite
AX	The X value of the X-Y Composite subfield
AY	The Y value of the X-Y Composite subfield
AID	The ID Chain
ASTR	The character string associated with the component
AKCD	The key code associated with the component
AVECT	The length of the list of points, and the addresses of the vectors of X values and Y values
ACONT	The length parameter and address parameter from the contents subfield

b) Alpha key hit

c) Function key hit

d) Special key hit

e) Alpha key string

f) Point string

g) Vector

h) New frame interaction

i) Timer hit maximum

j) End of message

Allowable user actions at the terminal are (in this case) the following:

a) Depressing the keys on the terminal keyboard (see Figure 8.1), and

b) Selecting displayed entities with the light pen.

For a more elaborate description of the display terminal, the reader is referred to the GRID Display Subsystem Hardware Reference Manual (Control Data Corporation, 1968).

PRIG display controller software considers the keyboard as divided into four classes:

a) Status keys, which are the four unlabeled keys at the top,

b) Function keys, which are those labelled F1, F2, ..., F9, F0, and INTER,

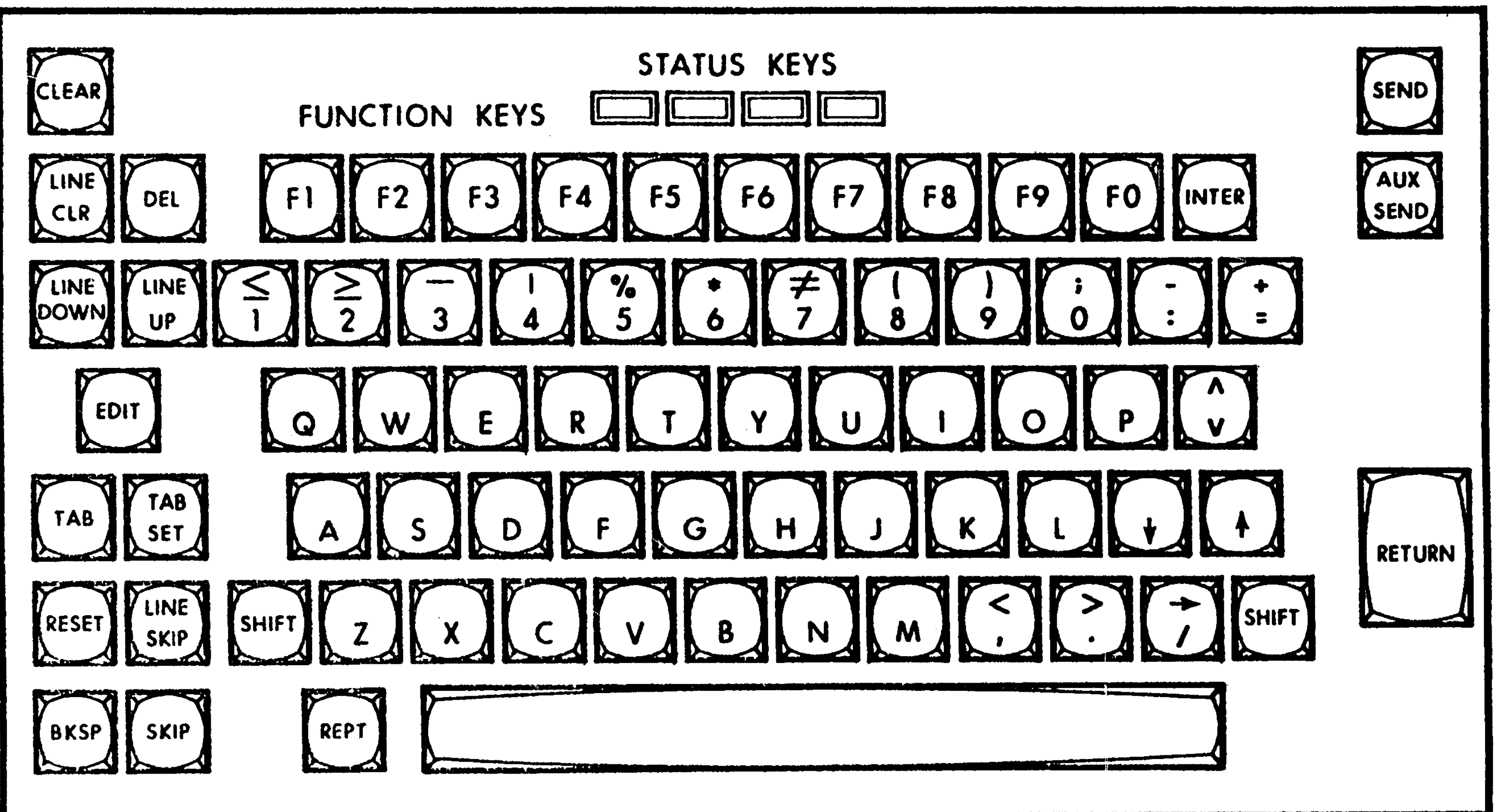


Figure 8.1 GRID Keyboard

c) Special keys, which are those labelled SEND, AUX SEND, RETURN, REPT, SHIFT, CLEAR, LINE, CLR, DEL, LINE DOWN, LINE UP, EDIT, TAB, TABSET, RESET, LINESKIP, BKSP, and SKIP, and

d) Alphanumeric keys, which are the remainder of the keys on the board.

The status keys do not generate an interrupt when activated. Instead, they are read by the display controller software whenever an interrupt occurs.

If the status keys are all disengaged, the status is zero, and the function keys have particular functions which control the state of the terminal, and entries into the message buffer. If any or all of the status keys are depressed, depressing a function key generates an message component of the type 'function key hit'. Status zero function key hits will be discussed later.

Some of the special keys have particular functions (to be discussed later), but usually they generate 'special key hit's when depressed.

The alphanumeric keys generate 'alpha key hit's when depressed. If no other manual interrupts intervene between 'alpha key hit's, they are concatenated to form an 'alpha key string'. There is a position indicator displayed by the

display controller software. This position indicator, hereinafter called the cursor, is utilized by the display controller software to indicate the position which the display controller last acknowledged. An entered 'alpha key string' is displayed starting at the current cursor position. As the string is typed in, the cursor moves, indicating the next position in which a character can be typed. A second cursor, called the echo cursor, remains at the beginning of the displayed 'alpha key string'.

The above section describes the generation of message components b through e. Message components f through j are created by a composite of software and user actions. As they do to some extent depend on the status zero function keys, let us now consider the actions caused by depressing them.

Function key 1 is a position indicating key. It determines the light pen position by displaying a complete scan of the screen. The cursor is moved to the position at which the light pen is pointed. The echo cursor is left at the last cursor position acknowledged at a status 0 function key 4 hit, or an alpha key hit, or an alpha key string's first character.

Function key 2 is also a position indicating key. It determines the light pen position by tracking the light pen,

that is, by displaying a small figure which is detected by the light pen and moved across the screen by 'dragging' it with the light pen. The cursor is moved with the tracking figure, and unless the operator is drawing a vector, the echo cursor is not moved. This key acts as a latch type switch. Repeated depressions of it turn the tracking figure on and off.

Function key 3 is a control switch for the resolution of the tracking process. Repeated depressions of the key cause the tracking resolution to alternate between a fine and a coarse degree.

Function key 4 is used to indicate that the cursor position is of interest. It causes the echo cursor to move to the current cursor. If the user is in vector drawing mode, the point is added to the vector.

Function key 5 is used to indicate that the current cursor position is to be entered into the message as a 'point'. As a 'point string' may be one or more points, this is the beginning of, or entire generation of, the message component 'point string'. Consecutive 'point' entries are merged into a 'point string' if no other interrupts intervene.

Function key 6 is the vector drawing mode latch. Repeated depressions of the key cause the display controller software to alternately enter and leave vector drawing mode. Upon entering vector drawing mode, the current cursor position is taken as the starting point of the vector. If the user uses function key 1 to position the cursor, he can depress the Here Key, function key 4. That point is then added to the vector, and the echo cursor is moved to that position. If the operator uses tracking to move the cursor, points are inserted into the vector (and the echo cursor moved correspondingly) at regular intervals of line length. The line length is selected by the selection of tracking resolution. If necessary, the tracking cursor positioning mode can be prompted to insert a point by using the Here Key. The use of FK1 and FK2 positioning may be used in combination to draw a vector. When Function key 6 is depressed to exit from vector drawing mode, the message component 'vector' has been entered into the message.

Function key 7 is the timer reset key. Depressing it causes the timer to be reset to zero, and does not generate a message component.

Function key 8 is the timer mode latch. Depressing it causes the timer mode to be changed to the opposite of its present mode. No message component is generated.

Function key 9 is the status display latch. Repeated depressions of the switch cause a status display to appear or disappear. The following information is displayed in the status display:

a) Tracking latch

b) Tracking Resolution latch

c) Vector latch

d) Timer value

e) Timer mode

f) Input buffer space remaining

g) Light pen interruptibility (see below)

h) Current Status/Function key latch

No message component is generated by Function key 9.

Function key 0 is the display protection latch. Repeated depressing of this key causes all display information (other than positional displays) to be alternately completely locked out (protected against light pen picks), or enabled to light pen interrupts as originally specified. No message component is generated.

The ATTN key is used as the New Frame Request key. Depressing it causes the display controller software to activate the next frame if it is available. Frames are selected in the order of their frame indices, with a wrap-around occurring from the highest frame index to the first frame. The message component 'new frame interaction' is

created regardless of whether or not the new frame is displayed.

The message component 'timer hit maximum' is generated automatically at the point when the timer hits a value of 2047, which in fine operation is approximately 41 seconds, and in coarse operation is approximately 44 minutes. When the timer hits maximum, it holds at the maximum value until it is reset by function key 7 or by output from the PRIG program.

A 'light pen hit' is generated when a displayed entity other than the positioning displays has been picked by the light pen. A 'light pen hit' moves the cursor, and generates a special echo cursor that is not erased until the message is sent.

The message component 'end of message' is generated and inserted automatically by depressing the SEND key, which also forwards the message to the ALAS program.

There are some simple editing and control functions available through the use of certain special keys, and these are as follows:

a) The CLEAR key empties the current input buffer when depressed.

b) The DELETE key deletes the message component

currently under construction.

c) The BKSP key destroys the last element entered in the construction of the current message component. For example, an 'alpha key string' could be altered by backspacing and retyping.

d) The RETURN key generates the equivalent of a carriage return. The cursor is returned to the echo cursor, and moved down an appropriate distance. The RETURN key also causes a special key hit.

CHAPTER IX
EXTENSIONS

9.1 INTRODUCTION

In this chapter, the author describes an approach to interactive graphical display programming to indicate a possible route for extension of the ALAS/PRIG system to include higher levels of support. In planning this system, certain problems and questions of technique are encountered, which are mentioned, though not explored in detail.

9.2 AN APPROACH TO INTERACTIVE GRAPHICAL DISPLAY PROGRAMMING

To consider higher levels of graphics support, one must select an approach for formalizing the definition of the interactive graphical display programming problem. One such approach consists of considering the interactive graphical display programming problem as a problem in language definition and implementation. The language being designed is a special purpose problem oriented language, a medium through which the user may direct and observe the attempted solution of his problems In particular, this language is expected to make use of graphical communication in reaching this solution. This approach is similar to that adopted by Kulsrud (1968) and Roberts (1966).

If one localizes the effects of a graphical display terminal to those areas of input recognition and output generation, one may choose from a number of techniques of translator generation, for example: Graham (1964), Samelson and Bauer (1969), Kanner, Kosinski and Robinson (1965), Cheatham and Sattley (1964), Irons (1961), Rosen (1964), Feldman and Gries (1968). As the design of a translator is as subject to personal factors as is any other program, it is as difficult to specify an "ideal" translator generator as it is to specify a programming language.

One style considered by the author involves the use of a structure driven analyzer, such as one of those described by Cohen and Gottlieb (1970), to generate an intermediate form of the input, this intermediate form then being used to direct control to a group of semantics routines. The display terminal presents some intriguing problems from the point of view of input entity generation; such as that of that of identifying elements picked by a light pen, for example. However, the author believes that these problems may be solved through the use of a special scanning program whose prime purpose is to identify user actions and resolve them into entities of a consistent style. Given that this may be done, the string of user actions effectively being resolved into a string of elements of a consistent generic type (such as integers, or characters), a translator can then function in terms of a language defined in terms of

this type, reducing the problem of translator generation to one which has already been explored and solved by others. The support system would consist of:

a) Scanner
b) Analyzer
c) Semanticizer
d) Preprocessing Routines
e) Display Output Routines

The scanner, analyzer and semanticizer would be table (or structure) driven. The preprocessing routines would include facilities to prepare the necessary tables (structures) for the scanner, analyzer, and semanticizer, as well as translators needed for the semantics routines. The display output routines include facilities for generating displays and controlling the environment to effect proper input recognition. To clarify facilities of the latter nature, an example facility is the ID chain facility in PRIG. In specifying Identity Characters for the nodes in a PRIG picture structure, one is creating a context that may be used for determining which displayed elements have been picked by the light pen.

Use of the system is a two phase process:

a) Phase I - Generation: In this phase, the language description is translated into various tables, structures and modules of machine code for use by the second

phase.

b) Phase II - Operation: In this phase, the output of Phase I is utilized to control the operation of the scanner, analyzer and semanticizer in processing input from and generating output for the terminal.

The programmer would prepare a four part description of the interpreter he desires:

a) Terminal Symbol Descriptions: - In each description, the terminal symbol is named, and described in terms of the required user actions (such as light pen picks) and in terms of required environmental considerations (such as displayed entities with a particular range of ID chains). For example, terminal symbol ALPHA may be described as a light pen pick (a user action) of an element with the ID chain 'ABC' (an environmental consideration).

b) The grammar of the language plus information to aid interpretation: - This part includes a description of the syntax of the language, with information that describes relevant intermediate forms to be generated upon recognition of various constructs within the input string. For the purpose of this discussion, this information will be considered in the form of tags appended to the rules describing the affected constructs, these tags indicating the significance of the constructs recognized.

c) Semantics Mapping Rules: - This part describes the mapping of the tags encoded in part b) to calls to various semantics routines.

d) Semantics Routines: - This part includes that code which the programmer desires executed upon recognition of various input.

In Phase I, the preprocessing routines would accept the above description and generate an intermediate form of the interpreter consisting of the following four components:

a) A tabular or structural representation of terminal symbol descriptions,

b) A tabular or structural representation of the grammar of the language, including semantics tags,

c) A tabular or structural representation of the semantics mapping rules, and

d) The machine representation of the semantics routines.

In Phase II, the intermediate form specified above would be used by the scanner, analyzer and semanticizer to process input from the terminal as follows:

a) A sequence of input actions are performed at the terminal and delimited as an input string.

b) Through the combined efforts of the scanner and the analyzer, the input string is transformed into an intermediate form for input to the semanticizer.

c) The semanticizer directs control to the various semantics routines under control of the output from the analyzer. In processing the message, some of the semantics routines may call upon the display output routines to update and display information.

The system is not different in principle from some other types of translator generator. The differences in detail are that:

a) One needs a special formalism for defining the source language, which can be regarded as a string language whose symbols are sequences of actions taken by the terminal operator.

b) At least some of the semantics routines must be written in a language, such as PRIG, which is oriented toward graphical communication.

The support system described above is analagous in operation to the common compile-execute sequence. The compile phase is Phase I, in which the language description is 'compiled' into an intermediate form, corresponding roughly to the object module output of a compiler. This 'object module' (the intermediate form) is then executed on an 'extended' machine (Phase II), which contains the scanner, analyzer, semanticizer and display output routines.

One may observe that in this system, 'recompilation' of an entire description is not necessarily essential to implement changes. One may also observe that the technique of implementing this system (that is, the <u>support</u> mechanisms) may vary from in one case implementing an entirely table driven support system, to in another case implementing extensive preprocessors which almost eliminate the need for the scanner, analyzer and semanticizer because they essentially create a scanner, analyzer and semanticizer which are quite rudimentary, and tailored to the language description used as input.

Another relevant consideration in implementing a support system of this nature is determining the formalism for grammar specification. Perhaps by restricting the programmer to a certain style of grammar for his language, a considerable increase in speed of analysis and/or scanning may be realized. The questions to be explored in this area are

i) What restrictions may be imposed, and

ii) Are these restrictions worth the added speed?

An added advantage of the support system as structured above is that the programmer may use only part or all of it as he desires. For example, in the case of a simple application, the programmer may require only the scanner, preferring to construct his own analyzer. In another case,

the programmer may find that the recognition of terminal symbols is extremely simple, and may construct his own scanner, while retaining the remainder of the system. Choices of this nature are largely a matter of taste, and the advantage of the above system is that such choices may be made.

The author views the construction of a system styled as above as a desirable objective, one which would benefit from the solid foundation of the picture modelling system, PRIG. PRIG's capability for embedding and recognizing contextual information in display input/output would supply an effective basis for input scanning. The structural programming capabilities of ALAS are useful for creating structure driven programs such as the analyzer and the scanner. Cohen and Gotlieb's (1970) analysis algorithms which operate under the direction of a structure describing the syntax are cases in point.

9.3 CONCLUSION

The preceding section describes a route through which ALAS and PRIG may be expanded to provide a higher level of support. The author views the conjunction of ALAS, PRIG, and structure driven input processing and structure controlled semantics as a family of support software, running the gamut from assembler level languages such as

ALAS through the highest level languages available on the machine of implementation, which can be used to write semantics routines.

CHAPTER X

CONCLUSIONS

By studying interactive graphical display programs, the author has brought out the functions common to such programs. Consideration of support software for these programs led to the design of ALAS, an assembler level data structure language, and PRIG, a graphical display interface language intended for implementation in ALAS.

Interactive graphical display programs have in common the requirement for recognition and interpretation of graphical input.

Recognition of graphical input is complicated by the use of symbolic graphical communication, that is, communication in which the displayed entities represent other more abstract entities. Graphical communication is made most flexible through the use of a data structure model of the displayed picture, which can be organized in a way which aids recognition of graphical input, generation of graphical output, and modification of the current display.

The need to act interpretively is a direct consequence of the interactive nature of such programs. An interactive graphics program is essentially an interpreter which has, as its source language, input (graphical or otherwise) from the

display terminal. Support software for the construction of interpreters is typically at the lowest level, as interpreters are usually systems programs developed by systems programmers. The operation of interpreters involves the scanning of input strings, and the resolution of the questions of what actions must be taken. Providing such support can be done most effectively, in the author's opinion, through the use of a data structure system.

As prospective graphics programmers should not be expected to be systems programmers, support should be available at a quite high level. To insure well designed software at the highest level, one must have well designed support at lower levels. The author from the outset envisaged four levels of support:

a) Level 0 - no specific support for graphics

b) Level 1 - graphics terminal interface

c) Level 2 - automated input analysis

d) Level 3 - automated input analysis and semantics sequencing

Level 1 support can be provided adequately through an appropriate modelling system. Level 2 and Level 3 support would characteristically involve table or structure driven processing. These factors, combined with the desire to develop each level of support in terms of lower levels, led to the conclusion that data structure capabilities at Level 0 would be almost essential. The desire to develop each

level of support in terms of lower levels also implies that Level 0 support should be at the assembler level to maximize efficiency.

With these objectives in mind, the author considered the suitability of several data structure languages. Of those considered, all except L^6 were at a reasonably high level, and were therefore considered unsuitable. In the case of L^6, the author felt that its capabilities in some areas were inadequate, noteably for string processing and for interfacing with program modules generated from some other source language.

To fill this gap, the author designed ALAS. ALAS has a storage management technique which allows both static and dynamic storage allocation. Static allocation can be handled by the assembler, and can result in faster code being generated than that for equivalent operations on dynamically allocated storage. Thus, by not being restricted to the use of dynamically allocated storage, the programmer is expected to obtain faster execution of his programs. The string processing capabilities of ALAS include string matching and editing instructions. ALAS's facilities for linked list processing are supplied in an incremental fashion, to afford the programmer maximum flexibility in creating and processing data structures.

A Level 1 support system, PRIG, was designed to be implemented in ALAS for the IBM 360 computer and Control Data GRID terminal. The principal component of a PRIG program is a data structure which models the displayed picture. PRIG consists of ALAS macros which create and modify picture structures, and process information input from the terminal. PRIG, to be written in ALAS, is at the same programming language level as ALAS, and should have the efficiency characteristic of such a level. PRIG was designed as an example of Level 1 support, embodying a technique of using data structures as a vehicle of communication. Being low level, it is necessarily hardware oriented. However, the structured picture concept can be applied in virtually any hardware environment.

In extending support to higher levels, table or structure driven processing holds the greatest promise. Where this type of software is available, the programmer would define a console language consisting of the statements needed to request solutions to problems in his area. This language description would be transformed into a series of structures to be used for controlling the interpretation of strings in the language as they are input from the terminal.

While ALAS is a product of graphics research, it is a general purpose data structure system. It contains facilities for string and linked list processing, arithmetic

processing, and dynamic storage management. ALAS is at the assembler level, to provide efficient coding and maximize flexibility. It has a macro handling facility which, when used in conjunction with the conditional assembly feature, affords great flexibility in the use of macros to "extend" the language. Its variable length registers and interpretive address processor represent departures from "standard" computer structure, departures intended to enhance the flexibility of ALAS. These and other features contribute to making ALAS a powerful, general purpose, data structure language.

BIBLIOGRAPHY

Barnes, Robert F., "Language Problems Posed by Heavily Structured Data", C.A.C.M., Vol. 5, No. 1, pp. 28-34, 1962.

Baskin, H.B., and Morse, S.P., "A Multilevel Modelling Structure for Interactive Graphic Design", IBM Systems Journal, Vol. 7, Nos. 3&4, 1968.

Bauer, F.L., and Samelson, K., "Sequential Formula Translation", C.A.C.M., Vol. 3, No. 2, pp. 76-83, 1960.

Bobrow, Daniel G., and Murphy, Daniel L., "Structure of a LISP System Using Two-Level Storage", C.A.C.M., Vol. 10, No. 3, pp. 155-159, 1967.

Cameron, Scott H., Ewing, Duncan, and Liveright, Michael, "DIALOG: A Conversational System with Graphical Orientation", C.A.C.M., Vol. 10, No. 6, pp. 349-357, 1967.

Cheatham, T.E., and Sattley, Kirk, "Syntax Directed Compiling", Proc. Eastern Joint Computer Conf., AFIPS, Vol. 25, pp. 31-57, 1964.

Cohen, Doron J., and Gotlieb, C.C., "A List Structure Form of Grammars for Syntactic Analysis", Computing Surveys, Vol. 2, No. 1, pp. 65-82, 1970.

Cohen, Jacques, "A use of Fast and Slow Memories in List Processing Languages", C.A.C.M., Vol. 10, No. 2, pp. 82-86, 1967.

Collins, George E., "A Method for Overlapping and Erasure of Lists", C.A.C.M., Vol. 3, No. 12, pp. 655-657, 1960.

Conway, R., Delfausse, J., Maxwell, W., and Walker, W., "CLP - The Cornell List Processor", C.A.C.M., Vol. 8, No. 4, pp. 299-304, 1963.

Coons, Steven A., "An Outline of the Requirements for a Computer-Aided Design System", Proc. Spring Joint Computer Conf., AFIPS, Vol. 23, pp. 299-304, 1963.

Corbin, Harold S., and Frank, Werner L., "Display Oriented Computer Usage System", Proc. 21st National Conf., ACM, pp. 515-526, 1966.

Deecker, G.F.P., *Interactive Graphics & A Planning Problem*, M.Sc. Thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, 1970.

Denil, N.J., "A Language and Model for Computer Aided Design", *Proc. 21st National Conf.*, ACM, pp. 527-535, 1966.

Desautels, E.J., and Smith, Douglas K., "An Introduction to the String Processing Language SNOBOL", *Programming Systems & Languages*, ed. Saul Rosen (New York: McGraw-Hill Book Company), pp. 419-454, 1967.

Deuel, Phillip, "On a Storage Mapping Function for Data Structures", *C.A.C.M.*, Vol. 9, No. 5, pp. 344-347, 1966.

Feldman, Jerome, and Gries, David, "Translator Writing Systems", *C.A.C.M.*, Vol. 11, No. 2, pp. 75-113, 1968.

Fitzwater, D.R., "A Storage Allocation and Reference Structure", *C.A.C.M.*, Vol. 7, No. 9, pp. 542-545, 1964.

Floyd, R.W., "The Syntax of Programming Languages - A Survey", *IEEE Transactions on Electronic Computers*, Vol. EC-13, pp. 346-353, 1964.

Graham, R.M., "Bounded Context Translation", *Proc. Eastern Joint Computer Conf.*, AFIPS, Vol. 25, pp. 17-29, 1964.

Gray, J.C., "Compound Data Structures for Computer Aided Design", *Proc. 22nd National Conf.*, ACM, pp. 355-365, 1967.

Haddon, B.K., and Waite, W.M., "A Compaction Procedure for Variable-Length Storage Elements", *Computer Journal*, Vol. 10, pp. 162-164, 1967.

Hammer, Peter L., and Rudeanu, Sergiu, *Boolean Methods in Operations Research and Related Areas*. Heidelberg: Springer-Verlag Berlin, 1968.

Huen, W.H., *A Graphical Display Subroutine Package*, M.Sc. Thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, 1969.

Irons, Edgar T., "A Syntax Directed Compiler for ALGOL 60", *C.A.C.M.*, Vol. 4, No. 1, pp. 51-55, 1961.

Johnson, B.V., *Computer Graphics in Logic Circuit Design*, M.Sc. Thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, 1970.

Johnson, Timothy E., "SKETCHPAD III A Computer Program for Drawing in Three Dimensions", *Proc. Spring Joint Computer Conf.*, AFIPS, Vol. 23, pp. 347-353, 1963.

Kanner, H., Kosinski, P., and Robinson, C.L., "The Structure of Yet Another Algol Compiler", *C.A.C.M.*, Vol. 8, No. 7, pp. 427-438, 1965.

Kapps, Charles A., "SPRINT A Direct Approach to List Processing Languages", *Proc. Spring Joint Computer Conf.*, AFIPS, Vol. 30, pp. 677-683, 1967.

Knowlton, Kenneth C., "A Programmer's Description of L^6", *C.A.C.M.*, Vol. 9, No. 8, pp. 616-625, 1966.

Kulsrud, H.E., "A General Purpose Graphics Language", *C.A.C.M.*, Vol. 11, No. 4, pp. 247-254, 1968.

Landin, P.J., "The Next 700 Programming languages", *C.A.C.M.*, Vol. 9, No. 3, pp. 157-166, 1966.

Lang, C.A., and Gray, J.C., "ASP - Ring Implemented Associative Structure Package", *C.A.C.M.*, Vol. 11, No. 8, pp. 550-555, 1968.

Lawson, Harold W., "PI/I List Processing", *C.A.C.M.*, Vol. 10, No. 6, pp. 358-367, 1967.

Ledley, R.S., Jacobsen, J., and Belson, M., "BUGSYS: A Programming System for Picture Processing - Not for Debugging", *C.A.C.M.*, Vol. 9, No. 2, pp. 79-84, 1966.

Lee, John A.N., *The Anatomy of a Compiler*. Reinhold Publishing Corporation, 1967.

Madnick, Stuart E., "String Processing Techniques", *C.A.C.M.*, Vol. 10, No. 7, pp. 420-424, 1967.

McCarthy, John, "Recursive Functions of Symbolic Expressions and Their Computation, Part I", *C.A.C.M.*, Vol. 3, No. 4, pp. 184-204, 1960.

Mezei, L., "SPARTA, a Procedure Oriented Programming Language for the Manipulation of Arbitrary Line Drawings", *Proc. IFIP Congress 1968*, Vol. 1, pp. 597-604, 1968.

Narasimhan, R., "Syntax-Directed Interpretation of Classes of Pictures", C.A.C.M., Vol. 9, No. 3, pp. 166-172, 1966.

Newell, A., and Tonge, F.M., "An Introduction to Information Processing Language V", C.A.C.M., Vol. 3, No. 4, pp. 205-211, 1960.

Newman, W.M., "A System for Interactive Graphical Programming", Proc. Spring Joint Computer Conf., AFIPS, Vol. 32, pp. 47-54, 1968.

Ophir, D., Rankowitz, S., Shepherd, B.J., and Spinrad, R.J., "BRAD: The Brookhaven Raster Display", C.A.C.M., Vol. 11, No. 6, pp. 415-416, 1968.

Pingle, Karl, A List Processing System for Picture Processing, Stanford Artificial Intelligence Laboratory Operating Note No. 33, 1967.

Ramamoorthy, C.V., "Code Structures for Protection and Manipulation of Variable Length Items", C.A.C.M., Vol. 8, No. 1, pp. 35-38, 1965.

Raphael, Bertram, "The Structure of Programming Languages", C.A.C.M., Vol. 9, No. 2, pp. 67-71, 1966.

Roberts, Lawrence G., "Graphical Communication and Control Languages", Proc. 2nd Information System Sciences Conf., (Washington: Spartan Books), 1964.

Roberts, Lawrence G., "A Graphical Service System with Variable Syntax", C.A.C.M., Vol. 9, No. 3, pp. 173-176, 1966.

Rosen, Saul, "A Compiler-Building System Developed by Brooker and Morris", C.A.C.M., Vol. 7, No. 7, pp. 403-414, 1964.

Ross, Douglas T., and Rodriguez, Jorge E., "Theoretical Foundations for the Computer Aided Design System", Proc. Spring Joint Computer Conf., AFIPS, Vol. 23, pp. 305-322, 1963.

Rotenberg, Naomi, and Opler, Ascher, "Variable Width Stacks", C.A.C.M., Vol. 6, No. 10, pp. 608-610, 1963.

Schorr, H., and Waite, W.M., "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures", C.A.C.M., Vol. 10, No. 8, pp. 501-506, 1967.

Shalla, Leon, "Automatic Analysis of Electronic Digital Circuits Using List Processing", C.A.C.M., Vol. 9, No. 5, pp. 372-380, 1966.

Storm, Edward F., "CHAMP - Character Manipulation Procedures", C.A.C.M., Vol. 11, No. 8, pp. 561-566, 1968.

Stotz, Robert, "Man-Machine Console Facilities for Computer Aided Design", Proc. Spring Joint Computer Conf., AFIPS, Vol. 23, pp. 323-328, 1963.

Sutherland, I.E., "SKETCHPAD, A Man-Machine Communication System", Proc. Spring Joint Computer Conf., AFIPS, Vol. 23, pp. 329-346, 1963.

Sutherland, William R., "The CORAL Language and Data Structure", Appendix C of The On-Line Graphical Specification of Computer Procedures, a Doctoral Thesis submitted to M.I.T. on January 10, 1966.

Thomas, Eugene M., "'GRASP' - A Graphic Service Program", Proc. 22nd National Conf., ACM, pp. 395-402, 1967.

Trundle, F.W.L., "LITHP - an ALGOL List Processor", Computer Journal, Vol. 9, pp. 167-172, 1966.

Van Dam, Andries, and Evans, David, "A Compact Data Structure for Storing, Retrieving and Manipulating Line Drawings", Proc. Spring Joint Computer Conf., AFIPS, Vol. 30, pp. 601-610, 1967.

Waite, William M., "A Language Independent Macro Processor", C.A.C.M., Vol. 10, No. 7, pp. 433-440, 1967.

Warshall, Stephen, and Shapiro, Robert M., "A General Purpose Table-Driven Compiler", Proc. Eastern Computer Conf., AFIPS, Vol. 25, pp. 59-65, 1964.

Weizenbaum, J., "Knotted List Structures", C.A.C.M., Vol. 5, No. 3, pp. 161-165, 1962.

Weizenbaum, J., "Symmetric List Processor", C.A.C.M., Vol. 6, No. 9, pp. 524-536, 1963.

Wiseman, N.E., And Hiles, J.O., "A Ring Structure Processor for a Small Computer", Computer Journal, Vol. 10, pp. 338-346, 1967.

Yershov, A.P., "One View of Man-Machine Interaction", J.A.C.M., Vol. 12, No. 3, pp. 315-325, 1969.

APPENDIX A

A FREEHAND INPUT PROGRAM

The objective of the following program is to provide a simple freehand drawing input capability through the use of the light pen. The user utilizes the vector, point and text input modes of the display controller software to draw on the CRT, and the program builds a structural representation of the picture to provide for editing and processing for hard copy.

The display format consists of a series of 'command words' displayed along the top of the display area, and a drawing area, surrounded by a frame. The user's input may be considered as strings in a language which is defined by the grammar in Table A.1.

The semantics of the language are as follows:

i) An [input element] of the type [picture name] attaches the [alpha key string] within it to the picture as its title.

ii) An [input element] of the type PLOT causes the picture currently viewed to be converted for hard copy.

iii) An [input element] of the type [erase command] causes part or all of the picture currently viewed to be erased. If the [light pen pick list option] is empty, the entire picture is erased. Otherwise, the elements picked

Table A.1 Syntax of Input for Freehand Input Program

```
[input string]::=[end of message]•[input element][input
                          string]

[input element]::=[command]•[drawing information]•[null
                          indicant]

[command]::=[picture name]•[plot]•[erase command]•[stop]

[picture name]::=PICTURE•PICTURE[alpha key string]

[plot]::=PLOT

[erase command]::=ERASE•ERASE[light pen pick list]

[stop]::=STOP

[light pen pick list]::=[light pen pick]•[light pen pick
                          list][light pen pick]

[drawing information]::=[drawing element]•[drawing
                          information][drawing element]

[drawing element]::=[alpha key hit]•[alpha key string]•
                          [point string]•[vector]

[null indicant]::=[function key hit]•[special key hit]•[new
                          frame interaction]•[timer hit
                          maximum]
```

Terminal Symbols:

PICTURE PLOT ERASE STOP	These are light pen picks of the command words displayed on the screen.
[alpha key hit] [light pen pick] [alpha key string] [point string] [vector] [special key hit] [function key hit] [timer hit maximum] [new frame interaction] [end of message]	These are fundamental elements of PRIG input, and are identified by PRIG display controller software (See Sections 8.4 and 8.5).

(other than command words) are deleted from the picture.

iv) An [input element] of the type STOP causes the program to terminate.

v) An [input element] of the type [drawing element] causes an addition to the picture such that the [drawing element] is displayed, and can be uniquely identified for the [erase command].

vi) An [input element] of the type [null indicant] is ignored.

Table A.2 is the ALAS/PRIG program to perform the above task. Two modules have been left out in the interest of brevity, as they would add nothing to the example. These two modules, and their functions, are:

i) PLOTPREP, which prepares the plotter interface, and

ii) PLOTTER, which parses the current picture in order to generate code for the plotter which draws the entire picture except for the command words.

Table A.2 Freehand Input Program in PRIG

```
*   PBLOCK: MAIN
*
*   PBLCCK *MAIN* IS THE OVERALL CONTROLLING MODULE OF THE
*   FREEHAND INPUT PROGRAM, AND INCLUDES THE INPUT
*   INTERPRETATION AND TRANSMIT/RECEIVE LOOPS
*
                PBLOCK
                EXTRN
                GLBL
*   INITIALIZATION
                IDISP      2
                E          PLOTPREP
                E          PRIMER
*   MASTER LOOP: SEND/RECEIVE CYCLE
LETSGO          FRAME      MAST
                SEND       22
                AWGINP
                ZE         I0
*   INPUT INTERPRETATION LOOP
NEXT            A          I0,=F'1'
                ACTION     I0,I1
ACTBR           B          ACT<I1>_
*   MESSAGE COMPONENT IS A SINGLE ALPHA KEY HIT
ONEALPHA        AKCD       I0,C0
                B          ALPND
*   MESSAGE COMPONENT IS AN ALPHA KEY STRING
STRALPHA        ASTR       I0,C0
*   ATTACH ALPHA KEY COMPONENT TO PICTURE
ALPND           E          ANODER
                B          NEXT
*   MESSAGE COMPONENT IS A POINT STRING
STRPOINT        AVECT      I0,I2,A0,A1
*   ATTACH POINT STRING TO PICTURE
                E          PNODER
                B          NEXT
*   MESSAGE COMPONENT IS A VECTOR
STRVECT   AVECT            I0,I2,A0,A1
*   ATTACH VECTOR TO PICTURE
                E          VNODER
                B          NEXT
*   MESSAGE COMPONENT IS A LIGHT PEN HIT;
*   IS IT A COMMAND?
COMTEST   AID              I0,C0
                C          C0,=X'00'
*   IF IT ISN'T AT THIS POINT, IGNORE IT
                BC         7,NEXT
*   IT IS A COMMAND; WHICH ONE?
COMTEST1        L          I1,C0:(1,1)
                B          COM<I1>_
```

```
*  COMMAND WAS STOP; SHUT DOWN AND CLEAR OUT
TERMINUS       XDISP
               REV
*  COMMAND WAS PLOT; PLOT IT
PLOTIT         E         PLOTTER
               B         NEXT
*  COMMAND WAS PICTURE; CHANGE THE TITLE
TITLIT         E         TNODER
               B         NEXT
*  COMMAND WAS ERASE; IS ALL OF THE PICTURE TO GO?
ERASE          A         I0,=F'1'
               ACTION    I0,I1
               C         I1,=F'1'
*  IF THE NEXT COMPONENT IS NOT A LIGHT PEN HIT, YES
               BC        7,DROPIC
               AID       I0,C0
               C         C0,=X'00'
*  IF THE NEXT COMPONENT PICKS A COMMAND, YES
               BC        8,DROPIC
*  OTHERWISE, THE ENTITY PICKED IS DESTINED FOR OBLIVION
ERSLOOP  ST              C0,IDCH
*  FIND THE COMMAND WHICH SHOWS THE ENTITY
               LOSH      MAST,3,,=F'1',IDCH
*  WIPE IT OUT
               SETND     NULL
*  IS THE NEXT MESSAGE COMPONENT ALSO A LIGHT PEN PICK?
               A         I0,=F'1'
               ACTION    I0,I1
               C         I1,=F'1'
*  IF NOT, BACK TO THE MAIN INPUT INTERPRETATION LOOP
               BC        7,ACTBR
*  IF SO, DOES IT PICK A COMMAND?
               AID       I0,C0
               C         C0,=X'00'
*  IF IT PICKS A COMMAND, BACK TO THE COMMAND INTERP LOOP
               BC        8,COMTEST1
*  OTHERWISE, ANOTHER ENTITY FOR THE AXE
               B         ERSLOOP
*  TIME TO DISPOSE OF THE ENTIRE PICTURE
DROPIC         E         BILDMAST
               B         ACTBR
*  CONSTANTS, ETC.
IDA            DFA       C
IDB            DFA       C
IDCH           DFA       2C
               END
*
*
```

```
*
*  ADDRESS BLOCKS USED BY *MAIN* IN INTERPRETING INPUT
*
ACT            ABLCCK
               DFA         A'NEXT'
               DFA         A'COMTEST'
               DFA         A'ONEALPHA'
               DFA         A'NEXT'
               DFA         A'NEXT'
               DFA         A'STRALPHA'
               DFA         A'STRPOINT'
               DFA         A'STRVECT'
               DFA         A'NEXT'
               DFA         A'NEXT'
               DFA         A'LETSGO'
               END
*
*
COM            ABLCCK
               DFA         A'TERMINUS'
               DFA         A'TITLIT'
               DFA         A'PLOTIT'
               DFA         A'ERASE'
               DFA         A'NEXT'
               END
*
*
*
*  THE FOLLOWING SEVEN PBLOCKS ARE USED IN COMPLETING
*  THE FUNCTIONS OF *MAIN*
*
*  PBLCCK *ANODER* ATTACHES AN ALPHA KEY COMPONENT TO
*  THE PICTURE
*
ANODER         PBLOCK
               EXTRN
*  GET THE LOCATION OF THE ALPHA KEY COMPONENT
               AX          I0,I2
               ST          I2,FX
               AY          I0,12
               ST          I2,FY
*  GENERATE NODE *SPARE* WITH ALPHA ENTITY
               NODE        SPARE
               ID          IDB
               TEXTR       C0,FX,FY
               ENDNODE
*  ATTACH A COMMAND *SHOW*ING *SPARE* TO *MAST*
               E           CNER
               REV
FX             DFA         I
FY             DFA         I
               END
*
```

```
*
*  PBLOCK *BILDMAST* (RE)CREATES THE MAIN NODE *MAST*
*
BILDMAST       PBLCCK
               EXTRN
               NODE      MAST
               ID        =X'00'
               SHOW      PC
               SHOW      PLC
               SHCW      ERC
               SHOW      STC
               SHOW      TNODE
               ID        =X'01'
               ENDNODE
*  INITIALIZE THE ID CHARACTERS
               ZE        I2
               ST        I2,IDB
               A         I2,=F'1'
               ST        I2,IDA
               REV
               END
*
*
*  PBLOCK *CNER* ATTACHES A COMMAND *SHOW*ING *SPARE*,
*  INCREMENTING THE ID AS NECESSARY
*
CNER           PBLCCK
               EXTRN
*  CREATE A NODE CONTAINING THE *SHOW*, AND AN *ID*
*  IF NECESSARY
               NODE      ESHOW
               SHOW      SPARE
*  INCREMENT ID
               L         I2,IDB
               A         I2,=F'1'
               ST        I2,IDB
               S         I2,=F'256'
               BC        4,GO
REA            L         I2,IDA
               A         I2,=F'1'
               ST        I2,IDA
               S         I2,=F'256'
               BC        11,REA
               ID        IDA
GO             ENDNODE
               CHAIN     MAST,ESHOW
               REV
               END
*
```

```
*
*   PBLOCK *PNODER* ATTACHES A POINT STRING TO THE PICTURE
*
PNODER          PBLOCK
                EXTRN
*   STORE THE NUMBER OF POINTS IN THE STRING
                ST        I2,NP
*   GENERATE NODE *SPARE* CONTAINING THE POINT STRING
                NODE      SPARE
                ID        IDB
                POINT     NP,0(A0),0(A1)
                ENDNODE
*   ATTACH A COMMAND *SHOW*ING *SPARE* TO *MAST*
                E         CNER
                REV
NP              DFA       F
                END
*
*
*   PBLOCK *PRIMER* SETS UP THE COMMAND WORD NODES, AND
*   BUILDS THE FIRST INSTANCE OF NODE *MAST*
*
PRIMER          PBLOCK
                EXTRN
                NODE      PC
                ID        =X'01'
                TEXT      =F'7',=C'PICTURE',=F'100',=F'950'
                ENDNODE
                NODE      PIC
                ID        =X'02'
                TEXT      =F'4',=C'PLOT',=F'350',=F'950'
                ENDNODE
                NODE      ERC
                ID        =X'03'
                TEXT      =F'5',=C'ERASE',=F'600',=F'950'
                ENDNODE
                NODE      STC
                ID        =X'00'
                TEXT      =F'4',=C'STOP',=F'850',=F'950'
                ENDNODE
                NODE      TNODE
                ID        =X'04'
                ENDNODE
                E         BILDMAST
                REV
                END
*
```

```
*
*   PBLOCK *TNODER* CREATES A NEW TITLE NODE, AND
*   DISPLACES THE OLD ONE WITH IT
*
TNODER          PBLOCK
                EXTRN
*   FIND OLD TITLE NODE
                LOSH        MAST,4,TNODE
*   WIPE IT OUT
                SETND       NULL
*   START BUILDING A NEW ONE
                NODE        TNODE
                ID          =X'04'
                A           I0,=F'1'
                ACTION      I0,I1
                C           I1,=F'2'
                BC          8,ONEC
                C           I1,=F'5'
                BC          7,NOTIT
                ASTR        I0,C0
                B           FITIT
ONEC            AKCD        I0,C0
FIXIT           SIZE        C0,F0
                M           F0,=F'-8'
                A           F0,=F'512'
                ST          F0,TX
                TEXTR       C0,TX,=F'900'
                B           FIN
NOTIT           S           I0,=F'1'
FIN             ENDNODE
                SETND       TNODE
                REV
TX              DFA         F
                END
*
*
*   PBLOCK *VNODER* ATTACHES A VECTOR TO THE PICTURE
*
VNODER          PBLCCK
                EXTRN
                S           I2,=F'1'
                ST          I2,NV
*   CREATE NODE *SPARE* WHICH CONTAINS THE VECTOR
                NODE        SPARE
                ID          IDB
                VCT         NV,0(A0),0(A1)
                ENDNODE
*   ATTACH *SPARE* TO *MAST*
                E           CNER
                REV
NV              DFA         F
                END
```

APPENDIX B

TABULATION OF MNEMONICS

This appendix contains a complete tabulation of the mnemonics discussed in connection with ALAS and PRIG. The mnemonics are listed in alphabetical order, complete with the instruction name, type and host section within the thesis.

The instruction type is indicated by a character, A for assembler, M for machine, and P for PRIG.

Mnemonic	Name	Type	Section
A	Add	M	5.7.2
ABLOCK	Address Block	A	5.8.2
ACONT	Access Contents Subfield	P	8.4
ACTICN	Access Action ID	P	8.4
ADNODE	Add Node	P	8.3
ADS	Add Dynamic Space	M	5.7.6
AFRAM	Access Frame Index	P	8.4
AID	Access ID Chain	P	8.4
AKCD	Access Key Code	P	8.4
ALLCB	Allocator Control Block	A	5.8.2
AR	Add Register	M	5.7.2
ASTAT	Access Status	P	8.4
ASTFRTI	Access Status/Frame/Time Composite	P	8.4

Mnemonic	Name	Type	Section
ASTR	Access Character String	P	8.4
ATIME	Access Timer Value	P	8.4
AVECT	Access Vector Addresses	P	8.4
AWGINP	Await Graphical Input	P	8.4
AX	Access X Value	P	8.4
AXY	Access X-Y Composite	P	8.4
AY	Access Y Value	P	8.4
BC	Branch Conditional	M	5.7.7
BCR	Branch Conditional Register	M	5.7.7
BGINP	Start Input	M	5.7.7
BGOUT	Start Output	M	5.7.7
BLANK	Set Blanking On	P	8.2.4
BLINK	Set Blinking On	P	8.2.4
BUL	Build Left	M	5.7.3
BULR	Build Left Register	M	5.7.3
BUR	Build Right	M	5.7.3
BURR	Build Right Register	M	5.7.3
C	Compare	M	5.7.1
CALL	Call	M	5.7.7
CHAIN	Chain	P	8.2.4
CL	Compare Length	M	5.7.3
CLIR	Compare Length to Integer Register	M	5.7.3
CLR	Compare Length Register	M	5.7.3
CMP	Compress	M	5.7.3
CNODE	Copy Node	P	8.2.4

Mnemonic	Name	Type	Section
CNT	Count	M	5.7.3
CNTI	Count Immediate	M	5.7.3
CNTR	Count Register	M	5.7.3
CR	Compare Register	M	5.7.1
D	Divide	M	5.7.2
DBB	Device Busy Branch	M	5.7.7
DBBR	Device Busy Branch Register	M	5.7.7
DBE	Device Busy Execute	M	5.7.7
DBER	Device Busy Execute Register	M	5.7.7
DBLOCK	Get Dynamic Block	M	5.7.6
DFA	Define Field Absolute	A	5.8.2
DFAD	Get Dynamic Absolute Field	M	5.7.6
DFR	Define Field Relative	A	5.8.2
DFRD	Get Dynamic Relative Field	M	5.7.6
DIODE	Declare I/O Device	A	5.8.2
DISINT	Disable Interrupts	P	8.2.3
DP	Define Pattern	A	5.8.2
DR	Divide Register	M	5.7.2
DV	Define Vector	A	5.8.2
EC	Execute Conditional	M	5.7.7
ECR	Execute Conditional Register	M	5.7.7
EJECT	Eject Page	A	5.8.1
EL	Eliminate	M	5.7.3
ELR	Eliminate Register	M	5.7.3
END	End	A	5.8.2

Mnemonic	Name	Type	Section
ENDNCDE	Endnode	P	8.2.1
ENINT	Enable Interrupts	P	8.2.3
ENTRY	Entry	A	5.8.3
EXP	Expand	M	5.7.3
EXTRN	External	A	5.8.3
EXTRND	External Dynamic	A	5.8.3
FD	Full Divide	M	5.7.2
FDR	Full Divide Register	M	5.7.2
FKBE	Function Key Branch Equal	P	8.2.3
FKBNE	Function Key Branch Note Equal	P	8.2.3
FM	Full Multiply	M	5.7.2
FMR	Full Multiply Register	M	5.7.2
FRAME	Frame	P	8.3
FSTST	Frame Status	P	8.3
GLBL	Global	A	5.8.3
GLBLD	Global Dynamic	A	5.8.3
ID	Identify	P	8.2.3
IDISP	Initialize Display	P	8.3
IEF*	Index Element Found	M	5.7.3
IEU*	Index Element Unfound	M	5.7.3
IIEF*	Index Immediate Element Found	M	5.7.3
IIEU*	Index Immediate Element Unfound	M	5.7.3
IND	Indirect	M	5.7.4
INP	Input	M	5.7.7
INTCB	Interrupt Control Block	A	5.8.2

Mnemonic	Name	Type	Section
IODCB	I/O Device Control Block	A	5.8.2
ISF*	Index Substring Found	M	5.7.3
KIO	Halt Input/Output	M	5.7.7
L	Load	M	5.7.1
LBL	Length of Block	M	5.7.6
LEND	Loop End	M	5.7.7
LM	Load Multiple	M	5.7.1
LN	Load Negative	M	5.7.2
LNR	Load Negative Register	M	5.7.2
LOFF	Listing Off	A	5.8.1
LON	Listing On	A	5.8.1
LOOP	Loop	M	5.7.7
LOSH	Locate Show	P	8.2.4
LP	Load Positive	M	5.7.2
LPR	Load Positive Register	M	5.7.2
LR	Load Register	M	5.7.1
LS	Load Specified	M	5.7.1
LT	Load and Test	M	5.7.1
LTR	Load and Test Register	M	5.7.1
M	Multiply	M	5.7.2
MEMR	Membership Register	M	5.7.3
MR	Multiply Register	M	5.7.2
MRG	Merge	M	5.7.3
MRGR	Merge Register	M	5.7.3
ND	And	M	5.7.3

Mnemonic	Name	Type	Section
NDR	And Register	M	5.7.3
NODE	Node	P	8.2.1
NR	Negate Register	M	5.7.2
NXFR	Request Next Frame	P	8.2.3
NXSH	Locate Next Show	P	8.2.4
OR	Or	M	5.7.3
ORR	Or Register	M	5.7.3
OUT	Output	M	5.7.7
OV	Overlay	M	5.7.3
OVR	Overlay Register	M	5.7.3
PBLOCK	Program Block	A	5.8.2
PIND	Push and Indirect	M	5.7.4
PLOOP	Pop Loop Stack	M	5.7.7
PNT	Point	M	5.7.4
POINT	Point	P	8.2.2
POP	Pop	M	5.7.1
PSH	Push	M	5.7.1
QBLA	Query Blank	P	8.2.4
QBLI	Query Blink	P	8.2.4
QDE	Query Depth of Execute	M	5.7.7
QDL	Query Depth of Loop	M	5.7.7
QDS	Query Dynamic Space	M	5.7.6
QFS	Query Free Space	M	5.7.6
QOR	Query Origin	P	8.2.4
QSC	Query Scale	P	8.2.4

Mnemonic	Name	Type	Section
QTI	Query Timer Limits	P	8.2.4
REDISP	Reinitialize Display	P	8.3
REV	Revert	M	5.7.7
RNODE	Repeat Node	P	8.2.1
RT	Rotate	M	5.7.3
RTR	Rotate Register	M	5.7.3
S	Subtract	M	5.7.2
SBLOCK	Static Block	A	5.8.2
SCAN	Scan	M	5.7.4
SCANL	Scan Limited	M	5.7.4
SCC	Set Condition Code	M	5.7.7
SDS	Set Dynamic Space	M	5.7.6
SEND	Send	P	8.3
SETND	Set Node	P	8.2.4
SETOR	Set Origin	P	8.2.4
SETSC	Set Scale	P	8.2.4
SETTI	Set Timer Limits	P	8.2.4
SH	Shift	M	5.7.3
SHOW	Show	P	8.2.1
SHR	Shift Register	M	5.7.3
SIZE	Size	M	5.7.3
SM	Set Mask	M	5.7.7
SPACE	Space Listing	A	5.8.1
SR	Subtract Register	M	5.7.2
SS	Set Status	M	5.7.7

Mnemonic	Name	Type	Section
ST	Store	M	5.7.1
STJ	Store Justified	M	5.7.1
STM	Store Multiple	M	5.7.1
STS	Store Specified	M	5.7.1
SW	Swap	M	5.7.1
SWR	Swap Register	M	5.7.1
SYMBCL	Symbol	P	8.2.2
TEXT	Text	P	8.2.2
TEXTC	Text Continued	P	8.2.2
TEXTR	Text Register	P	8.2.2
TEXTRC	Text Register Continued	P	8.2.2
TITLE	Title	A	5.8.1
TMBL	Branch on Timer Low	P	8.2.3
TMBNL	Branch on Timer Not Low	P	8.2.3
TMC	Timer Control	P	8.2.3
TR	Translate	M	5.7.3
TRB	Translate and Branch	M	5.7.7
TRE	Translate and Execute	M	5.7.7
TRT*	Translate and Test	M	5.7.3
UNBLANK	Set Blanking Off	P	8.2.4
UNBLINK	Set Blinking Off	P	8.2.4
VBC	Convert Bits to Characters	M	5.7.5
VCB	Convert Characters to Bits	M	5.7.5
VCN	Convert Characters to Numerics	M	5.7.5
VCT	Vector	P	8.2.2

Mnemonic	Name	Type	Section
VCTC	Vector Continued	P	8.2.2
VEC	Convert Floating to Characters	M	5.7.5
VECE	Convert Floating to Characters	M	5.7.5
VFC	Convert Fixed to Characters	M	5.7.5
VNN	Convert Numeric to Numeric	M	5.7.5
WAIT	Wait	M	5.7.7
WD	Widen	M	5.7.3
XBL	Delete Block	M	5.7.6
XDISP	Close Display	M	8.3
XFI	Delete Field	M	5.7.6
XNODE	Delete Node	P	8.2.1
XR	Exclusive Or	M	5.7.3
XRR	Exclusive Or Register	M	5.7.3
ZE	Zero or Empty	M	5.7.1
ZEB	Zero or Empty Branch	M	5.7.7
ZEBR	Zero or Empty Branch Register	M	5.7.7
ZEE	Zero or Empty Execute	M	5.7.7
ZEER	Zero or Empty Execute Register	M	5.7.7