A SYSTEM DESIGN TOOL FOR AUTOMATICALLY
GENERATING FLOWCHARTS AND PREPROCESSING PASCAL.

Howard

James W. Keller
Captain USAF

Dec 79

118

Approved for public release; distribution unlimited

012245
A SYSTEM DESIGN TOOL FOR AUTOMATICALLY
GENERATING FLOWCHARTS AND PREPROCESSING PASCAL

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

James H. Keller, B.A.
Captain USAF
Graduate Computer Science
December 1979

Approved for public release; distribution unlimited
Acknowledgements

I would like first and foremost to express my thanks to the members of the thesis committee, Professors Ross, Lamont, Rutledge, Borky, and Black. Their criticisms and recommendations were helpful and appreciated. Professor Hartrum, in addition to providing routines that were used in the graphics handler of chapter 4, was helpful in guiding me to solutions of several specific problems with the PDP-11 system.

I received much assistance from three individuals associated with the Air Force Avionics Laboratory: Captain Walter Seward and Mr. Joseph McClendon, AFAL/AAF-2, and Mr. Neil Eastridge, DEC. Without their help, little progress would have been made on the data structure handler of chapter 3.

I am deeply indebted to the assistance offered me by two of my associates at AFIT, Captains Brian Johnson and Brian Boesch. These two gifted individuals gave unselfishly of their time to help me transport source files among alternative demonstration devices and to tutor me in the use of UCSD Pascal.

Lastly, I am indebted to Bernadine Lanzano, Professor R. Oldehoeft, and Professor Leonard Weiner for taking time from their busy schedules to answer my correspondences.

James H. Keller
Table of Contents

1. Introduction 1

1.1 The Choice of an Implementation Language 2
1.2 The Use of Non-Standard Flowcharts 3
1.3 What this Research Demonstrates 3
  1.3.1 Generating Source Code by Refining Flowcharts 4
  1.3.2 Graphical Debugging 4
  1.3.3 Facilitating Software Maintenance 5
1.4 Anticipated Benefits 6
  1.4.1 Neat, Up-to-date Flowcharts 6
  1.4.2 Debugging 7
  1.4.3 Improved Software Structure 7
  1.4.4 More Reliable Software 8
1.5 Limitations 8
  1.5.1 Familiarization with Pascal 8
  1.5.2 System Resources Dependency 9
  1.5.3 Limited Pascal Capabilities 10

2. Discussion of Related literature 11

2.1 Available Products for Automating Program Documentation 11
2.2 The Relationship between Flowcharts and Code 12
  2.2.1 Automating Flowcharts and Code 12
  2.2.2 Conceptualization Relation between Flowcharts and Code 14
2.3 Discussion Concerning Software Maintenance and Reliability 15
  2.3.1 Software Maintenance 15
  2.3.2 Reliability 16
2.4 Some Fundamental Concepts in Flowcharting 16
  2.4.1 Two-Dimensional Grammers 16
  2.4.2 Limiting the Use of Constructs 17
  2.4.3 A Structured Flowcharting Convention 18

3. The Automatic Generation of Flow Charts and Source Code 20

3.1 The PDP/Tektronix Graphics System 20
  3.1.1 Documentation of the Graphics Handler Routines 20
  3.1.2 Stored Flowchart Figures 21
3.2 The Data Structure Handler 21
  3.2.1 Data Storage Representation 22
  3.2.2 Explanation of the Handler’s Commands 24
    3.2.2.1 Creating a New System Description 25
    3.2.2.2 Getting a System Description from Disk 26
    3.2.2.3 Editing the System Description 26
    3.2.2.4 Saving the Description on Disk 27
    3.2.2.5 Producing Pascal Source Output 27
    3.2.2.6 Producing Flowchart Drawings 27
    3.2.2.7 Exiting the Handler Routine 27

4. The PDP-11/Tektronix Graph Drawing System 28

4.1 Introduction 28
4.2 Equipment 28
4.3 Running the graph system 28
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-1:</td>
<td>The If-Then-Else Construct</td>
<td>4</td>
</tr>
<tr>
<td>Figure 1-2:</td>
<td>Ambiguity of an If-then-if-then-else Construct</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2-1:</td>
<td>Jackson's Basic Control Structures</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2-2:</td>
<td>Sample Program Representations</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3-1:</td>
<td>The If-then-else Construct</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3-2:</td>
<td>The Do-while Construct</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3-3:</td>
<td>The Case Construct</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3-4:</td>
<td>A Data Record in the Data Structure</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3-5:</td>
<td>Organization of the Data Records</td>
<td>23</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Storage and Output Representations of Entry Types</td>
<td>23</td>
</tr>
<tr>
<td>4-1</td>
<td>Format of Data Table Description</td>
<td>30</td>
</tr>
<tr>
<td>4-2</td>
<td>File Control List</td>
<td>32</td>
</tr>
</tbody>
</table>
Abstract

The portion of overall system costs attributable to software development and maintenance is presently near 50% and is continually increasing. Programmers and analysts are diligently searching for tools to assist them by automating the analysis, design, and documentation of software systems.

Flowcharting has lost some of its support as a powerful design tool due to the need for discipline, patience, and to some degree artistic talent. Automatic flowcharting, designed for specific languages and machines, provides automatic documentation only. No attempt has been made to link the automatic flowcharting to the compiler-ready code.

This study begins the development of an automatic program design tool to graphically display and update flowcharts and provide this link between the flowchart and the system it represents. A method of detailed, automatic design of programs, down to the elemental source language level, is proposed which displays graphical flowchart constructs and provides for iterative, stepwise refinements of the flowcharts. The final system, described by selecting flowchart constructs and completing the descriptions of the details of each construct, is maintained in a data structure that allows for subsequent refinement and for optionally producing a compiler-ready source listing.
1. Introduction

Are flowcharts worth the effort in software design? Considerable differences of opinion exist. Some programmers believe flowcharts only in documenting the final product and thus they use other tools, such as structured English, to aid them in the design process. Others believe flowcharts are indispensable in the development of efficient and structured code. Perhaps a middle-of-the-road position is reflected by Kernighan and Plauger who comment on program documentation in general [10]:

"The best documentation for a computer program is a clean structure. It also helps if the code is well formatted, with good mnemonic identifiers, labels, and a smattering of enlightening comments. Flowcharts and program descriptions are of secondary importance; the only reliable documentation of a computer program is the code itself. The reason is simple — whenever there are multiple representations of a program, the chance for discrepancy exists. If the code is in error, artistic flowcharts and detailed comments are to no avail."

One of the main objections to developing accurate and detailed flowcharts may be the frustrations experienced by programmers with limited artistic talents. If a significant effort is used to create an early edition of the flowcharts, reluctance rapidly builds up against redrawing when changes are subsequently necessary. Automating the process of flowcharting would be extremely beneficial to the programmer. The initial design would be neat and subsequent redrawings, made

---

1 The use of term "programmer" in this report is intended to include the tasks of the "analyst" or "designer"; the terms are considered synonymous.

2 Although flowcharting is one of several graphical tools for the design and analysis of systems, only flowcharts will be discussed in this investigation.
necessary by the seemingly endless succession of modifications, would be just as presentable as the first.

The iteration between changing the code and changing the flowchart is extremely awkward and time consuming. Lanzano commented on the considerable time wasted in program evolution by the flowchart-to-code-to-analysis-to-flowchart process [11]. She suggested a computer-aided design approach to developing flowcharts to aid the programmer. The objective of this investigation is to demonstrate an interactive system which will aid the programmer in designing flowcharts and will simultaneously produce a source input file of the same program version. The proposed system will display to the programmer a menu of flowchart constructs that can be included in a series of successive, top-down refinements of a flowchart. The refinement of the system thus being designed will continue until the precise source language statements are specified. The data structure which keeps track of construct or source statements will also be used to generate the precise source code for the program.

This study assumes the programmer will prefer flowcharts as a tool in the process of designing and coding. Former flowcharters who have become frustrated with managing the flowcharting effort should find the automation of flowcharting proposed a considerable help in their work.

1.1 The Choice of an Implementation Language

Once a software system is adequately defined in terms of flowcharts, the transition to precise language statements should be simple. By providing the programmer with a set of three structured flowchart constructs, the data structure handler will help guide the programmer
toward the development of highly structured code. Because of this structuring characteristic, Pascal will be the language used for demonstration of source language preparation and output. The primary consideration for this choice is the parallelism between basic programming constructs (block structures, if-then-else, do-while, and case constructs) and the Pascal language itself. Secondly, Pascal was chosen because of its degree of acceptance in areas of computing ranging from hobby computers to the base language for the programming language Ada [7]. Although Pascal was chosen for the above reasons, other languages could have been targeted for output with the same results expected. Only slight modifications of the data structure handler would be necessary.

1.2 The Use of Non-Standard Flowcharts

Throughout this report, the use of ANSI standard flow charts was rejected in favor of the flowcharting standards designed and prescribed for use at Arizona State University by Professors Roman and Oldehoeft [13]. For use in this investigation, this standard is far superior to the ANSI standard in two important areas:

1. It is a structured flowcharting system, with a structure that is identical to three main programming constructs of Pascal (see section 3.1.2), and

2. The flowchart diagrams require much less space on the printed page—a characteristic that will be extremely helpful when conceptualizing program composition from flowchart displays.

1.3 What this Research Demonstrates

This system will demonstrate three basic capabilities:

- The capability to generate a completely specified source program by stepwise refinement of graphically displayed flowcharts (section 1.3.1)
- The capability to provide a method for graphical debugging of a system (section 1.3.2)

- The capability to provide a simpler and more reliable end-product documentation that will facilitate software maintenance (section 1.3.3).

1.3.1 Generating Source Code by Refining Flowcharts

To demonstrate the capability to generate a completely specified source program by stepwise refinement of flowcharts, a data structure handler will be constructed that will interact with the programmer, record his/her menu selections, and display the flowchart as specified up to that point. The programmer will continue to refine his/her system of flowcharts until the elemental Pascal statements are all included within the flowcharts. The data structure will then be comprised of only two general types of entries – flowchart constructs and Pascal source statements.

Along with the ability to display flowcharts, the data structure handler will be designed to list the precise Pascal source statements properly structured and formatted for subsequent compilation. Figure 1-1 illustrates the two output products of the data structure handler for a representative flowchart construct. Section 3.2 discusses the organization and functions of the data structure handler.

1.3.2 Graphical Debugging

With a system that can generate flowcharts and the equivalent Pascal source instructions, a debug processor could be developed that would provide valuable assistance to the programmer by displaying the portion of the flowcharts currently being executed by the Pascal program.
Is test1 true?  
\[ \begin{align*} 
& \text{yes} \\
& \text{Block1} \\
& \quad \downarrow \\
& \text{Block2} \\
& \quad \downarrow \\
& \text{Flowchart representation} \\
& \text{Pascal representation} \\
& \text{if test1 true then} \\
& \text{begin} \\
& \text{(Block1);} \\
& \text{end} \\
& \text{else} \\
& \text{begin} \\
& \text{(Block2);} \\
& \text{end;}
\end{align*} \]

Figure 1-1: The If-Then-Else Construct

Highlighting techniques might be employed to follow the execution through decision paths or through a series of procedure calls or computations. The debug capability will be discussed in more detail in section 5.4.3.

1.3.3 Facilitating Software Maintenance

Software maintenance is often the largest element of total computer system life cycle costs [3]. The associated expenditure could be greatly reduced by employing the proposed software development method. Because the Pascal source statements will be generated along with the flowcharts, the final software product will always be accompanied by the latest version of the flowcharts. The maintenance programmer would no longer have to study the flowcharts (hoping what he/she sees represents the latest version) to understand the code prior to making changes to the code (and changes to the flowcharts?). Instead he/she would study

3 Such techniques would involve changing the graphical representation of a vector to a different intensity or pattern, such as a diffused vector or a dotted line.
and revise the unified representation of the flowcharts and source code. This method of maintaining software should be especially valuable in environments where personnel arrive with diversified backgrounds and rotate rapidly to new positions.

As interesting and important as such advances in software maintenance may be, this study will not be able to evaluate the impact of the software development system on software maintenance. Such a study would conceivably require years to analyze. Although an evaluation of the usefulness of this system as a software system developing tool is feasible as a part of this research (and will be proposed in chapter 5), no extensive evaluation of software maintenance will be included.

1.4 Anticipated Benefits

The aim of this study is to design a system that will demonstrate the capabilities outlined previously and to implement as many of the capabilities as time will permit. It is expected that the following benefits could be realized if the system were expanded to include all the capabilities suggested. A method of verifying these assumed results is suggested in section 5.5.

1.4.1 Neat, Up-to-date Flowcharts

Every programmer can have at his/her disposal, with minimum effort and artistic ability, neat and accurate flowcharts before the first line of code is compiled. Furthermore, the iterative process of expanding flowcharts in a top-down manner as the design elements become clear can be accomplished automatically. We can thus eliminate the tedium of redrawing what has already been established. Perhaps this feature alone
would rekindle interest in using flowcharts.

1.4.2 Debugging

Another characteristic that could significantly decrease the occurrence of undetected errors is the capability to provide a graphical debug processor that would operate on the data structure. Whereas most debug processors operate on code in a linear manner, placing breakpoints at various locations, then allowing execution to continue line-by-line until the breakpoint is encountered, a graphical debugger could allow breakpoints to be established after any flowchart construct or assignment statement. As a result, programs could be debugged in much the same manner in which they were developed - in a top-down, modular fashion. The programmer could specify debugging at the highest levels of flowcharting, to check interaction among top-level modules, or at the lowest levels to confirm the smallest details of the system. This capability will not be designed in this investigation due to time limitations imposed, but section 5.4.3 will include a discussion of such a system.

1.4.3 Improved Software Structure

Structured programming has been credited for large gains in program correctness [16]. By using the flow-chart generator, the programmer will be restricted to using the basic if-then-else, do-while, and case constructs. Such restrictions will help assure a greater degree of software structure in all versions of the design and code. In addition to the construct restrictions, the process of refining flowcharts will result in strict adherence to the method of stepwise refinement advocated by Wirth [18].
1.4.4 More Reliable Software

Software system programmers should expect to produce more reliable systems by utilizing this flowcharting/coding system. Wirth assesses Pascal as a naturally reliable programming language [19]. Because the process of developing flowcharts is constrained in a manner that parallels Pascal's syntax, greater reliability can be initially incorporated. Since the programmer selects constructs which will simultaneously produce a flowchart picture and a block of code as in figure 1-1, the resulting code should more accurately represent the programmer's intent. For example, consider the nesting of an if-then construct within an if-then-else construct. Wirth pointed out that this may be interpreted ambiguously as an if-then-if-then-else construct: to which "if-then" does the final "else" belong [9]? The syntax of the Pascal language requires that the word "then" be followed by a compound statement instead of a statement. The ambiguity is demonstrated by figure 1-2 which shows a structured representation of both interpretations. The data structure handler would show the programmer (via the flowchart display) which else-segment was being filled in at that time. Referring back to figure 1-1 (page 4), if "Block 2" were an if-then construct, the programmer would have to explicitly end the void "else" segment before continuing with the outer else-segment.

1.5 Limitations

The flowcharting and coding system herein proposed is designed in a manner that includes some limitations that should be evaluated by the prospective user.

1.5.1 Familiarization with Pascal

This system will be most useful only to those programmers familiar
if A=B then
  if C=D then I := 4
  else J := 5
else J := 5

J = 5 if A = B and C ≠ D
j = 5 if A ≠ B

Figure 1-2: Ambiguity of an If-then-if-then-else Construct with Pascal or other ALGOL-like computer languages. The data structure handler calls for specific entries that correspond directly to the syntax of Pascal or ALGOL, such as completing the Boolean condition to be tested in an if-then-else statement. Although the interactive development of flowcharts would be helpful to a FORTRAN programmer, the source code output would be interspersed with invalid statements. It should be noted, however, that the data system structure handler could be easily modified to provide FORTRAN or other source language output.

1.5.2 System Resources Dependency

This system, as implemented, requires access to a Tektronix graphics terminal to develop flowcharts. Although the same abstractions in flow-chart development and in source file translation could be accomplished using standard line printer devices [11], no such development is attempted in this study.

Single-user access to a small computer with floppy-disk storage and with at least 16K bytes of central storage is required by this system as

The Data Structure Handler, except for certain Pascal cite implementation peculiarities, is device independent, but the graphics handlers of chapter 4 relate only to the Tektronix terminal
currently implemented. No discussion of generality or modifiability of this demonstration system for other computer configurations is offered.

1.5.3 Limited Pascal Capabilities

Due to the complexity of the project, no attempt will be made to develop a system that will allow all aspects of Pascal to be charted and translated to source code. Several permissible Pascal constructs, such as "repeat until", "with", and "goto", are not implemented because (1) any system can be described without these additions and (2) their inclusion would not materially contribute to the intent of this study.
2. Discussion of Related literature

The amount of literature relating to automation of flowcharts and code is remarkably scarce. Although Lanzano proposed a system to automate this process in 1970, no follow up development had been noticed by 1974 when Dr. Thomas E. Bell penned the forward to Lanzano's paper. The same seems to be true for the remainder of the decade. The automation of flowcharts by themselves is a frequent subject, but the bridge between flowcharts and code seems to be relegated to the programmer alone without automated assistance.

The following areas of discussion in the literature will be presented in the following four sections:

- a discussion of automated program documentation (section 2.1)
- a discussion of automating the relationship between flowcharts and code (section 2.2),
- a discussion of the relationships of maintenance and reliability of software systems to the total computer system life cycle (section 2.3)
- some specific background information concerning fundamentals of flowchart representations of programs (section 2.4).

2.1 Available Products for Automating Program Documentation

The amount of material describing various support programs that document code by producing flowcharts is impressive. Chapin has compiled a description of the historical development of over 40 such processors [4]. Most of these processors were developed for a specific machine or computer language during the 1960's.

Reifer and Trattner catalogued 70 different automated programmer aids, one of which is "'Flowcharter', a program used to show in detail the logical structure of a computer program" [14]. The authors describe
the use of such an aid as a product which represents program flow logic
and which can be compared against the original flowcharts designed to
represent the system. As examples of flowcharters, they offer AUTOFLOW
and FLOWGEN, which are relatively current commercial aids also
catalogued in Chapin.

2.2 The Relationship between Flowcharts and Code

Two main considerations of the relationship between flowcharts and
code are relevant to this thesis:

- Section 2.2.1 discusses the proposals by two senior
  programmers/managers, Lanzano of TRW Systems and Davis of
  Austin Development Center, to provide a tool that will
  automatically produce source code either from the flowchart or
  from some other representation of the flow chart.

- Section 2.2.2 discusses tools that programmers employ to
  synthesize their code into blocks or constructs.

2.2.1 Automating Flowcharts and Code

Lanzano, in her article referenced in chapter 1, proposed the
question which this research attempts to answer. In her discussion of
computer aided program development, she discusses a proposal to develop
"a system wherein the code and the final flow chart no longer appear as
computer aided design techniques, a translator would interpret the
geometries of the flow chart into source language, i.e. rectangles into
arithmetic statements, hexagons into calls, diamonds into "if"
statements, etc. Her proposed system required many specific geometries
which were strongly coupled to FORTRAN, including specific symbols for
loops, format statements, declarative statements, subroutine calls,
comments and exits. Graphical output would be to either a graphics
terminal, utilizing line-drawing techniques, or to typewriter terminals,
utilizing square brackets to enclose rectangles, "<" and ">", to enclose diamonds, etc. Updating of the previous edition of the program being developed would be accomplished by optical scanning devices, or some "alternative form of input would be made available". A capability would be included to produce a source language output for a compiler, such as punching a source deck.

Lanzano continued in this article to point out some projected benefits of such a proposal. Diagnostics would alert the programmer that some flowchart symbols remain unfilled. Type checking could be performed on data as output statements are being prepared (a format could appear as "TIME ####.###"). Program reliability would increase because "pictorial representations are considerably less error prone than word images". While analysts are normally required to "document the program", a tedious and laborious task, the proposed system would produce the desired documentation at any point in the development stage. An important result would be increased readability and reliability of the program.

Another opinion about automating flowcharts and code was presented by Davis in his discussion about ANS Standard X3. 5-1970 flowcharting. While the major emphasis of his letter concerns specific aspects of the Standard, he discusses a flowchart he prepared on an incremental plotter using the IRAFLO system he previously developed [6]:

"That system allows creation and storage of flowchart specifications in symbolic form, so that they may be modified, plotted, or even (in some hoped-for future) automatically translated to source language."

Davis further comments that "flowcharting is not dead -- though it is
certainly sleeping soundly", and he expresses delight in observing renewed interest in using flowcharts.

2.2.2 Conceptualization Relation between Flowcharts and Code

This author has long held to a technique of conceptualization with code that was assumed to be his own private practice. It involved drawing lines around his code to reflect control flow. Loops could then be easily identified by the scribbled-in lines, and goto's and subroutine returns were easier to identify. Although this practice was followed most frequently with assembly language code in the debugging stage, it was also common for this author to draw boxes around blocks in ALGOL or Pascal to isolate disjointed block structures. Such a practice of drawing control flow may be rather common among programmers, as pointed out by Woodward, Hennell and Hedley [20].

"At some stage most Fortran programmers will probably have laid out their program text in front of them and then proceeded to draw arrowed lines on one side of the text indicating where a jump occurs from one line of text to another.... Such a time honored procedure sometimes aids the programmer in following the flow of control through the program."

Although the intent of the authors was to develop a measure of control flow complexity, their approach does point out a crutch that programmers frequently reach for, namely, some means of collecting portions of code into a synthesized module and sketching in control flow relations with other modules.

Weiner [17] has developed a method of documenting assembly language code which further supports this contention. He proposes structuring the comments field in a manner that follows the rules of structured programming. The result is a column of assembly language code in
parallel with a column of comments which resemble ALGOL's structured programming. This documentation method, similar to the method quoted above, further implies that programmers are seeking a method of grouping and relating their linear code. Although structured programming accomplishes this grouping and relating to some extent, some programmers apparently want more such help. direction.

2.3 Discussion Concerning Software Maintenance and Reliability

2.3.1 Software Maintenance

Boehm [3] presented an excellent discussion of software maintenance in 1976. He pointed out that software maintenance, which contributed less than 10% of the total hardware-software costs in the early 1950's, increased to over 40% in the 1970's—and he predicts it will exceed 60% by 1985. It is not clear exactly how one might explain this change in proportionality: is it solely the gigantic decrease in the cost of hardware components or is it the complexity of the software systems that are being designed for extended use? Obviously, a blend of both is responsible, but the overwhelming conclusion should be that software maintenance should command a great deal of our attention in hardware and software design.

The amount of money being spent on software in the Department of Defense is staggering: $3 billion per year in 1975 [7]. If roughly half of this outlay is for software maintenance, then much effort should be directed toward providing tools for the software maintenance effort. Such a tool might be the new programming language Ada which has been developed to confront the currently defined problems in software maintenance (and reliability) [1].
2.3.2 Reliability

Considerations of reliability are important in the development of software systems. This investigation will demonstrate a system that should significantly improve software reliability as a byproduct of the graphical flowchart approach to program development. Wirth contends that the programming language Pascal aids the programmer significantly in the area of software reliability. Certain characteristics of the language increase clarity, contribute to transparent programming, distinguish between "types" and "variables", and facilitate file usage. He carefully distinguishes between "correctness" and "reliability". One of the requirements for a programming language to be reliable is that it "must rest on a foundation of simple, flexible, and neatly axiomatized features, comprising the basic structuring techniques of data and program" [19].

The claim for increased reliability of the proposed system is not attributable to Pascal alone. Rather, the process of generating flowcharts and refining them to the language statement level should increase reliability because of the requirement to employ top-down structured programming and stepwise refinement at every step of the development process.

2.4 Some Fundamental Concepts in Flowcharting

2.4.1 Two-Dimensional Grammars

Jackson has proposed a structured programming language utilizing two-dimensional grammars [8]. The graphical portion of this language has been used for several years at Oakland University. He points out that despite the appearance of two-dimensionality in structured
approaches to current languages, the code is still one-dimensional: the indentation provides only a superficial added dimension. Jackson proposes a language comprised of the three constructs illustrated in figure 2-1 and a pattern recognition process that scans the figures for syntactical evaluation.

Figure 2-1: Jackson's Basic Control Structures

Figure 2-2 shows a sample of Jackson's two-dimensional language and an ALGOL-like equivalent of the same program.

2.4.2 Limiting the Use of Constructs

In Jackson's proposal, only three constructs are used - sequence,
if-then-else, and while-do (figure 2-1). In the proposal of this investigation, four will be used: Jackson’s three, plus a case construct. Although programmers accustomed to the variety and power of current higher-order languages may rely on other constructs, this set is sufficient to represent an algorithm or any degree of complexity. Actually, fewer than these are needed in a minimum sufficient set of constructs. A proof has been offered by Ashcroft and Manna that establishes that any algorithm can be restructured to an equivalent algorithm utilizing two constructs: an assignment statement and a while statement [2].

2.4.3 A Structured Flowcharting Convention

A very simple and useful flowcharting convention was developed by Professors Oldehoeft and Roman at Arizona State University [13]. The convention provides a technique of structuring the flowchart in a manner that parallels the recommended programming structure. The structuring
of the flowchart is accomplished by disallowing any goto facility and by providing three basic flowchart constructs, shown in figures 3-1, 3-2, and 3-3. This convention was required for use in all programming courses as an aid in teaching program structure prior to developing code in any language. As a student, this author experienced enormous gains in program correctness and debugging ease at the expense of a few days of frustrations with the flowcharting restrictions.

The next chapter discusses the approach used to generate these structured flowchart constructs, group them into a meaningful program representation according to the programmer's selections, and control the output of flowcharts and source code.
3. The Automatic Generation of Flow Charts and Source Code

Chapters 1 and 2 discussed the motivation for this study and summarized some of the observations and proposals presented in the literature. Having noted the lack of automated tools for flowcharting and producing the related code, an effort is made in this study to create such a software system. This chapter includes a discussion of the accomplishments toward the overall objective, along with the accomplishments that were intended but due to time limitations can now only be proposed for further study.

3.1 The PDP/Tektronix Graphics System

In order to demonstrate interactive flowchart development as a system design tool, an initial selection of computer and peripheral systems had to be made. For reasons of accessibility, the PDP-11/10, along with the Tektronix 4014, was chosen. Both devices were readily available in the Digital Engineering Laboratory of AFIT, although software support (such as handler programs for the graphics terminal) was limited. In order to facilitate development work involving the graphics terminal, a series of handler programs had to be written.

3.1.1 Documentation of the Graphics Handler Routines

The handler routines were developed to provide simple line drawing and figure management modules that could be easily accessed by the data-structure handler described in section 3.2 below. Chapter 4 includes a separate report on the graphics system development which

5

In retrospect, this was a poor choice, predicated on an assumption that UDSC Pascal would be operational on the PDP-11. See section 5.4.2 for recommended device choices for further studies.
began as a separate introductory course and was then expanded for this investigation.

3.1.2 Stored Flowchart Figures

The graphics system that evolved from the Tektronix handler programs allows creating, storing and retrieving graphical figures using the floppy disk for auxiliary storage. The three flowchart construct types illustrated in figures 3-1, 3-2, and 3-3 were generated and stored on floppy disk for use by the data structure handler described in section 3.2. For a discussion of why these three constructs were chosen, see section 2.4.2.

![Flowchart representation](image)

Flowchart representation

Pascal representation

Figure 3-1: The If-then-else Construct

3.2 The Data Structure Handler

The function of the data structure handler is to monitor the system development with the aim of collecting all of the programmer's selections into flowcharts or Pascal source code. The data structure handler controls the process of presenting menus to the designer, regulates the flowchart symbols, maintains a linked list of the designer's choices (see figure 3-5), and manages transfers of linked
lists to and from disk storage. The data storage representation will be discussed next, with a functional explanation in section 3.2.2 of the options available to the system user.

3.2.1 Data Storage Representation

In the data storage representation, a "record" is a unit made up of the four elements shown in figure 3-4. These elements correspond to the description of the components of "logrec" defined in appendix C.

Each of these records is linked together with the previous and
following record as illustrated in figure 3-5. This figure also shows a sample program described by representative codes and statements.

Figure 3-5 shows several records containing links, codes and texts. The codes are precisely the options that may be selected during the development process discussed in section 3.2.2. "Text" fields are those entries solicited from the user or those entries which can be automatically provided by the system. Table 3-0 lists, for each possible type of entry, the code associated with the entry, the text field (either the programmer's input or the system's automatic entries), and the formatting done prior to providing source output.
Table 3-1: Storage and Output Representations of Entry Types

<table>
<thead>
<tr>
<th>TYPE OF ENTRY</th>
<th>CODE</th>
<th>TEXT FIELD</th>
<th>OUTPUT (note 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heading</td>
<td>H</td>
<td>&lt;input&gt;</td>
<td>&lt;input&gt;</td>
</tr>
<tr>
<td>Statement</td>
<td>S</td>
<td>&lt;input&gt;</td>
<td>&lt;input&gt;</td>
</tr>
<tr>
<td>Constant</td>
<td>C</td>
<td>&lt;input&gt;</td>
<td>&lt;input&gt;</td>
</tr>
<tr>
<td>Type</td>
<td>T</td>
<td>&lt;input&gt;</td>
<td>&lt;input&gt;</td>
</tr>
<tr>
<td>Variable</td>
<td>V</td>
<td>&lt;input&gt;</td>
<td>&lt;input&gt;</td>
</tr>
<tr>
<td>Block</td>
<td>B</td>
<td>&quot;Begin&quot;</td>
<td>&quot;Begin&quot;</td>
</tr>
<tr>
<td>If-then-else</td>
<td>I</td>
<td>&lt;input&gt;</td>
<td>&lt;input&gt;</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>&quot;Else&quot;</td>
<td>&quot;Else&quot;</td>
</tr>
<tr>
<td>While-Do</td>
<td>W</td>
<td>&lt;input&gt;</td>
<td>&quot;While &lt;input&gt; do&quot;</td>
</tr>
<tr>
<td>Case</td>
<td>C</td>
<td>&lt;input&gt;</td>
<td>&quot;Case &lt;input&gt; of&quot;</td>
</tr>
<tr>
<td>Case list element</td>
<td>:</td>
<td>&lt;input&gt;</td>
<td>&quot;'&quot;&lt;input&gt;'&quot;:&quot;</td>
</tr>
<tr>
<td>End</td>
<td>E</td>
<td>&quot;End&quot;</td>
<td>&quot;End&quot;</td>
</tr>
</tbody>
</table>

1. Trailing semicolon added when appropriate.
2. If-then-else results in two separate data records.

3.2.2 Explanation of the Handler’s Commands

The data structure handler was designed to be totally self documenting. Therefore, at any time the designer is prompted for input, the data structure handler provides a menu of the options that are allowed at that point. The menu is displayed by typing "?". At the highest level (the executive or entry level) the following options are displayed.

1. Create a new system description.
2. Get a system description from disk storage.
3. Edit old system description.
4. Save the current description on disk storage.
5. Produce Pascal source output.
6. Produce flowchart drawing.
7. Exit - return to monitor.
Each of these choices will be expanded in the following paragraphs. Expansions beyond this next level are not included in this report due to the extent of laborious detail. Interested readers can find a representation of calling priorities and module relationships in the structured design offered in appendix A or in the flowcharts included in appendix B. Additionally, the code is included in appendix C.

3.2.2.1 Creating a New System Description

This option allows the programmer to begin designing his system "from scratch". It assumes nothing is pre-established - similar to the programmer looking at a blank coding form. The options allowed at this point (again, available to the programmer by typing "?") include:

- Heading  
- Block  
- Constant definition  
- Type definition  
- Variable declaration  
- Statement

The "block" option has its own menu and includes options to select any of the three flowchart constructs (if-then-else, do-while, and case) depicted in figures 3-1 through 3-3. In turn, each of the three constructs allows for termination of the construct or recursively selecting either another "block" or any of the other three constructs. For a more complete illustration of the options and a representation of their relative calling hierarchy, see appendix A.

With the capabilities thus far described, a designer could generate a Pascal program of any degree of sophistication. Although certain
Pascal features were not implemented into specific constructs (see section 1.5.3), all others can be directly implemented with these options along with the "statement" option which allows straight (unmodified) insertion of text. More specifically, comment lines, labels, and even goto's can be introduced into the system. However, by inserting goto's or other structures by using the "statement" option, the code will appear without its associated control flow in the flowchart representation.

3.2.2.2 Getting a System Description from Disk

The second option listed in the preliminary menu is to recall a system that was previously developed and then stored on disk. By selecting this option, the designer will recall the file defined during the system load process for the DEC-10 system (the user defines INPUT and OUTPUT prior to execution) or the file defined by the RESET command for the LSI-11 system. The content of the file would include the second and third columns of table 3-0 for each entry previously selected and each selection (record) would be linked to the previous and next record as they are read in. See figure 3-5 for a representation of the data and linkages.

3.2.2.3 Editing the System Description

Once a previous system description is recalled from disk, or at some time during the initial creation stage, editing may be performed on the current data structure. Several editing options have been included to allow altering specific records in the data structure. The following record-oriented editing commands are available.

- Insert
3.2.2.4 Saving the Description on Disk

When the designer has completed creating and editing a system description, he/she can select the option to save the data structure on disk. As was the case with getting a description from disk, the only file option for saving must be the file defined in response to the system’s "OUTPUT" query at load time (DEC-10), or the file defined by the REWRITE command (LSI-11).

3.2.2.5 Producing Pascal Source Output

This option allows the system to produce compiler-ready source code from the system described in the data structure. For the demonstration purposes of this study, the output is directed to the terminal rather than another disk file. When this option is chosen, indenting is automatically provided and punctuation (semicolons and periods) are properly inserted. The proper Pascal reserved words are inserted in their places within each of the three constructs.

3.2.2.6 Producing Flowchart Drawings

This option was not developed, but was included as a stub for later expansion.

3.2.2.7 Exiting the Handler Routine

By selecting this option, return to the system monitor is provided. No checking is done for saving files, thus "save" needs to be considered prior to exiting.
4. The PDP-11/Tektronix Graph Drawing System

This chapter discusses the basic graphics handler routines and figure management modules that were developed to provide easier utilization of the graphics terminal for this investigation and other laboratory uses. The data structure handler, discussed in section 3.2, can be augmented to utilize these modules for displaying the flowchart figures. The remainder of this chapter was originally written and submitted as a separate laboratory study.

4.1 Introduction

The objective of this software project was to develop a set of software modules that would facilitate creating graphical figures in the AFIT Microprocessor Laboratory. The driving commands required by the graphic terminal had to be interfaced with an understandable set of user instructions; manipulating tools had to be made available so that the user could alter the configuration of his graphical creation; and a capability had to be added that would allow the user to store his newly created figure on floppy disk and to recall the figure from the disk for display or alteration.

4.2 Equipment

The minicomputer used for this project was the PDP-11 model 10, with a Tektronix model 4014 graphics display terminal.

4.3 Running the graph system

The system is initiated by loading the floppy disk (laboratory control #65-22) in disk drive #0 and typing "RUN GRAPH". The terminal will immediately list the options itemized in 4.4 below.
4.4 Functional description of the system

The graphic system is mostly self-documenting, i.e. help is provided via either an executive command menu or a draw command menu. The executive menu describes which functions of the graph system may be activated; the draw command menu explains each draw command allowed in the "draw" mode. Upon entering the system six options, each of which will be expanded in the following paragraphs, are displayed.

1. Draw
2. Retrieve from disk and initialize
3. Retrieve from disk and append
4. Store present figure on disk
5. Help with draw command options
6. Exit nicely

4.4.1 Draw a new figure

Upon choosing option 1, the computer forces the terminal into an initialization sequence which erases the screen, rings a bell, and readies the terminal for graphical input. Two cross-hairs appear. The intersection defines an xy-pair to which a vector is drawn after typing in the appropriate character. The valid characters that may be used to draw pictures, or to alter them (itemized by selecting option 5) are the following.

4.4.1.1 Vector Drawing Commands
A  
Insert alpha string (terminate string with "ESC")

M
Move cursor to new cross-hair position (XHP)

P
Draw a point at new XHP

D
Draw a solid line to new XHP

.
Draw a dotted line to new XHP

-
Draw a dashed line to new XHP

B
Back up to previous vector

Q
Quit drawing - mark end of picture table in core

4.4.1.2 Figure Handling Commands

B
Back up to previous vector

R
Redraw picture from present core table pointer to quit entry

S
Step one vector (redraw, but draw one vector at a time)

T
Translate geometrically to new XHP (all remaining vectors) - requires striking a second character after cross-hairs are positioned as desired.

Q
Quit - mark end of table - exit draw mode

4.4.2 Draw from disk file

By selecting option 2, the computer will search for a file with the device, name, and extension provided by the user. The table of xy-pairs and line types (see table 4-1) will then be copied from disk into core at the address of "TBLE" in the main program, overwriting any previous information stored there.

4.4.3 Append from disk file

Option 3 performs the same function as option 2, but the new figure
Table 4-1: Format of Data Table Description

<table>
<thead>
<tr>
<th>Location in &quot;TBLE&quot;</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD 0</td>
<td>X(0) VALUE</td>
</tr>
<tr>
<td>WORD 1</td>
<td>Y(0) VALUE</td>
</tr>
<tr>
<td>WORD 2</td>
<td>MODE(0)</td>
</tr>
<tr>
<td>WORD 3</td>
<td>X(1) VALUE</td>
</tr>
<tr>
<td>WORD 4</td>
<td>Y(1) VALUE</td>
</tr>
<tr>
<td>WORD 5</td>
<td>MODE(1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>WORD 3n-3</td>
<td>X(n) VALUE</td>
</tr>
<tr>
<td>WORD 3n-2</td>
<td>Y(n) VALUE</td>
</tr>
<tr>
<td>WORD 3n-1</td>
<td>MODE(n)</td>
</tr>
<tr>
<td>WORD 3n</td>
<td>X(n+1) VALUE</td>
</tr>
<tr>
<td>WORD 3n+1</td>
<td>Y(n+1) VALUE</td>
</tr>
<tr>
<td>WORD 3n+2</td>
<td>4 (quit)</td>
</tr>
</tbody>
</table>

From disk is appended onto the one already in core. The previous "quit" mark is overwritten with the first move or draw of the disk figure.

4.4.4 Store on disk file

By selecting option 4, the table of xy-pairs and the line types corresponding to the figure which has been created thus far will be stored on the specified disk according to the file name specified by the user. Previous information in that table will be destroyed.
4.4.5 Explain "DRAW" commands

If option 5 is selected, a menu of all available draw commands is displayed with a terse explanation of what they accomplish. The user is then asked if he/she wants more information. If the reply is yes, the program asks which command is to be clarified. The system then elaborates on this command.

4.4.6 Exit graph system

Option 6 allows for the orderly termination of the program and for returning control to the system monitor.

4.5 System design notes

The detailed assembly language code is included as appendix F. Some user hints and recommendations for use of the system — and for system enhancements for the enterprising reader — are included in appendix G. The structure diagram of the graph system is included in appendix D. The flowcharts are in appendix E.

4.6 File control

Figure 4-2 contains a summary of the location of source, relocatable (object), and executable files relevant to the development of this system. For the DEC10 system, files may be found under programmer/project number [6664,146].

4.7 Acknowledgement

Most of the modules to control graphic terminal states and vector drawing were contributed by Professor Ross. Professor Hartrum provided the subroutine to pack file names in radix-50 format and to handle information exchange between the disks and core.
Table 4-2: File Control List

<table>
<thead>
<tr>
<th>FILE CONTENT</th>
<th>NAME</th>
<th>DISK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Program</td>
<td>GRAPH.MAC</td>
<td>65-24</td>
</tr>
<tr>
<td></td>
<td>MSGS.MAC</td>
<td>65-24</td>
</tr>
<tr>
<td></td>
<td>TOMLIB.MAC</td>
<td>65-24</td>
</tr>
<tr>
<td>Source backup, version n</td>
<td>GRn.MAC</td>
<td>65-22</td>
</tr>
<tr>
<td>Compiled Object Code</td>
<td>GRAPH.OBJ</td>
<td>65-24</td>
</tr>
<tr>
<td>Executable Code</td>
<td>GRAPH.SAV</td>
<td>65-22</td>
</tr>
<tr>
<td>Available Pictures</td>
<td>filnam.PIX</td>
<td>65-23</td>
</tr>
<tr>
<td>Documentation for Upgrading</td>
<td>HINTS.MSS</td>
<td>DEC10</td>
</tr>
<tr>
<td>Text for this lab report</td>
<td>LABDOC.MSS</td>
<td>DEC10</td>
</tr>
</tbody>
</table>

4.8 Critique

Several not-so-difficult modifications would greatly enhance the capability of this system. These changes are outlined in appendix G. With these changes the system would very nicely handle such jobs as electronic circuit design or flowcharting.

This system is severely limited by not having the capability to produce hard copies of the graphic drawings. Priority should be given to acquiring a hard copy device to print copies of the graphic display’s output.

The shared printer is difficult to use. The procedure of unplugging the cable connected to the other lab devices and plugging in the correct one is time consuming and the cable is difficult to reach. The cable’s plug is subject to damage when it is pulled from the printer because it is so difficult to access. Recommend a box be constructed that will allow dial-type switching among computers connected to the line printer.
5. Results and Recommendations

In the previous two chapters, the software systems were described that managed the data structure (chapter 3) and provided an interface to the graphics terminal (chapter 4). This chapter will present a critique of some of the detailed accomplishments and recommendations for further development of the overall system.

5.1 Overall accomplishment

The systems discussed in chapter 4 demonstrate that a system design tool could be developed that would allow creating Pascal programs by successively refining flowcharts. Although the proposed system was not developed enough to perform an actual demonstration, sufficient progress was made to point to the structure and content of such a system and to encourage continued development of the system in a follow-on study.

5.2 The Graphic Handlers

The handler routines for the PDP-11/Tektronix 4014 system were described in chapter 4 and are included as appendix F. These routines provide a good facility for drawing flowcharts and for storing, recalling and modifying these flowcharts.

5.2.1 Critique

A detailed critique of the graphic handlers is presented in section 4.8 and appendix G.

Although the handlers were designed primarily to produce flowcharts, they also perform the same operations for any graphical figure, manually or automatically drawn (drawn with the output of a separate computing routine.)
5.2.2 Recommendations

The structure of the handler programs, as can be verified by studying the structure charts in appendix D, is awkward. Three people contributed to the final product, each with slightly different intentions. The handler routines should be revised if any of the following applies:

- Pascal is implemented on the PDP-11 (the redesign to implement graphic control using handlers written in Pascal would be extremely simple and flexible)

- The graphic handlers are transported to another device, such as the DEC 10 (the modifications needed for the new system might approach the effort required to redesign and rewrite)

- Considerably more modifications of the graphic handlers are anticipated.

Additional recommendations pertaining to the graphics handlers are included in appendix G.

5.3 The Data Structure System

The data structure handlers (chapter 3 and appendix C) provide a simple interface between the programmer/designer and the design system. The interface provides a medium in its data structure to describe the system created by the programmer/designer; stores, retrieves, and manages modification of this description; and produces from the data structure description a compiler-ready source code listing.

These structure charts were constructed according to the guidelines of Constantine and Yourdan who also explain methods of analysing structure to detect poor design [5].
5.3.1 Critique

The data structure handler was designed much more carefully than the graphics handlers and should be simple to increase capabilities or alter present features.

The system provides a chain of prompt messages that gives the programmer a history of where he has been in his design process. For instance, if a programmer selects the options "create", "block", and "while-do", the next prompt will be "CreBlkWdo>", thus confirming that the programmer is building the "while-do" construct. If one of the choices in the while-do construct is an if-then-else statement, the next prompt will be "CreBlkWdoIte". Although this capability was originally added as an aid in designing the data structure handler, it has proved to be a valuable tool for reminding the programmer where he is in the design process.

Each statement that is to be entered within a construct must be called for by selecting the "s" option. This action is easy to forget. When the system expects an option entry, it has frequently read the text of the entry instead, thus errors or inconveniences are frequently introduced. The option is not absolutely essential in the design of the system, but the only alternative would be a complex parsing system to identify each construct. The choice was therefore made to use option characters and suffer the trade-off requirements of patience and extra editing.

The editor provides only limited capabilities to change the data file. Changes can only be made one line at a time. Since programmers frequently delete or add entire blocks or constructs, the capabilities
of the editor do not closely match the needs of the user.

5.3.2 Recommendations

The most immediate - and simple - alteration would be to allow for mass addition or deletion of blocks or constructs of code within the editor. This might be accomplished by differentiating between upper and lower case options for construct vs. line changes.

More complex changes could be made to develop a useful Pascal preprocessing capability. The system could detect unmatched "begin" and "end" statements (although it would be nearly impossible for such a situation to result when using the data structure handler). Additionally, the system could perform scanning to determine undeclared variables prior to submitting the code to the compiler.

5.4 Recommendations for Further Development

While the previous paragraphs discuss relatively simple changes to the data structure handler only, the following recommendations pertain to further development of the flowchart generating system as a whole.

5.4.1 Combining the Graphics and Data Structure Handlers

In order to adequately demonstrate a design tool that could generate flowcharts and source code, the Data Structure Handler must be able to manage the graphics system. This capability was included in the design via modules that would allow external programs to call the graphic handlers and perform drawing of an externally stored data structure (modules FRDAW AND MDRAW). This was not completed in this investigation due to the limitation of time and several erroneous assumptions. Some of these assumptions were
- Pascal would be available on the PDP-11 system during the development of this investigation.

- The investigator's method of dual backups of critical files would be sufficient to withstand any reasonable attack by the operating system.

- If the PDP-11 would not suffice for the project, the software could be transported to the LSI-11 or the DEC-10 with relative ease.

Because of these errors, the two handler systems were never implemented on the same computer. The graphics handlers were completed on the PDP-11 while the data structure handler was completed on the DEC-10.

Thus, to further study the usefulness of the proposed system, both handlers must be implemented on one system. Appropriate calls from with the data structure handler should perform the drawing of the selected constructs. With a terminal with a large display, the interactive exchanges between the program and the programmer can be shown on one side of the screen, while the flowchart can be constructed automatically on the other. For smaller display devices, the flowchart may be postponed until the user opts to draw.

The nesting of flowcharts might best be managed by using a naming convention similar to the Structured Analysis and Design Technique [15]. When space limitation on the screen would prevent displaying the current construct, this construct would be represented in the embedding flowchart as a block reference. Block reference names, such as Al-5, would identify flowchart-5 (a block or construct) as a subunit or descendant of Al.

5.4.2 Chosing a New Host System

Among the systems that were available for this investigation, the
following substantiated choices are recommended in the order listed.

- DEC-10 (AF Avionics Laboratory) with DECGRAPHIC11 or other graphic system

  * All software included in this investigation is catalogued on this system
  * Pascal is well documented and supported
  * A cross-assembler, MACY11, is available for RT-11 modules.

- LSI-11 with Tektronix 4014 terminal

  * Although this system may be reserved for projects requiring embedded systems, this would be the next best choice
  * UCSD Pascal is not as well implemented as on the DEC-10
  * A separate version of the data structure handler was developed for the LSI-11 and is available on floppy disk number 34-64
  * Line printer capabilities on this system are very limited.

- PDP-11 with Tektronix 4014 terminal

  * This is not a reasonable alternative if Pascal is not implemented on the PDP-11
  * Neither version of the data structure handler is available for this system.

5.4.3 Adding a Debug Capability

This investigator believes that a great potential may exist in the form of a debug processor built around the flowcharting system. If a programmer designs his system using successively refined flowcharts and compiles the output code of the same system, it would be extremely helpful for him to be able to follow the execution of his program
directly on the flow charts. A similar capability exists today on many computer systems, utilizing control facilities of a "trace" processor. The trace processor maintains a list of which variables the programmer wants dumped or which modules traced and allows execution of the program to continue to a recognizable place in the code (i.e. a specific line number). In a similar manner, execution of the program could be allowed up to a certain block or construct, and the programmer could follow highlighting traces of the program's progress. If an incorrect branch is taken, the programmer could immediately spot where it occurred and what logic error caused it. Control variables or Boolean operators could be changed to test the correction. An option within the debug processor could call for all test changes to be applied to the input data structure, thus updating the flowcharts and the source input code to match the debug-tested version.

5.5 Recommended Evaluation

The proof of any claim of usefulness of this software design tool lies in a thorough evaluation. A separate investigation, when the above enhancements are complete, should be made with an organization which produces a large volume of Pascal (or ALGOL). Such a study should be aimed at the general features of software engineering referred to in this investigation, i.e., structure, reliability, and software maintenance.

5.6 Summary of Results and Recommendations

The concept of developing detailed software by stepwise refinement of flowcharts is feasible and attainable even though the results of the work put into this investigation does not clearly demonstrate it. A follow-on thesis should advance the development of this investigation as
outlined above. At the completion of this development, the system should be thoroughly evaluated to assess its effect on software structure, reliability, and maintainability.
Appendix A. Structured Design of the Data Structure Handler

Diagram: 
- EXBC
- OPT
- GETOPT
- EXECMENU
- CREATENEW
- GETFILE
- SAVEFILE
- EDI
- OUT
- DRAWFC
- STMT
- BLK
- CREATEMENU
Appendix B. Flowcharts of the Data Structure Handler

Start

QuitExec=0?

y

getopt

QuitExec = 1

exit

n

PutTxt

p.code = opt

p.txt = txtin

return
Insert
Print "insert before"
FindP
GetTxt
Insert
return

Append
p = head

p^.next <> nil ?

y

p = p^.next
oldp = p^.prev
CreateNew
return

Enter
oldp = p
new(p)
p^.code = opt
p^.text = txt
p^.next = nil
p^.prev = oldp

oldp = nil ?

y

head = p

oldp^.next = p

oldp = p
return
Depart

\[ (p^\cdot\text{prev} = \text{nil}) \text{ AND } (p^\cdot\text{next} = \text{nil})? \]
\[ n \]
\[ y \]
\[ \text{oldp} = \cdot \]
\[ \text{head} = \text{nil} \]
\[ p = \text{nil} \]

\[ (p^\cdot\text{prev} = \text{nil}) \text{ AND } (p^\cdot\text{next} <> \text{nil})? \]
\[ n \]
\[ y \]
\[ \text{oldp} = \text{nil} \]
\[ p^\cdot\text{next} = \text{nil} \]
\[ p^\cdot\text{next} = \text{oldp} \]
\[ \text{head} = p^\cdot\text{next} \]
\[ p = p^\cdot\text{next} \]

\[ p^\cdot\text{next} <> \text{nil}? \]
\[ n \]
\[ y \]
\[ \text{oldp} = \text{nil} \]
\[ \text{oldp}^\cdot\text{next} = p^\cdot\text{next} \]
\[ \text{oldp} = \text{nil} \]
\[ \text{oldp} = p^\cdot\text{prev} \]

\[ \text{return} \]

Replace

\[ p <> \text{nil}? \]
\[ n \]
\[ y \]
\[ \text{GetTxt} \]
\[ \text{PutTxt} \]
\[ \text{Return} \]

\[ 51 \]
Insert
  new(p)  n
oldp <> nil ?
      y
      p^.next = oldp^.next
      p^.prev = oldp
      oldp^.next = p
      p^.next^.prev = p
      head^.prev = p
      p^.next = head
      p^.prev = nil
      head = p
      PutTxt
      Return

If-Then-Else
  tracer('Ite')
  write "'if' test:'
  read txtin
  enter
  opt = 'b'
  write "'then' block"
  block
  opt = 'b'
  write "'else' block"
  block
  return

WhileDo
  tracer('Wdo')
  write "'while' text:'
  read txtin
  enter
  opt = 'w'
  write "'while-do block:'
  opt = 'b'
  block
  return

52
CListBlock

Write "Case label list"
read txtin:i (*count chars*)

n
i > 0 ?
  
  opt = 'i'
  enter
  CreateNew
  CListBlock
  return

CaseOf

Tracer('Cas')
write "case expression:"
read txtin
  enter
  CListBlock
  return
PutCode

\[ \text{tab} = 0 \]
\[ \text{p} = \text{head} \]

\[ \text{p} \not= \text{nil} \]

\[ \text{tab} = \text{tab} + 1 \]
\[ \text{indent} \]
\[ \text{write(p}.\text{txt}) \]

\[ \text{indent} \]
\[ \text{write("while ",p}.\text{txt},"," do") \]

\[ \text{indent} \]
\[ \text{write("if ",p}.\text{txt}," then") \]

\[ \text{indent} \]
\[ \text{write(txt)} \]

\[ \text{indent} \]
\[ \text{write("case ",txt," of")} \]

\[ \text{indent} \]
\[ \text{write("","",txt," ",")} \]

\[ \text{indent} \]
\[ \text{p}.\text{next}.\text{code} \not= 1 ? \]

\[ \text{write(p}.\text{txt},";")) \]
\[ \text{write(p}.\text{txt}) \]

\[ \text{p}.\text{next} = \text{nil} ? \]

\[ \text{write(txt,";")) \]
\[ \text{write(txt,"."")} \]

\[ \text{tab} = \text{tab} - 1 \]

\[ \text{return} \]
Appendix C. Listing of the DEC-10 Data Structure Handler Program

Program DataStructureHandler(Input,Output);

const linelength = 40;
type    link = "logrec;
   str = packed array [1..linelength] of char;
   prompt = packed array [1..3] of char;
   logrec = record
      code : char;
      txt : str;
      prev : link;
      next : link
   end;
var head, p, oldp : link;
txtin : str;
opt : char;
i,k,quitExec : integer;
tracer : array [1..15] of prompt;
nextpr : prompt;
Exc,Cre,Blk,
Ite,Wdo,Cas,
Edt,Rpl,Sav,Get : prompt;

procedure intro;
begin
   writeln(tty,' <<< DSH >>>');
   writeln(tty,' Data Structure Handler');
   writeln(tty,'');
   writeln(tty,'For a menu, type "?" after the prompt "">"');
   writeln(tty,'');
   (**
   (* Set up prompt equates *)
   (**)
   Exc := 'Exc'; Cr := 'Cre'; Blk := 'Blk'; Ite := 'Ite';
   Wdo := 'Wdo'; Cas := 'Cas'; Edt := 'Edt'; Rpl := 'Rpl';
   Sav := 'Sav'; Get := 'Get';
   end;
procedure CreateMenu;
begin
  writeln(tty,'[H] Heading');
  writeln(tty,'[K] Constant definition');
  writeln(tty,'[B] Block');
  writeln(tty,'[T] Type definition');
  writeln(tty,'[S] Statement');
  writeln(tty,'[V] Variable declaration');
  writeln(tty,'[X] exit to Exec');
end;

procedure ExecMenu;
begin
  writeln(tty,'[C] Create new system description.');
  writeln(tty,'[G] Get a system description from device.');
  writeln(tty,'[E] Edit old system description.');
  writeln(tty,'[S] Save the current description on device.');
  writeln(tty,'[P] Produce Pascal source output.');
  writeln(tty,'[F] Produce flowchart drawing.');
  writeln(tty,'[X] Exit - return to monitor.');
end;

procedure EditMenu;
begin
  writeln(tty,'[D] Delete a record');
  writeln(tty,'[A] Append to list');
  writeln(tty,'[I] Insert a record');
  writeln(tty,'[R] Replace a record');
  writeln(tty,'[E] Erase previous record');
  writeln(tty,'[X] Exit EditOld');
end;

procedure BlockMenu;
begin
  writeln(tty,'[I] If-then-else construct');
  writeln(tty,'[S] Statement');
  writeln(tty,'[W] While-Do construct');
  writeln(tty,'[C] Case construct');
  writeln(tty,'[B] Back up one record');
  writeln(tty,'[E] End of Block');
end;

procedure TypeMenu;
begin
  writeln(tty,'[V] Variable Declaration');
  writeln(tty,'[C] Constant');
  writeln(tty,'[T] Type definition');
  writeln(tty,'[S] Statement');
  writeln(tty,'[E] End Type block');
end;

57
(**) (* Solicit and read text *)

procedure GetTxt;
begin
  writeln(tty,'text:');
  readln(tty);
  read(tty,txtini)
end;

(**) (* Load opt and txtin into their pointer file positions *)

procedure PutTxt;
begin
  p^code := opt;
  p^txt := txtin;
end;
function GetOpt: char;
begin
readln(tty);
read(tty,opt);
getopt := opt
end;

(* walk through the list until the desired record is found*)
(* return p=nil if end of list*)
procedure FindP;
var ans : char;
begin
ans := 'n';
p := head;
while (p<> nil) and ((ans = 'n') or (ans = ' ')) do
begin
writeln(tty,p^.code, ', ',p^.txt,' ... is this it? [y/n]');
readln(tty);
read(tty,ans);
if ans <> 'y' then p := p^.next
end;
if p <> nil then
oldp := p^.prev
else writeln(tty,'end of list found');
end;

(* Appends incoming text string (a prompt) to prompt vector *)
(* and puts prompt vector into I/O Buffer *)
Procedure PutTracer(nextpr : prompt);
var j : integer;
begin
k := k + 1;
tracer[k] := nextpr;
j := 1;
while j <= k do
begin
write(tty,tracer[j]);
j := j + 1;
end;
writeln(tty,'>');
end;
procedure enter;
begin
  oldp := p;
  new(p); (*point to new record*)
  PutTxt;
  p^.next := nil;
  p^.prev := oldp;
  if oldp = nil then
    head := p
  else oldp^.next := p;
end;

procedure depart;
begin
  if (p^.prev=nil) and (p^.next=nil) then
    begin (*case only one record exists*)
      oldp := nil;
      head := nil;
      p := nil
    end
  else if (p^.prev=nil) and (p^.next<>nil) then
    begin (*two records exist; delete 1st*)
      oldp := nil;
      p^.next^.prev := nil;
      head := p^.next;
      p := p^.next
    end
  else if p^.next <> nil then (*implied p^.prev<>nil*)
    begin (*comfortably in the middle*)
      oldp^.next := p^.next;
      p^.next^.prev := oldp;
      p := oldp^.next
    end
  else
    begin (*last record in list*)
      oldp^.next := nil;
      p := oldp;
      oldp := p^.prev
    end
end;

procedure Insert;
begin
  new(p);
  if oldp <> nil then
    begin (*normal insert in list*)
      p^.next := oldp^.next;
      p^.prev := oldp;
    end
end;
oldp^.next := p;
p^.next^.prev := p
end
else
begin
(* p points to first list elt *)
head^.prev := p;
p^.next := head;
p^.prev := nil;
head := p
end;
PutTxt;
end;

procedure replace;
begin
if p <> nil then
begin
PutTracer(Rpl); (* revise option entry*)
p^.code := getopt;
GetTxt;
PutTxt;
k := k-1;
end;
end;
procedure Statement;
begin
    GetTxt;
    enter;
end;

procedure CreateNew; forward;
procedure Block; forward;

procedure IfThenElse;
begin
    PutTracer(ite);
    write(tty,"'if'");
    GetTxt;
    enter;
    writeln(tty,"'then" block:');
    opt := 'b';
    Block; (*put a whole subprogram here, maybe*)
    opt := 'l'; (*option to flag the solo "else" in output*)
    txtin := 'else enter;
    writeln(tty,"'else" block:');
    opt := 'b';
    Block; (* again *)
k := k-1;
end;

procedure WhileDo;
begin
    PutTracer(Wdo);
    write(tty,"'While'");
    GetTxt;
    enter;
    writeln(tty,"'While-do" block:');
    Block;
k := k-1;
end;

procedure CaseOf;

procedure CListBlock;
var i : integer;
begin
    writeln(tty,'case label list:');
    readln(tty); i;
    read(tty,txtin:i);
    if i > 0 then
        begin
            opt := ':'; (*flag each case label list*)
            enter;
            Block;
            CListBlock;
        end;
end; (* Exit if a blank line is typed *)
begin
PutTracer(Cas);
writeln(tty,'>');
write(tty,"Case <expression> '");
GetTxt;
enter;
CListBlock;
\[ k := k-1; \]
end;

procedure EndBlock;
begin
  txtin := 'end
  enter;
end;

procedure Block;
var QuitBlock : integer;
begin
  opt := 'b'; (* force new option to 'b' *)
  txtin := 'begin
  enter;
  QuitBlock := 0;
  While QuitBlock = 0 do
    begin
      PutTracer(Blk);
      case getopt of
        's': statement;
        'w': WhileDo;
        'i': IfThenElse;
        'c': CaseOf;
        'e': QuitBlock := 1;
        '?': BlockMenu;
      end; (* Note UCSD and Dec 10 non-standard *)
    (* handling of undefined options *)
    \[ k := k-1; \]
  end;
  EndBlock;
end;
procedure CreateNew;
var quitCre : integer;
begin
quitCre := 0;
while quitCre = 0 do
begin
PutTracer(Cre);
case getopt of
    '?' : CreateMenu;
    'b' : Block;
    't','s','k','v','h' : Statement;
    'x' : quitCre := 1
end;
k := k-1;
end
end;

Procedure GetFile;
begin
p := nil;
while not eof(input) do
begin
    readln(input,opt,txtin);
    enter;
end;
end;

procedure EditOld;
var quitEdit : integer;
begin
quitEdit := 0;
while quitEdit = 0 do
begin
    PutTracer(Edt);
case getopt of
    '?' : EditMenu;
    'd' : begin
        FindP;
        if p<>nil then Depart;
        end;
    'i' : begin
        writeln(tty,'Insert before ...');
        FindP;
        writeln(tty,'new option:');
        opt := getopt;
        GetTxt;
        Insert;
        end;
    'a' : begin
        p := head;
        while p^.next<>nil do p:=p^.next;
        oldp := p^.prev;
        CreateNew;
        end;
end;
'r': begin
    FindP;
    replace;
end;
'b': replace;
'x': quitEdit := 1;
    end;
    k := k-1;
    end
end;
(* Save this data structure on floppy disk *)

procedure SaveFile;
begin
  PutTracer(Sav);
  p := head;
  while p <> nil do
  begin
    writeln(p^.code,p^.txt);
    p := p^.next;
  end;
  p := head;
  k := k-1;
end;

(* Put ASCII card images out to TTY *)

procedure PutCode;
const tabval = 8;
var tab : integer;

procedure Indent;
var j : integer;
begin
  j := tab;
  while j>0 do
  begin
    write(tty,' :8);
    j := j - 1;
  end;
end;

begin
  tab := 0;
  p := head;
  while p<> nil do
  begin
    case p^.code of
      's', 'h', 't', 'k', 'v': begin
        Indent;
        If p^.next^.code <> 'l' then
          writeln(p^.txt, ';');
        else writeln(p^.txt);
        end;
      'b': begin
        tab := tab + 1;
        Indent;
        writeln(tty,p^.txt);
      end;
      'e': begin
        Indent;
        If p^.next = nil then
          66
    end;
  end;
end;
```pascal
else if (p^.next^.code = 'l') or (p^.next^.code = 'e') then
  writeln(tty,p^.txt);
else writeln(tty,p^.txt,';');

end;

end;

procedure DrawFC;
begin
  writeln(tty,p^.txt,';');
  if (p^.next^.code = 'l') or (p^.next^.code = 'e') then
    writeln(tty,p^.txt);
  else writeln(tty,p^.txt,';');

  tab := tab - 1;
  end;

  'w':
  begin
    begin
      Indent;
      writeln(tty,'while ',p^.txt,' do');
      end;
  end;

  'i':
  begin
    begin
      Indent;
      writeln(tty,'if ',p^.txt,' then');
      end;
  end;

  'l':
  begin
    begin
      Indent;
      writeln(tty,p^.txt);
      end;
  end;

  'c':
  begin
    begin
      Indent;
      writeln(tty,'case ',p^.txt, ' of');
      end;

      p := p^.next;
      end;
  end;

  end;

  procedure DrawFC;
begin
  write(tty,'Exec-DrawFC-');
  writeln(tty);
  end;
```
begin
intro;
quitExec := 0;
while quitExec = 0 do
  begin
    k := 0;
    PutTracer(Exc);
    case getopt of
      '?': ExecMenu;
      'c': begin
        p := nil;
        CreateNew;
        end;
      'g': GetFile;
      'e': EditOld;
      's': SaveFile;
      'p': PutCode;
      'f': DrawFC;
      'x': quitExec := 1
    end
  end.
end.
Appendix E. Flowcharts of the Graph Drawing System

GrExec

Exit = true?

Gropts

GRDraw

GRRet

MDraw

GRRt2

Mraw

Draw

GRApnd

MDraw

GRSave

GRHelp

GRExit

exit

GRDraw

Init

Tekplo

R4 = #TBLE

Draw

return

73
Draw
Tekgin
read CHAR, XF, YF

Char
\[ r_2 = 1 \]
\[ r_2 = 2 \]
\[ r_2 = 3 \]
\[ r_2 = 5 \]
\[ r_2 = 6 \]
plot
\[ \text{transl} \]
\[ \text{REDR} = 1 \]
\[ \text{step} \]
\[ r_4 = r_4 - 6 \]

Set tty echo mode
\[ r_2 = 4 \]
\[ \text{fix tbl} \]
\[ \text{return} \]
Step

$XF, YF, MODE = \text{table values}$

$mode = 4 \ ?$

$\text{y}$

$r4 = r4 - 6$

$\text{bell}$

$\text{REDR} = 0$

$return$

$\text{tekpio}$

$\text{REDR} = 0 \ ?$

$\text{y}$

$return$

FixTbl

$mode = r2$

$\text{table values} = XF, YF, mode$

$return$

Trans1

$\text{save r4}$

$get XF, YF$

$XT = XF - XP$

$YT = YF - YP$

$mode <> 4 \ ?$

$\text{y}$

$XP = XP + XT$

$YP = YP + YT$

$\text{restore r4}$

$return$
Appendix F. Listing of the Graph Drawing System Program

.TITLE GRAPH GENERATING SYSTEM
.SBTL GREXEC - GRAPH EXECUTIVE MODULE

; THIS IS THE EXECUTIVE PROGRAM WHICH GOVERNS THE MODULES OF THE
; GRAPHICS SYSTEM. THE USER IS QUERIED BY THE OPTIONS MODULE
; (GROPTS) TO CHOOSE ONE OF THE FOLLOWING OPTIONS:
;
; 1 - DRAW A NEW PICTURE  4 - SAVE ON DISC
; 2 - RETRIEVE (& INITIALIZE)  5 - EXPLAIN DRAW COMMANDS
; 3 - RETRIEVE (APPEND TO FIX)  6 - EXIT TO RT-11 MONITOR
;
; THIS MODULE, AS THE EXEC FOR THE GRAPHICS SYSTEM, SIMPLY
; DIRECTS TRAFFIC TO ITS SUBORDINATES ACCORDING TO THE ABOVE
; OPTION. THE OPTION IS RETURNED TO EXEC AS A BINARY INTEGER
; AVAILABLE IN THE RO REGISTER.
;
.MCALL .V2.,.REGDEF,.EXIT,.TTYIN,.TTYOUT,.PRINT
.MCALL .TTINR
.GLOBL GROPTS,GRDRAW,GRRETR,GRSAVE,GRHELP,GREXIT
.REGDEF

GREXEC:

MOV  #0,RO          ;SAFETY FIRST
1$: JSR  PC,GROPTS   ;GET USER'S OPTION
CMP  RO,#1          ;IS OPT = 1 (DRAW) ?
BNE  2$           ; - NO
JSR  PC,GRDRAW      ; - YES
BR  1$

2$: CMP  RO,#2       ;IS OPT = 2 (RETRIEVE) ?
BNE  3$             ; - NO
JSR  PC,GRRETR      ; - YES
JSR  PC,MDRAW       ;QUICKLY REDRAW IT
JSR  PC,DRAW        ;GET INTO INPUT/ PLOT LOOP
BR  1$

3$: CMP  RO,#3       ;IS OPT = 3 (APPEND) ?
BNE  4$             ; - NO
SUB  #6,R4          ;REPLACE QUIT COMMAND WITH
MOV  #0,(R4)+        ; HOME-CURSER
MOV  #0,(R4)+        ; (DARK MOVE)
MOV  #2,(R4)+
MOV  R4,R3
JSR  PC,GRAPND
JSR  PC,MDRAW
JSR  PC,DRAW
BR  1$
4$:  CMP  RO,#4 ;IS OPT = 4 (SAVE) ?
    BNE  5$ ; - NO
    JSR  PC,GRSAVE ; - YES
    BR   1$

;; GREXIT RETURNS WITH RO = ZERO IF WE ARE INDEED READY TO EXIT 

5$:  CMP  RO,#5 ;IS IT 5 (HELP) ?
    BNE  6$ ; - NO
    JSR  PC,GRHELP
    BR   1$

6$:  CMP  RO,#6 ;IS OPT = 6 (QUIT) ?
    BNE  1$ ;SUSPECT KEYSTROKE ERROR
    JSR  PC,GREXIT
    CMP  RO,#0
    BNE  1$ ; - NOT READY TO EXIT

EXIT

.PAGE

;SBTTL GRDRAW - CONTROL THE GRAPH DRAW PROCESS

;*******************************  GRDRAW  *******************************

;*  *  THIS MODULE, CALLED BY GREXEC, CONTROLS THE DRAWING OF ALL
*  *  GRAPHICAL FIGURES. IT DOES NOT RECALL PREVIOUSLY DRAWN
*  *  FIGURES (SEE GRRTRV MODULE FOR THAT CAPABILITY).
*  *
;*******************************  GRDRAW  *******************************

;GLOBL  FDRAW,MDRAW
;GLOBL  TEKERA,TEKGRA,TEKPLO,TEKALP,BELL,REDRAW
;GLOBL  TEKGIN,XF,YF,MODE,LOX,GRDRAW,TBLE,INIT

LOX:  .WORD 0
XF:   .WORD 0
YF:   .WORD 0 ; STORAGE FOR X AND Y DESTINATIONS
MODE: .WORD 0

TEKPLO:
    ADD  #6,TBLEND ;BUMP END POINTER BY 3 WORDS
    CMP  MODE,#1 ;MODE 1 INDICATES A NORMAL DRAW
    BEQ  NOR
    CMP  MODE,#2 ;MODE 2 INDICATES A MOVE
    BEQ  MVE
    CMP  MODE,#3 ;MODE 3 INDICATES PLOT A POINT
    BEQ  PNT
    CMP  MODE,#5 ;MODE 5 INDICATES DOTTED LINES
    BEQ  DOT
    CMP  MODE,#6 ;MODE 6 INDICATES DASHED LINES
    BEQ  DASH
    JSR  PC,TEKALP ;GO ALPHA MODE
    MOV  MODE,RO ; - WITH CURSOR AT XHAIR
;--INITIALIZATION ROUTINE--

INIT: BIS #010000, 44

MOV #0, XF

MOV #0, YF

MOV #0, MODE

MOV #TBLE, TLBEND

JSR PC, TEKERA

JSR PC, TEKGRA

BR DRWVEC

--;--END OF INITIALIZATION--

--;--POINT PLOT------

PNT: MOV #34, RO

TTYOUT

BR DRWVEC

--;--DOTTED LINES-----

DOT: MOV #33, RO

TTYOUT

MOV #141, RO

TTYOUT

BR DRW

--;--DASHED LINES------

DASH: MOV #33, RO

TTYOUT

MOV #144, RO

TTYOUT

BR DRW

--;--NORMAL LINES------

NOR: MOV #33, RO

TTYOUT

MOV #140, RO

TTYOUT

BR DRW

--;--

DRW: JSR PC, TEKGRA

MOV LOX, RO

;TO DRAW, GO TO GRAPHICS

81
TTYOUT

; AND SENT LO X TO GET OUT OF DARK
BR DRWVEC

;----------SET UP FOR A DARK VECTOR (MOVE)--------
;
MVE: JSR PC,TEKGRA ; SET TO GRAPHICS MODE
;
;----------NOW COMMON FOR ANY VECTOR----------
;
DRWVEC: MOV YF,RO ; SET UP FOR HIY
MOV $5,R1
1$: ROR R0
DEC R0
BNE 1$
BIC #177740,RO ; MASK EXTRA BITS
BIS #40,RO ; AFFIX HIY PREAMBLE
TTYOUT ; OUTPUT HIY
;
MOV YF,RO ; GET LOY
BIC #177740,RO
BIS #140,RO ; PREAMBLE
TTYOUT ; OUTPUT LOY
;
MOV XF,RO ; GET HIX
MOV $5,R1
2$: ROR R1
DEC R0
BNE 2$
BIC #177740,RO
BIS #40,RO
TTYOUT ; OUTPUT HIX
;
MOV XF,RO ; GET LOX
BIC #177740,RO
BIS #100,RO
MOV RO,LOX
TTYOUT ; OUTPUT LOX
;
RTS PC
;
;
TEKGRA: ; ROUTINE TO GO TO GRAPHICS MODE
MOV $35,R0 ; CONTROL CHARACTER FOR GRAPHICS
TTYOUT
RTS PC
;
;
TEKERA: ; SUBROUTINE TO CLEAR THE SCREEN
.PRINT $38 ; OUTPUT CONTROL CHAR
MOV $6,R1
1$: MOV $77777,R2 ; WAIT LOOP FOR SCREEN TO CLEAR
2$: DEC R2
82
BNE 28
DEC R1
BNE 18
RTS PC
3$: .ASCII <33><14><7>
.ASCII 200
.EVEN

; TEKALP: ; ROUTINE TO GO TO ALPHA MODE
MOV #37,R0  ; PUT CONTROL CHAR IN RO
TTYOUT
RTS PC

; BELL: ; ROUTINE TO RING THE BELL
MOV #7,R0
TTYOUT
RTS PC

; TEKGIN: MOV #33,R0 TTYOUT
MOV #32,R0 TTYOUT
RTS PC

; CURADR: TTYIN
MOV R0,R1
BIC #177740,R1  ; FIRST COMPONENT IS HIGH BYTE
MOV #5,R2
10$: ROL R1
DEC R2
BNE 10$
TTYIN
BIC #177740,R0  ; LOW BYTE
BIS R0,R1
RTS PC

; INPVEC:
JSR PC,TEKGIN  ; GO TO GIN MODE
CMP CHAR,#124  ; T - PROMPT ANOTHER CHAR TO TRANSLATE
BEQ 10$
TTYIN

5$: MOV R0,CHAR  ; INPUT KEYSTROKE
JSR PC,CURADR  ; AND STORE IN CHAR
MOV R1,XF
BIS #100,R0
JSR PC,CURADR  ; GET CURSOR ADDRESS
MOV R1,YF
RTS PC

83
10$: BIS $100, 44 ; NO-WAITE 10
   .TTINR
   BCS 20$
   BR 5$
20$: .TTINR
   BCC 5$
   MOV $33, R0
   .TTYOUT
   MOV $160, R0
   .TTYOUT
   MOV $124, R0
   .TTYOUT
   MOV $131, R0
   .TTYOUT
   BR 20$
GRDRAW: JSR PC, INIT
   MOV $TBLE, R4 ; POINT R4 TO START OF TBLE
   ;
DRAW: JSR PC, INPVEC ; INPUT A VECTOR VIA TEKTRONIX
   ;
1$: CMP CHAR, #104 ; WAS IT A "D"?
   BNE 2$
   MOV #1, R2
   JSR PC, PLOT
   JMP DRAW
2$: CMP CHAR, #115 ; MOVE WITH A 'M'?
   BNE 3$
   MOV #2, R2
   JSR PC, PLOT
   JMP DRAW
3$: CMP CHAR, #120 ; PLOT A POINT WITH A 'P'?
   BNE 4$
   MOV #3, R2
   JSR PC, PLOT
   JMP DRAW
4$: CMP CHAR, #121 ; QUIT WITH A 'Q'?
   BNE 5$
   BIC $010000, 44
   ;
   MOV #4, R2
   JSR PC, FIXTBL
   RTS PC ; EXIT POINT FOR "DRAW"
5$: CMP CHAR, #56 ; DOTTED WITH A 'PERIOD'?
   BNE 6$
   MOV #5, R2
   JSR PC, PLOT
   JMP DRAW
6$: CMP CHAR, #55 ; DASHED WITH A '-'?
   BNE 7$
   ;
MOV  #6,R2
JSR  PC,PLOT
JMP  DRAW
;
7$:  CMP  CHAR,#123         ;S - STEP THRU OLD PIX TBLE
    BNE  8$
    JSR  PC,STEP
    JMP  DRAW

8$:  CMP  CHAR,#122         ;R - REDRAW FROM TABLE VALUES
    BNE  9$
    JSR  PC,REDRAW
    JMP  DRAW

9$:  CMP  CHAR,#102         ;B - BACK UP (DELETE) A COMMAND
    BNE  10$
    JSR  PC,BACKUP
    JMP  DRAW
;
10$:  CMP  CHAR,#124         ;T - TRANSLATE REMAINDER OF TBLE
    BNE  11$
    JSR  PC,TRANS
    JSR  PC,REDRAW
    JMP  DRAW

11$:  CMP  CHAR,#101         ;A - DO ALPHAS
    BNE  12$

111$: JSR  PC,INPVEC         ;GET NEW XF,YF,CHAR
    CMP  CHAR,#33            ;IS IT "ESC"?
    BEQ  12$
    ; YES - END OF CHAR STRING
    MOV  CHAR,R2
    JSR  PC,PLOT
    ;STUFF TBLE AND DO TEKPLO
    JMP  111$

12$:  JSR  PC,BELL           ;ANY OTHER - RING BELL
    JMP  DRAW
    ; AND TRY AGAIN
;
PLOT:  JSR  PC,FIXTBL
    JSR  PC,TEKPLO
    RTS  PC
;
FIXTBL: MOV  R2,MODE
    MOV  XF,(R4)+
    MOV  YF,(R4)+
    MOV  R2,(R4)+
    RTS  PC
;
BACKUP: SUB  #6,R4          ;GO BACK 6 BYTES
    SUB  #6,TBLEND         ;BACK UP END POINTER 6 BYTES
    RTS  PC                ;BACK TO MODE8
;
REDR:   .WORD  0            ;FLAG FOR REDRAW STATE
TRANS:  MOV  R4,-(SP)
    JSR  PC,INPVEC
    SUB  (R4),XF
    SUB  +2(R4),YF          ;GIVES TRANSLATION VECTOR IN XF,YF
1$:  ADD  XF,(R4)+          ;DO TRANSLATION ON EACH X,Y ENTRY
ADD     YF,(R4)+
CMP     #4,(R4)+         ;QUIT MODE?
BNE     1$
MOV     (SP)+,R4
RTS     PC
REDRAW: MOV     #1,REDR  ;SET FLAG TO LOOP ON STEP UNTIL QUIT
STEP:   MOV     (R4)+,XF  ;GET XF FROM TBLE
        MOV     (R4)+,YF
        MOV     (R4),MODE
        CMP     (R4)+,#4 ;QUIT MODE?
        BNE     10$
        SUB     #6,R4     ;BACK UP ONE COMMAND
        JSR     PC,BELL
        MOV     #0,REDR
        RTS     PC
10$:    JSR     PC,TEKPLO
        JSR     PC,TEKALP ;ASSURE WE'RE OUT OF DARK MODE
        CMP     #1,REDR
        BEQ     STEP      ;STAY IN REDRAW LOOP
        RTS     PC
CHAR:   .WORD
        .PAGE
        .SBTTL FDRAW, MDRAW - EXTERNAL GEN CALLS

; THIS MODULE CALLS TWO MODULES, INIT AND REDRAW, AFTER BEING
; CALLED FROM ANOTHER PROGRAM. THE CALLER MUST PASS TO THIS
; MODULE THE ADDRESS OF HIS BUFFER WHICH CONTAINS XF, YF,
; AND MODE FOR EACH SUCCESSIVE POINT TO BE DRAWN. THE LAST
; POINT MUST BE FOLLOWED WITH XF, YF, AND "4" TO FLAG THE
; END OF THE DRAW LIST.

; IF CALLED FROM A FORTRAN ROUTINE, THE CALL IS:
; CALL FDRAW(TABLE).

; IF CALLED FROM A MACRO PROGRAM:
; MOV     #TABLE,R4
; JSR     PC,MDRAW.

FDRAW:  MOV     +2(R5),R4  ;GET VALUE OF ARG-1 FM CALL LIST
MDRAW:  JSR     PC,INIT
        JSR     PC,REDRAW
        RTS     PC

Page dimensions: 614.4x798.7
ON EXIT, R2 CONTAINS THE ADDRESS OF THE RAD-50 FILE NAME BUFFER.

ARGL1: .WORD 3
BUFADR: .WORD ASCBUF ;ASCII BUFFER
WORD CHCNT
FILNAM : .WORD ;POINT TO FILNAM LOCATION
ASCBUF: .BLK 7
CHCNT: .WORD 16
FILNAM: .BLK 4

G sophistication: .ASCIZ /*ENTER DEVICE, FILE NAME, EXT (DDD:FFFFF.EEE)...*/
.EVEN
.LIST
BEX

GETFN:

MOV R3, -(SP)
.PRINT #GFNMSG
MOV #ARGL1, R5 ;SET UP FOR FORTRAN-LIKE SUBR CALL
BIC #010000, 44 ;
MOV #16, CHCNT ;RESTORE MAX CHAR COUNT
JSR PC, LINEIN ;GET ASCII NAME FM CONSOLE
MOV #ARGL1, R5 ;SET UP FOR FORTRAN-LIKE SUBR CALL
JSR PC, PAKNAM ;PACK TO RADIX-50
MOV (SP)+, R3
RTS PC

.GRRETR

GRAPND: ;EP HERE IF R3 IS ALREADY SET TO THE OLD VALUE OF R4 (APPEND A FILE)
JSR PC, GETFN ;GET FILE NAME (ABOVE)
CMP FILNAM, #177777 ;IF (FILNAM -1)
BNE 10$ ;
.PRINT #EM1 ; THEN PRINT EM1
RTS PC ; TAKE ERROR RETURN

10$: MOV #ARGL2, R5 ; ELSE CALL GETFIL
MOV R3, TBLPTR
JSR PC, GETFIL
CMP R5, #0 ; IF (R5 = 0)
BNE 20$
.PRINT #EM2 ; THEN PRINT EM2
RTS PC ; TAKE ERROR RETURN

20$: MOV #TBL, R4 ; ELSE FIX POINTER FOR REDRAW
RTS PC

ARGL2: .WORD 3
WORD FILNAM
TBLPTR: .WORD 0
WORD WDCNT
WDCNT: .WORD 1000

PAGE

GRRETR, GRAPND - RETRIEVE GRAPH FROM DISK
GLOBL LINEIN, PAKNAM, PAK6, GETFIL, PUTFIL

10$: MOV #TBL, R3 ; INITIALIZE TABLE POINTER

JSR PC, GETFN ;EP HERE IF R3 IS ALREADY SET TO THE OLD
CMP FILNAM, #177777 ;IF (FILNAM -1)
BNE 10$ ;
.PRINT #EM1 ; THEN PRINT EM1
RTS PC ; TAKE ERROR RETURN

20$: MOV #TBL, R4 ; ELSE FIX POINTER FOR REDRAW
RTS PC

PAGE

GRRETR, GRSAVE - GRAPH SAVE ON DISK

87
GRSAVE:

JSR PC,GETFN
CMP FILNAM,#177777 ;IF (FILNAM = -1)
BNE 10$

PRINT #EM1 ; THEN PRINT EM1
RTS PC ; TAKE ERROR RETURN

10$: MOV #TBL, TBLPTR ; ELSE CALL PUTFIL
MOV TBLEND-TBLPTR, R5 ; WDCNT <- (END PTR - HEAD)
ASR R5 ;
MOV R5, WDCNT
MOV #ARGL2, R5
JSR PC, PUTFIL
CMP R5, #0 ; IF (R5 = 0)
BNE 20$

PRINT #EM3 ; THEN PRINT EM3
RTS PC ; TAKE ERROR RETURN

20$: MOV #TBL, R4 ; ELSE FIX POINTER FOR REDRAW
RTS PC

NLIST BEX

EM1: .ASCIZ /ERROR IN PAKNAM/<15><12>
EM2: .ASCIZ /ERROR IN GETFIL/<15><12>
EM3: .ASCIZ /ERROR IN PUTFIL/<15><12>

LIST BEX

; PAGE

.SBTTL GREXIT - GRAPH EXIT MODULE

GREXIT:

MOV #0, R0 ; RETURN A ZERO IN RO
MOV #4, R1
RTS PC

;
;
TBLEND: .WORD 0
TBLE: .BLKW 2000
.END GREXEC
.TITLE MSGS - "GRAPH" ASCII MESSAGES
.SBTTL GRHELP - EXPLAIN DRAW COMMANDS
.MCALL .V2..PRINT,.TTYIN,.REGDEF
.GLOBL GRHELP,GROPTS,LOOKUP
.REGDEF

MSGPTR:
1$: .WORD HLPD
2$: .WORD HLPM
3$: .WORD HLPF
4$: .WORD HLPQ
5$: .WORD HLFDOT
6$: .WORD HLPDSH
7$: .WORD HLPA
8$: .WORD HLPB
9$: .WORD HLPR
10$: .WORD HLPS
11$: .WORD HLPT
CRLF: .BYTE <15>
     .BYTE <12>
     .BYTE <200>
.EVEN

GRHELP:
.PRTN #1S
BIS #10000, 44 ;MAKE SURE NO-ECHO INPUT MODE
.TTYIN RO,#'Y ;GET YEA OR NAY
CMPB RO,#'Y
BNE 10$ ; WAS NAY - EXIT HELP MODULE
.PRTN #20S ;SOLICIT WHICH CMND TO EXPAND
.TTYIN ;GET COMMAND FOR EXPANSION
JSR PC,LOOKUP
ASL R2 ; *2
ADD #MSGPTR,R2
.PRTN #CRLF ;CAR'G RETN
.PRTN (R2) ;INDIRECT THRU R2
10$:
RTS PC

.NLIST BEX
20$: .ASCII /TYPE THE COMMAND YOU WANT HELP WITH:/<15><12>
     .BYTE <200>
     .EVEN
1$: .ASCII <15><12>/DRAW MODULE COMMANDS:/<15><12>
     .ASCII / A - ALPHA CHARACTERS/<15><12>
     .ASCII / D - DRAW LINE  M - MOVE CURSER/<15><12>
     .ASCII / P - DRAW POINT  . - DRAW DOTTED LINE/<15><12>
     .ASCII / - - DRAW DASHES  R - REDRAW PREVIOUS PICTURE/<15><12>
     .ASCII / M - MOVE CURSER  B - BACK UP ONE VECTOR/<15><12>
     .ASCII / T - TRANSLATE GEOMETRICALLY/<15><12>
     .ASCII / Q - QUIT DRAWING/<15><12>
     .ASCII / WANT MORE HELP? (Y/N):/
     .BYTE <200>
     .EVEN

HLFD: .ASCII / [ D ] THE D COMMAND IS USED TO DRAW A SOLID/<15><12>
     .ASCII / LINE FROM THE PREVIOUS CROSS-HAIR POSI-/<15><12>
A SYSTEM DESIGN TOOL FOR AUTOMATICALLY GENERATING FLOWCHARTS

UNCLASSIFIED

NL
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D</strong></td>
<td>Draws a dark vector from the origin to the cross-hair position.</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>Draws a point at the present cross-hair position.</td>
</tr>
<tr>
<td><strong>R</strong></td>
<td>Redraws the entire table from the present table pointer to the quit entry.</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td>Redraws only one vector at a time.</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td>Draws a point at the present cross-hair position.</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>Backs up one vector in core table and allows more draws at end of table.</td>
</tr>
<tr>
<td><strong>Q</strong></td>
<td>The Q command terminates the drawing mode.</td>
</tr>
</tbody>
</table>

**Hints:**
- Always back up two vectors, then skip one with the **S** command.
- When listing options for drawing or file handling, place a "4" in "TBLE" to signify end of table.
- For file handling, use the **S** command to skip to the initial point of the vector to be drawn.
.ASCII / ARE SHIFTED BY THE DIFFERENCE BETWEEN/<15><12>
.ASCII / THE OLD AND NEW VECTOR. THE NEW/<15><12>
.ASCII / CROSS-HAIR POSITION MUST BE SENT TO/<15><12>
.ASCII / THE COMPUTER BY ANY KEYSTROKE AFTER THE/<15><12>
.ASCII / CROSS-HAIR IS AT THE DESIRED POSITION./<15><12>
.BYTE <200>
.EVEN
.LIST BEX

.PAGE
.SBTTL GROPTS - GRAPH OPTIONS MODULE

; THIS MODULE LISTS ALL OPTIONS AVAILABLE TO THE USER
; FOR THIS SYSTEM, AND PROMPTS THE TTY OPERATOR TO
; SELECT ONE OF THE OPTIONS. THE RESULT IS RETURNED
; IN RO.
;
; GROPTS:

;PRINT #1$ ;PRINT THE ASCII STRING
BIS #010000, 44 ;NO-ECHO INPUT
.TTIN ;NOW READ THE CHOICE
SUB #0,RO ;ONLY NEED LAST 3 BITS
RTS PC

.NLIST BEX

1$: .ASCII /SELECT .../<15><12><15><12>
.ASCII / 1 - DRAW A NEW PICTURE/<15><12>
.ASCII / 2 - RETRIEVE PICTURE FROM DISK AND INITIALIZE/<15><12>
.ASCII / 3 - RETRIEVE PICTURE FROM DISK AND APPEND/<15><12>
.ASCII / 4 - STORE THIS PICTURE ON DISK/<15><12>
.ASCII / 5 - HELP! EXPLAIN DRAW COMMANDS/<15><12>
.ASCII / 6 - ALL DONE - EXIT NICELY/<15><12>
.BYTE <200>
.EVEN
.LIST BEX

.PAGE
.SBTTL LOOKUP - CHARACTER TABLE LOOK-UP ROUTINE

;******************************************************************************
; ENTER WITH A CHARACTER IN RO.
; ROUTINE SEARCHES "CHARS", A TABLE ON ANTICIPATED
; CHARACTERS, AND INCREMENTS R2 BY ONE UNTIL THE
; MATCH IS FOUND.
;
;******************************************************************************

CHARS: .WORD "DM
.WORD "PQ
.WORD ":-
.WORD "AB
.WORD "RS
.WORD "T"

LOOKUP:

MOV   #000377,R2 ;MINUS ONE IN BYTE NOTATION

1$:

INCB  R2

CMPD  CHARS(R2),R0 ;RO-BYTE IN (CHAR+R2)?

BNE   1$

RTS   PC

.END
GCS LIBRARY

VERSION OF 13 JULY 79

CURRENT CALLABLE ROUTINES ARE:

PAK6 - PACKS 6 CHARS TO RAD50
LINEIN - GETS LINE FROM TELETYPewriter
GETFIL - COPIES FILE FROM DISK TO MEMORY
PUTFIL - COPIES FILE FROM MEMORY TO DISK
PAKNAM - PACKS DEV:FILENAME.EXT TO RAD50

SUBROUTINE CALL FORMAT IS FORTRAN COMPATIBLE
ALL CALLED BY "JSR PC,XXX"
R5 MUST CONTAIN POINTER TO ARGUMENT LIST
ARGUMENT LIST FORMAT IS:

******************************************************************************
* UNDEFINED * # OF ARGUMENTS *
******************************************************************************
* ADDRESS OF ARGUMENT # 1 *
******************************************************************************
* ...
* ...
* ...
******************************************************************************
* ADDRESS OF ARGUMENT # N *
******************************************************************************

.MCALL .V2...,REGDEF,ENTER,LOOKUP,READW
.MCALL WRITW,SAV STATUS,REOPEN,CLOSE,PRINT
.MCALL TTYIN
.GLOBL PAK6,LINEIN,GETFIL,PUTFIL,PAKNAM
.REGDEF

COMMON STORAGE FOR ROUTINES

ST00: 0
ST01: 0
ST02: 0
ST03: 0
ST04: 0
ST05: 0
ST06: 0
ST07: 0
ST08: 0
ST09: 0
ST10: 0
ST11: 0
SBTTL PAK6 - RADIX50 PACKING ROUTINE

;ROUTINE PAK6 /HARTRUN/ 2 JULY 79
;PACKS 6 CHARACTERS INTO RADIX 50
;FIRST ARGUMENT IS POINTER TO 5 WORD BLOCK:
;FIRST 3 WORDS CONTAIN ASCII CHARS
;LAST 2 WORDS WILL RETURN PACKED RAD50
;IF ANY CHARACTERS ARE ILLEGAL,
;LAST 2 WORDS WILL RETURN 177777

PAK6:  MOV    R0,-(SP)  ;SAVE REGISTERS
        MOV    R1,-(SP)
        MOV    R2,-(SP)
        MOV    R3,-(SP)
        MOV    R4,-(SP)
        MOV    R5,-(SP)
        ADD    #2,R5       ;R5-> ADDRESS OF WORD BLOCK
        MOV    (R5),R0     ;R0-> WORD BLOCK
        MOV    R0,ST00     ;SAVE POINTER
        MOV    ST00,ST01   ;ST01 POINTS
        ADD    #6,ST01     ;TO END OF CHARS
1$:   MOVB   (R0),R1   ;GET NEXT BYTE
        BIC    #177600,R1  ;7-BIT ASCII
        CMPB   #40,R1      ;IS IT SPACE?
        BNE    2$          ;IF YES,
        MOV    R1,(R0)+    ;STORE IT
        BR     6$          ;
2$:   BIT    #100,R1   ;IS IT A-Z?
        BEQ    3$          ;IF YES,
        BIC    #177700,R1  ;GET SIX BITS
        CMP    #40,R1      ;IS IT LEGAL?
        BNE    5$          ;IF YES,
        MOV    R1,(R0)+    ;STORE IT
        BR     6$          ;
3$:   CMP    #44,R1    ;IS IT $?
        BNE    4$          ;IF YES,
        MOV    #33,(R0)+   ;STORE 33
        BR     6$          ;
4$:   CMP    #56,R1    ;IS IT .?
        BNE    5$          ;IF YES,
        MOV    #34,(R0)+   ;STORE 34
        BR     6$          ;
5$:   CMP    #60,R1    ;IS IT LEGAL?
        BGT    7$          ;GET DIGIT
        CMP    #71,R1      ;CONVERT TO RAD50
        BLT    7$          ;
        SUB    #60,R1      ;GET DIGIT
        ADD    #36,R1      ;CONVERT TO RAD50
        }
MOVE R1,(RO) + ; STORE IT

6$: CMP R0, STO1 ; ARE WE DONE?
BLT 1$ ; DO IT AGAIN
BR PACK ; ELSE PACK IT

IF ILLEGAL CHAR,

7$: MOV STO1, R1 ; POINT TO PACKED
MOV #177777, (R1) + ; SET PACKED WORDS
MOV #177777, (R1) ; TO 177777
BR REST ; AND RETURN

NOW FIRST 3 WORDS CONTAIN RAD50 CODES
NEXT PACK REF. ECKHOUSE P. 149

PACK: MOV STO0, STO2 ; STO2 POINTS TO
ADD #2, STO2 ; THIRD CHAR
MOV STO0, STO3 ; STO3 POINTS TO
ADD #5, STO3 ; SIXTH CHAR
MOV STO0, R0 ; R0-> FIRST CHAR
MOV STO2, R3 ; R3-> THIRD CHAR
MOV STO1, R4 ; R4-> PACKED WORDS

1$: CLR R1 ; SUM = 0
2$: CLR R2 ; R2 = 0
MOV R2, (R0) + ; GET CHAR
ADD R2, R1 ; SUM = SUM + CHAR
CMP R0, R3 ; DONE 3 CHAR YET?
BGT 3$ ; IF NOT,
ASL R1 ; MULTIPLY
ASL R1 ; BY 8
ASL R1 ; DECIMAL
MOV R1, -(SP) ; SAVE PARTIAL RESULT
ASL R1 ; MULTIPLY BY
ASL R1 ; 32 DECIMAL TOTAL
ADD (SP) +, R1 ; 32 + 8 DEC = 50 OCTAL
BR 2$ ; PROCESS NEXT CHAR

3$: MOV R1, (R4) + ; STORE PACKED WORD
MOV STO3, R3 ; R3-> SIXTH CHAR
CMP R0, STO1 ; DONE?
BLT 1$ ; DO NEXT THREE

REST: MOV (SP) +, R5 ; RESTORE REGISTERS
MOV (SP) +, R4
MOV (SP) +, R3
MOV (SP) +, R2
MOV (SP) +, R1
MOV (SP) +, R0
RTS PC

; RETURN TO MAIN PROGRAM

PAGE

SUBTL LINEIN - READ LINE FROM TELETYPEN
; ROUTINE LINEIN/HARTRUM/ 21 JUNE 79
; GETS A LINE FROM THE TELETYPEN
; LESS THAN 80 CHARACTERS
; FIRST ARGUMENT IS BUFFER ADDRESS
; SECOND ARGUMENT IS CHARACTER COUNT
; ON CALL, CONTAINS DESIRED NUMBER

95
; ON RETURN, CONTAINS ACTUAL NUMBER
; NOTE - <CR> AND <LF> ARE NOT STORED

LINEIN: MOV R0,-(SP) ; SAVE REGISTERS
MOV R1,-(SP)
MOV R2,-(SP)
ADD #2,R5 ; GET 1ST ARG
MOV (R5)+,R1 ; BUFFER ADDR
MOV R5)+,STOO ; BYTE CNT DESIRED
CLR R2 ; COUNT BYTES DONE

1$: .TTYIN ; GET CHAR
CMPB #15,R0 ; WAS IT <CR> ?
BEQ 1$ ; GET THE <LF>
CMPB #12,R0 ; WAS IT <LF> ?
BEQ 2$ ; ALL DONE
CMP STOO,R2 ; BUFFER FULL ?
BEQ 1$ ; IGNORE THE CHAR
MOVB R0,(R1)+ ; STORE IT
INC R2 ; COUNT THEM BYTES!
BR 1$ ; DO IT AGAIN

2$: MOV R2, ; RETURN ACTUAL COUNT
     R5
MOV (SP)+,R2 ; RESTORE REGISTERS
MOV (SP)+,R1
MOV (SP)+,R0
RTS ; GO HOME

PAGE

.SBTTL GETFIL AND PUTFIL ROUTINES
; ROUTINES GETFIL AND PUTFIL/HARTRUM/22 JUN 79
; GETFIL COPIES A FILE FROM DISK TO MEMORY
; PUTFIL COPIES A FILE FROM MEMORY TO DISK
;
; FIRST ARGUMENT IS DBLK ADDRESS, 0 TO DEFAULT
; DBLK: DEVICE CODE IN RAD50
; FILENAME, FIRST 3, IN RAD50
; FILENAME, LAST 3, IN RAD50
; EXTENSION, IN RAD50
; DEFAULT IS FD0:DRAW.PIX
; SECOND ARGUMENT IS 1ST WORD OF FILE BUFFER
; *
; *NOTE - GETFIL WILL RETURN AN INTEGER NUMBER OF
; * 256-WORD BLOCKS. THEREFORE, THE FILE BUFFER
; * MUST CONTAIN AN APPROPRIATE NUMBER OF
; * 256-WORD (512-BYTE) BLOCKS TO HOLD THE FILE.
; *
; THIRD ARGUMENT IS # OF WORDS TO TRANSFER
; MUST BE SUPPLIED FOR PUTFIL ONLY
; GETFIL RETURNS ACTUAL # OF WORDS
; R5 WILL RETURN 0 IF ERROR OCCURS
;
GETFIL: MOV R1,-(SP) ; SAVE REGISTERS
MOV R2,-(SP)
MOV R3,-(SP)
MOV  R4,-(SP)
ADD  #2,R5       ;GET DBLK ADDRESS
MOV  (R5)+,R2
BNE  1$         ;SKIP IF USER DEFINED
MOV  #FILNAM,R2 ;DEFAULT FILENAME
1$:  .LOOKUP  #STOO,#0,R2  ;OPEN FILE @R2 ON CHANNEL 0
BCS  ERROR      ;STOO IS 3 WORD COMMO BLOCK
.SAVESTATUS #STO10,#0,#STATUS ;GET DIRECTORY
BCS  ERROR
CLR  R1
MOVB  STATUS+5,R1  ;IS BLOCK COUNT
BNE  ERROR       ;>ONE BYTE?
MOVB  STATUS+4,R1  ;GET BLOCK COUNT
SWAB  R1         ;WORDCOUNT=256XR1
.REOPEN  #STO10,#0,#STATUS ;REOPEN FILE
BCS  ERROR
MOV  (R5)+,R3  ;GET BUFFER ADDRESS
MOV  R1,(R5)+  ;SAVE WORD COUNT
.READW  #STOO,#0,R3,R1,#0 ;READ FILE
BCS  ERROR
.CLOSE  #0      ;CLOSE FILE
BCS  ERROR
BR  DONE       ;GET OUT

PUTFIL:  MOV  R1,-(SP)    ;SAVE REGISTERS
MOV  R2,-(SP)
MOV  R3,-(SP)
MOV  R4,-(SP)
ADD  #2,R5        ;GET DBLK ADDRESS
MOV  (R5)+,R2
MOV  (R5)+,R3  ;GET BUFFER ADDRESS
MOV  R5)+,R1       ;GET WORD COUNT
TST  R2           ;USER DEFINED FILENAME?
BNE  1$          ;IF YES,SKIP
MOV  #FILNAM,R2 ;DEFAULT FILENAME
MOV  R1,R4       ;TO GET BLOCK #
CLR#B  R4         ;DIVIDE BY 256
SWAB  R4          ;THEN ADD 1
INC  R4           ;TO GET IT ALL
.ENTER  #STOO,#0,R2,R4  ;OPEN FILE ON CHANNEL 0
BCS  ERROR
.WRITE  #STOO,#0,R3,R1,#0 ;WRITE FILE
BCS  ERROR
.CLOSE  #0        ;CLOSE FILE
BCS  ERROR
DONE:  MOV  (SP)+,R4  ;RESTORE REGISTERS
MOV  (SP)+,R3
MOV  (SP)+,R2
MOV  (SP)+,R1
RTS  PC          ;GO HOME
ERROR:  CLR  R5   ;SET ERROR RETURN
*PRINT  #EMSG
BR  DONE      ;AND QUIT

FILNAM:  .RAD50 /PDO/    ;DEFAULT DBLK
        .RAD50 /DRA/
.RAD50 /W/
.RAD50 /PIX/

STATUS:
.WORD 0 ; CHANNEL STATUS WORD
.WORD 0 ; STARTING BLOCK #
.WORD 0 ; FILE LENGTH IN 256-WORD BLOCKS
.WORD 0 ; UNUSED
.WORD 0 ; UNIT # OF DEVICE // I/O COUNT

EMSG:
.ASCIIZ /ERROR........./

.EVEN

; PAGE

.SBTLT PAKHAM - PACK DEV:FILENAME.EXT TO RAD50
;ROUTINE PAKHAM/HARTRUM/13 JULY 79
;PACKS PDP-11 FILENAMES INTO
; FOUR RADIX-50 WORDS.
;USES ROUTINE PAK6.
;FIRST ARGUMENT IS ASCII BUFFER.
;SECOND ARGUMENT IS ASCII COUNT.
;THIRD ARGUMENT IS 4-WORD BUFFER.
; 177777 RETURNED IF ANY ERRORS.

; PAKHAM:
MOV R0,-(SP) ; SAVE REGISTERS
MOV R1,-(SP)
MOV R2,-(SP)
MOV R3,-(SP)
MOV R4,-(SP)
ADD #2,R5 ; R5 -> ADDR OF ASCII BUFFER
MOV (R5)+,BUFLOC ; BUFLOC -> ASCII BUFFER
MOV R5+,PAKCNT ; # OF CHAR
MOV (R5),R2 ; R2 -> 4-WORD ANSWER
MOV (R5),ANSWPT ; AND SAVE IT.
MOV #NAMPAK,R1 ; R1 -> 5-WORD AREA

; SCAN:
CLR COLON ; SEARCH ASCII STRING
CLR PERIOD ; FOR COLON AND PERIOD
CLR ALL
MOV BUFLOC,R0 ; START OF STRING
CLR R3 ; CHAR COUNT

1$: 
INC R3
CMPB #72,(R0) ; IS IT "":
BNE 2$; YES, SET FLAG

2$: 
INC COLON ; YES, SET FLAG
CMPB #56,(R0)+ ; IS IT "."?
BNE 3$; YES, SET FLAG

3$: 
INC PERIOD ; YES, SET FLAG
CMP R3,PAKCNT ; ALL DONE?
BLT 1$; END OF STRING SEARCH
MOV BUFLOC,R0 ; SET PASS 1
CLR PASS
CLR R3 ; ASCII BUFFER COUNT
CLR R4 ; FIELD CHAR COUNT

PAKIT: 
TST COLON ; DEVICE CODE?
BNE 1$; YES, IT EXISTS
MOVB #106,(R1)+ ; NO COLON,
MOV #104, (R1)+ ; USE DEFAULT
MOV #60, (R1)+ ; OF FDO:
BR 3$

; THIS SECTION PACKS DEV CODE

1$:
INC R3
INC R4
CMP #72, (R0) ; IS IT ":" ?
BEQ 2$
CMP #4, R4
BEQ PAKER1 ; DEV CODE > 3 CHAR
MOV (R0)+, (R1)+ ; STORE IT
BR 1$ ; GET NEXT

2$:
TSTR (R0)+ ; SKIP ":"

22$:
CMP #4, R4 ; WERE THERE 3 CHAR?
BEQ 3$
MOV #40, (R1)+ ; TRAILING BLANKS
INC R4
BR 22$ ; TO FILL IT UP

; THIS SECTION STORES
; 3 CHARACTERS OF FILENAME

3$:
CLR R4
CMP R3, PAKCNT ; DEVICE NAME ONLY?
BGE 5$

4$:
TST ALL ; ARE WE DONE?
BNE 6$ ; (USED ON PASS 2)

44$:
CMP #56, (R0) ; IS IT "." ?
BEQ 6$ ; END OF FILENAME
INC R3
INC R4
CMP #4, R4
BEQ PAKER2 ; FILENAME > 6 CHAR
MOV (R0)+, (R1)+ ; STORE IT
CMP R1, #NAMPAK+6 ; END OF PAK AREA?
BEQ 7$
CMP R3, PAKCNT
BGE 5$
BR 44$

5$:
INC ALL ; FLAG FOR BUFFER END

6$:
INC R4
CMP R4, #4 ; WERE THERE THREE CHAR?
BEQ 7$
MOV #40, (R1)+ ; TRAILING BLANKS
BR 6$

; THIS SECTION PACKS 6 CHARACTERS
; BY CALLING PAK6

7$:
CMP PASS, #1
BEQ 8$ ; SKIP ON PASS 2
MOV #AREA, R5 ; SET UP
MOV #NAMPAK, AREA+2 ; PARAMETERS
JSR PC,PAK6 ;PACK 6 CHAR
CMP NAMPAK+6,#177777;DID IT WORK?
BEQ PAKER3 ;WHOOPS!
MOV NAMPAK+6,(R2)+ ;LOAD FOR
MOV NAMPAK+10,(R2)+ ;RETURN
TST PASS ;WHICH PASS?
BNE DONE2 ;IF 2, DONE
INC PASS ;PREPARE PASS 2
MOV #NAMPAK,R1
CLR R4
BR 4$

; THIS SECTION STORES EXTENSION

8$: CLR R4 ;NOW PAK EXTENSION
TST PERIOD ;WAS THERE ONE?
BEQ 9$ ;FORGET IT!
TSTB (R0)+ ;SKIP "."
INC R3

9$: TST ALL
BNE 99$ ;NO MORE
INC R3
INC R4
CMP R3,PAKCNT ;END OF ASCII?
BGT 10$ ;MORE THAN 3 CHAR?
CMP R4,#4 ;MORE THAN 3 CHAR?
BGE 7$ ;YES, TRUNCATE
MOVB (R0)+,(R1)+ ;STORE CHAR
BR 9$ ;PAK IT!

99$: INC R4
10$: CMP R4,#3 ;3 CHAR?
BLE 11$ ;PAK IT!
INC PASS
BR 7$ ;PAK IT!

11$: MOVB #40,(R1)+ ;TRAILING BLANKS
INC R4
BR 10$ ;

; ROUTINES

PAKER1: .PRINT #EMSG2 ;ERROR ROUTINES
BR ALLERR
PAKER2: .PRINT #EMSG3 ;ERROR ROUTINES
BR ALLERR
PAKER3: .PRINT #EMSG4 ;ERROR ROUTINES
BR ALLERR
ALLERR: MOV AN5WPT,R2 ;R2->ANSWER AREA
MOV #177777,(R2)+ ;SET ALL
MOV #177777,(R2)+ ;TO 177777
MOV #177777,(R2)+
MOV #177777,(R2)
BR DONE2

DONE2: MOV (SP)+,R4 ;RESTORE REGISTERS
MOV (SP)+,R3
MOV (SP)+, R2
MOV (SP)+, R1
MOV (SP)+, RO
RTS PC
;
; STORAGE
;
NAMPAK: .BLKW 6
PAKCNT: 0
ANSWPT: 0
BUFLOC: 0
PASS: 0
COLON: 0
PERIOD: 0
ALL: 0
AREA: .BLKW 2
;
; MESSAGES
;
EMSG2: .ASCIZ /DEVICE CODE > 3 CHARS/
EMSG3: .ASCIZ /FILENAME > 6 CHARS/
EMSG4: .ASCIZ /ERROR IN PAK6 ROUTINE/
.END
G.1 User hints

The following hints should make it easier for you to use the graph system the way you think it should work.

Runaway redraws: Always assure that you have included a quit command at the end of your figure. The quit command places a "4" at the end of the data base. Redraw looks for the number 4 as the tail indicator of the table.

How to call for a redraw: Once you have entered a picture that looks fairly good, you may want to redraw it to clear erroneous vectors (see below about correcting erroneous vectors). To redraw, quit, then select option 1 (draw), then type redraw. That's a lot of work, but the initialization and file handling is easier than allowing the redraw options without leaving and then reentering the draw module.

How to back up nicely: Suppose you draw a vector that you want to change. Simply back up with the B command once for each vector until you arrive at the correct place to make the change. Suggest here that you back up one more command than necessary, then use the step command ("S"). This will correctly reset the graphics terminal's origin pointer. Now type in the replacement vector. The erroneous vector will still appear, but the replacement vector will be drawn correctly. If redrawn, the erroneous vector will not appear.

Calling GRAPH from other programs: Graph can be called by FORTRAN or Macro programs in order to plot a graph of calculations or drawings.
made within those programs. The table must be prepared in the proper format (see figure 4-1. Call formats are:

```
FORTRAN                      MACRO
CALL FDRAW(TABLE)           MOV   #TABLE,R4
JSR   PC,MDRAW
```

G.2 Recommendations for Improvement

As I see it, the following are the most obvious areas for improvement for the interested programmer.

Table Insertions: The data base would be easier to manage - and complicated changes to the graphical figure would be simpler - if a figure could be inserted in a particular place in the data base. This way when translations occur, any changes included at the end of the drawing session can be excluded from translation if so desired. Generally speaking, a more logical arrangement of the data base would result.

Figure Stretching (Scaling): A capability should be built in that allows for expanding the values of all x or y coordinates relative to a center or focal point (i.e. add a scale or zoom capability per [12], chapter 4).

Figure Translation: The translation capability, although it works correctly, should be changed to conform to the 3X3 transformation matrix technique in [12], chapter 4.

Alpha mode storage economy: There is no reason to require 6 bytes of storage for a string of alpha-numerics. One byte is sufficient with a non-printable character like escape to terminate the string. Another
option would be to include a byte counter to indicate the length of the string. Only the first character in each string must have an associated xy-pair.

Suppression of menu: After a few tries with the system, the printing of the menu becomes a bother. Recommend changing the system so that the user is told that a menu will be available at any time by typing "?". Only at this time should the system display the lengthy menus or the additional "help" cues.
BIBLIOGRAPHY


7. Glass, Robert L. "From Pascal to Pebbleman...and Beyond". Datamation 25, 8 (July 1979), 146-150.


VITA

James Howard Keller was born on 16 July 1942 in White Plains, New York. He graduated from high school in White Plains, New York in 1960. He attended Purdue University and Hunter College until he enlisted in the US Air Force in August 1963. His enlisted tours included Bremerhaven, Germany, and Hurlburt Field, Florida. The latter included an academic assignment to the University of West Florida where he was awarded the Bachelor of Arts degree in Mathematics in April 1971. Following commissioning at Officer Training School, he was assigned administrative management positions at Williams AFB, Arizona and Osan AB, Korea until September, 1975. He then returned to Williams AFB as a computer systems programmer/analyst with the Air Force Human Resources Laboratory (Air Force Systems Command). In June, 1978 he entered the School of Engineering, Air Force Institute of Technology.

Permanent address: 2094 Auburn Avenue

Dayton, Ohio 45406
# A SYSTEM DESIGN TOOL FOR AUTOMATICALLY GENERATING FLOWCHARTS AND PREPROCESSING PASCAL

**Author(s):**
James H. Keller  
Captain

**Performing Organization Name and Address:**
Air Force Institute of Technology (AFIT-EN)  
Wright-Patterson AFB, Ohio 45433

**Controlling Office Name and Address:**
Air Force Avionics Laboratory (AFAL/AAF-2)  
Wright Patterson AFB, Ohio 45433

**Report Date:**
Dec, 1979

**Number of Pages:**
116

**Security Class:**
UNCLASSIFIED

**Distribution Statement:**
Approved for public release; distribution unlimited

**Supplementary Notes:**
Approved for public release; IAW AFR 190-17  
JOSEPH P. HIPPS, Maj, USAF  
Director of Public Affairs

**Keywords:**
Flowcharts  
Stepwise Refinement  
Automatic Programming  
Computer aided design  
Documentation

**Abstract:**
The portion of overall system costs attributable to software development and maintenance is presently near 50% and is continually increasing. Programmers and analysts are diligently searching for tools...
to assist them by automating the analysis, design, and documentation of software systems.

Flowcharting has lost some of its support as a powerful design tool due to the need for discipline, patience, and to some degree artistic talent. Automatic flowcharting, designed for specific languages and machines, provides automatic documentation only. No attempt has been made to link the automatic flowcharting to the compiler-ready code.

This study begins the development of an automatic program design tool to graphically display and update flowcharts and provide this link between the flowchart and the system it represents. A method of detailed, automatic design of programs, down to the elemental source language level, is proposed which displays graphical flowchart constructs and provides for iterative, stepwise refinements of the flowcharts. The final system, described by selecting flowchart constructs and completing the descriptions of the details of each construct, is maintained in a data structure that allows for subsequent refinement and for optionally producing a compiler-ready source listing.