INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
AUTOMATIC FLOWSHEET DRAWING FOR THE COMPUTER-AIDED
DESIGN OF CHEMICAL PROCESSES USING INTERACTIVE COMPUTER
GRAPHICS

The Ohio State University

Ph.D.

1980

University
Microfilms
International

300 N. Zeeb Road, Ann Arbor, MI 48106
18 Bedford Row, London WC1R 4EJ, England

Copyright 1980
by
Steacy, Peter Evans
All Rights Reserved
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs ______
2. Colored illustrations ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Print shows through as there is text on both sides of page ______
6. Indistinct, broken or small print on several pages ______ throughout ______
7. Tightly bound copy with print lost in spine ______
8. Computer printout pages with indistinct print ______
9. Page(s) ______ lacking when material received, and not available from school or author ______
10. Page(s) ______ seem to be missing in numbering only as text follows ______
11. Poor carbon copy ______
12. Not original copy, several pages with blurred type ______
13. Appendix pages are poor copy ______
14. Original copy with light type ______
15. Curling and wrinkled pages ______
16. Other _______________________________
AUTOMATIC FLOWSHEET DRAWING
FOR THE COMPUTER AIDED DESIGN OF CHEMICAL PROCESSES
USING INTERACTIVE COMPUTER GRAPHICS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of the Ohio State University

By

Peter Evans Steacy, B.S. in Ch.E., M.E.

****

The Ohio State University
1980

Reading Committee:                   Approved By
Robert S. Brodkey                        Thomas L. Sweeney
Robert D. Mohler                         Adviser
Thomas L. Sweeney

Department of Chemical Engineering
ACKNOWLEDGMENTS

The following individuals and organizations deserve special recognition for their help.

1. John T. Heibel for his suggestion that I think about the use of interactive computer graphics for computer aided design in chemical engineering.

2. Thomas L. Sweeney for taking on the difficult job of advisor for a half completed research project.

3. Thomas L. Sweeney, Robert S. Brodkey, Robert D. Mohler and Hooshang Hemami for their speedy response and constructive comments when asked to read this dissertation.

4. The Engineering Graphics Department at the Ohio State University for two years of French Fellowship support.

5. The IRCC/CIS Computing Lab at the Ohio State University for the use of their text processing facilities.

6. The faculties of Lehigh University, the University of Florida and the Ohio State University for holding my interest.

7. My parents for support of all kinds.

8. My wife, Diane, for encouragement and help with the preparation of the figures and listings.
VITA

August 16, 1948... Born - Boston, Massachusetts

1970................. B.S. in Ch.E., Lehigh University, Bethlehem, Pennsylvania

1970-1972............ Graduate Assistant, Department of Chemical Engineering, University of Florida, Gainesville, Florida

1972............... M.E., University of Florida, Gainesville, Florida

1972-1974............ Engineer, E. I. du Pont Company, Wilmington, North Carolina

1975-1976............ Teaching Assistant, Department of Chemical Engineering, The Ohio State University, Columbus, Ohio

1976-1978............ Consultant, Industrial Data Terminals, Columbus, Ohio

1977............... Instructor, Department of Chemical Engineering, The Ohio State University, Columbus, Ohio
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>VITA</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF ALGORITHMS</td>
<td>viii</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>I. PROPOSAL FOR A NEW COMPUTER AIDED DESIGN SYSTEM</td>
<td>3</td>
</tr>
<tr>
<td>II. DATA STRUCTURES</td>
<td>14</td>
</tr>
<tr>
<td>III. AUTOMATIC FLOWSHEET DRAWING</td>
<td>22</td>
</tr>
<tr>
<td>IV. THE PROTOTYPE SYSTEM</td>
<td>98</td>
</tr>
<tr>
<td>V. CONCLUSIONS AND RECOMMENDATIONS</td>
<td>119</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>126</td>
</tr>
<tr>
<td>Appendix</td>
<td></td>
</tr>
<tr>
<td>A. THE RATFOR PROGRAMMING LANGUAGE</td>
<td>129</td>
</tr>
<tr>
<td>B. THE GRAPHICS SYSTEM</td>
<td>138</td>
</tr>
<tr>
<td>C. COMPUTER PROGRAM LISTINGS</td>
<td>143</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Computer Aided Design System Structure.</td>
<td>9</td>
</tr>
<tr>
<td>2. Revised System Structure.</td>
<td>11</td>
</tr>
<tr>
<td>3. Avoiding Stream Crossings.</td>
<td>23</td>
</tr>
<tr>
<td>4. Minimizing Number of Bends.</td>
<td>25</td>
</tr>
<tr>
<td>5. Avoiding Zig-Zag.</td>
<td>26</td>
</tr>
<tr>
<td>6. Non-Planar Graphs.</td>
<td>28</td>
</tr>
<tr>
<td>7. Derivation of a Graph from a Flowsheet.</td>
<td>30</td>
</tr>
<tr>
<td>8. Alternate Representations for the Graph in Figure 7.</td>
<td>34</td>
</tr>
<tr>
<td>9. Graph Corresponding to Unrealizable Flowsheet.</td>
<td>36</td>
</tr>
<tr>
<td>10. Representations of the Process Topology.</td>
<td>38</td>
</tr>
<tr>
<td>11. Example Graph and Adjacency Lists.</td>
<td>40</td>
</tr>
<tr>
<td>12. Palm Tree Derived from Graph in Figure 11.</td>
<td>45</td>
</tr>
<tr>
<td>13. Data Structures for the Planarity Algorithm.</td>
<td>52</td>
</tr>
<tr>
<td>14. Graph With No Branches.</td>
<td>53</td>
</tr>
<tr>
<td>15. Conflict Between Paths 2 and 3 of the Graph in Figure 14.</td>
<td>55</td>
</tr>
<tr>
<td>16. Final Embedding of Graph from Figure 14.</td>
<td>56</td>
</tr>
<tr>
<td>17. Graph Requiring Recursive Application of the Algorithm.</td>
<td>58</td>
</tr>
<tr>
<td>18. Returning from Recursive Application of the Algorithm.</td>
<td>59</td>
</tr>
</tbody>
</table>
LIST OF FIGURES (continued)

19. Contents of Stacks at Various Points in the Progress of the Planarity Algorithm. 64
20. Dependency Graph for Graph from Figure 12 with Paths from Table 3. 75
21. Vertex Labeled Version of the Dependency Graph in Figure 20. 77
22. Planar Representation for Graph in Figures 11 and 12. 78
23. Relationship Between Paths and Branches. 83
24. Graph in Figure 22 with an Added Edge (3, 1). 87
25. Branch Placement Problem. 88
26. Handedness of Units. 90
27. Combinations of Entering and Exiting Spanning Tree Edges for a Symbol. 92
28. Choosing Hand of Unit to Avoid Zig-Zag. 95
29. Initial Screen Contents. 99
30. After Selecting ADD UNIT. 103
31. After Selecting ADD STREAM. 104
32. After Selecting DRAW FLOWSHEET. 106
33. After Selecting DRAW GRAPH. 107
34. Adding a Unit to an Existing Flowsheet. 108
35. After Adding Streams to Flowsheet in Figure 34. 109
36. Graph for Flowsheet in Figure 35. 111
37. After Selecting DELETE UNIT. 112
38. After Deleting Pump from Figure 37. 113
39. Example Abstract Graphs. 115
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Drawing Information for Graph in Figure 7.</td>
<td>32</td>
</tr>
<tr>
<td>2. Drawing Information for Graphs in Figure 8.</td>
<td>35</td>
</tr>
<tr>
<td>3. Sorted Adjacency Lists for Graph in Figure 12 and Paths as Generated by a Depth-First Search.</td>
<td>47</td>
</tr>
<tr>
<td>4. Resorted Adjacency Lists for Graph in Figures 11, 12, 22 and Paths as Generated by a Depth-First Search.</td>
<td>81</td>
</tr>
<tr>
<td>5. GRASP System Structure.</td>
<td>116</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Depth-First Search.</td>
<td>42</td>
</tr>
<tr>
<td>2. Conversion of a Graph to a Palm Tree.</td>
<td>49</td>
</tr>
<tr>
<td>3. Planarity Algorithm of Hopcroft and Tarjan.</td>
<td>61</td>
</tr>
<tr>
<td>4. Creating the Dependency Graph.</td>
<td>74</td>
</tr>
<tr>
<td>5. Converting the Dependency Graph from Edge to Vertex Labeling.</td>
<td>76</td>
</tr>
</tbody>
</table>
INTRODUCTION

Computer aided design systems for chemical engineering have been studied for more than 20 years. These are rather large computer programs designed to aid the chemical engineer by performing heat and material balances for a specified flowsheet. Of concern here are general systems designed for the steady-state simulation of user-specified processes (as opposed to systems written for a particular process or for dynamic simulations).

An introduction to the subject can be found in Perry and Chilton (1973). Westerberg, Huchison, Motard and Winter (1979) is a recent book which presents the state of the art with particular attention given to alternative approaches to the design of such systems. Review articles have been published regularly and provide an excellent summary of the technology as well as a nearly complete guide to the literature: Hlavacek (1977), Motard, Shacham and Rosen (1975), Flower and Whitehead (1973a, b), Kehat and Shacham (1973a, b, c) and Sargent (1967).

In this dissertation, a proposal is made for a new system for the computer aided design of chemical processes. Distinguishing this system would be its orientation toward fast effective interaction between engineer and computer.
Introduction

through the use of computer graphics. Thus, for example, emphasis would be given to providing the user with the ability to try out modifications to a proposed process rather than to providing optimization facilities. It will be shown that this orientation has two implications:

1. The normal structure of existing systems is inappropriate.

2. The computer must have the ability to automatically and artistically layout and draw a flowsheet given the process units and their interconnections.

Suggestions will be made for an improved system structure but most of the dissertation will be devoted to formalizing and solving the automatic flowsheet drawing problem.

A prototype system was built to demonstrate the interactive computer graphics features of the proposed system, including the automatic flowsheet drawing. The use of this system will be described and complete listings presented.
CHAPTER I

PROPOSAL FOR A NEW COMPUTER AIDED DESIGN SYSTEM

The aim of this proposal is to specify a system that takes advantage of interactive graphics technology to dramatically improve communication between the design engineer and the computer. By doing so it will be possible to better combine the engineer's experience and intuition with the computer's calculating power. The emphasis will be on the early stages of the design process - the exploration of alternative processing schemes, rather than optimization of an existing flowsheet. Phrased another way, emphasis is on helping the production of an optimal flowsheet as opposed to an optimized flowsheet. Thus, the goal is to produce a tool for the engineer to use in exercising his creativity. Present systems tend to stifle that creativity by imposing a high cost on the exploration of alternatives; data entry, error checking and analysis of results are all laborious.

The basic idea behind the proposed system is to use the graphics capability of the computer to display a flowsheet of the process being developed. Changes to the flowsheet will be made in an interactive fashion by an engineer using a light pen. Results of a computer calculation of the heat and material balances, and possibly an economic analysis,
The proposed system is best explained by outlining some possible approaches to its use. In this way it will also be possible to provide some comparisons with the capabilities of conventional systems.

1.0 CONVENTIONAL PROCESS SIMULATION

Use of most of today's computer aided design systems involves a complete specification of the process by an engineer, submission to a batch operated computer and subsequent analysis of a simulation of the process. Simulation is the calculation of output stream variables given input stream variables and equipment parameters. The design function (calculation of equipment parameters) occurs only by the engineer changing the process and resubmitting it to the computer for the job to be completely redone.

For conventional simulation the proposed system could increase the efficiency of the input and output jobs. A preprocessor could allow the engineer to specify the flowsheet interactively. Error checking would be greatly aided by having the flowsheet displayed. When the engineer was satisfied, the preprocessor could be instructed to prepare the input deck (or its equivalent) for the batch
Proposal

job. On the output end, a graphics oriented postprocessor could access data generated by the batch job, and display the flowsheet along with the results of the simulation. Note that the pre and post processors are automating jobs that the engineer would have to do anyway. No new capability is gained - just a reduction in effort.

2.0 STEP-WISE PROCESS SYNTHESIS

Advantage can be taken of the interactive structure of the system to allow an engineer to use the resources of the computer to help in the synthesis of the process flowsheet. The engineer can start by defining the raw material streams and the first processing unit. The computer will then respond with the calculated output streams, perhaps including investment and operating cost data. The engineer can then use this information in his development of the rest of the flowsheet.

Also at this time the engineer can bring his experience and judgment to bear on the often tricky trade-offs between desired outputs and reasonable equipment parameters. An important advantage to the interactive process is that problems with these trade-offs often suggest changes to the flowsheet.
Individual unit computations could be programmed for either simulation or design. Each piece of equipment has a number of degrees of freedom. Calculation of output streams given input streams and equipment parameters is known as simulation; calculation of equipment parameters given input and output streams is known as design. Note that design can be done by using simulation iteratively. Although it would be possible to calculate input streams given output specifications, prohibition of that option removes the potential for conflicts in the selection of design variables for units adjacent in the process flowsheet. The selection of design variables and the resultant effect on the design problem has been the subject of much study. See Rudd and Watson (1968) for an introduction.

Recycle is another problem that computer aided design systems have to overcome. While much research has been done on the computational difficulties that arise, it is even more important to optimize the use of recycle in the flowsheet. Using the proposed system the engineer will have the data available to develop the effective use of recycle. Questions occurring to the engineer like "I wonder if it is worth trying to recover some heat from this stream?" can be answered interactively without the high costs referred to earlier.
3.0 COMPUTER AIDED PROCESS SYNTHESIS

Automatic synthesis of processes has been an active research area in recent years. See Hendry, Seader and Rudd (1973) for a review article. These systems synthesize process flowsheets given specifications for the desired outputs. In an effort to make the problem computationally tenable, help is often given in the form of built in heuristics. These can have the effect, in some sense however, of stifling the creativity of the system. Thus combination of these techniques with the system being proposed here would appear to have interesting possibilities.

One approach would be to start with a computer synthesized process. The engineer could then apply his judgment and try various modifications. A second approach would be to proceed with step-wise synthesis as outlined above to specify key areas of the flowsheet. The computer could then be asked to automatically finish portions of the flowsheet such as separation trains and heat exchanger networks.
Most computer aided design systems for chemical engineering use a modular approach with a system structure similar to that in Figure 1. With this approach subroutines are provided to simulate common types of process equipment. The user specifies the process (units, their parameters and interconnections) and the executive simulates the process by calls to the unit computation subroutines. The executive must sequence the calculations, converge recycle loops and handle the interaction with the user. More extensive descriptions of this approach to flowsheet simulation can be found in Perry and Chilton (1973) and Motard, Shacham and Rosen (1975). Descriptions of specific systems are also available: Crowe, Hamielec, Hoffman, Johnson, Shannon and Woods (1971) for PACER; Motard and Lee (1971) for CHESS; Seader, Seider and Pauls (1974) for FLOWTRAN. The review articles by Kehat and Shacham (1973a, b, c) and Flower and Whitehead (1973a, b) include lists of existing programs.

The system being proposed here would modify the user interface and process data base portions of Figure 1. The user interface would be extended directly to the unit specific modules in addition to being implemented with computer graphics. The process data base would use data structures appropriate for the changing nature of the process that interaction implies. Thus recognition would be given to the fact that the size of the flowsheet will change
FIGURE 1: Computer Aided Design System Structure
during a session with the user; it will be necessary for units and streams to be deleted as well as added. A diagram of the revised structure is shown in Figure 2.

The use of interactive computer graphics for computer aided design in chemical engineering has not been dealt with extensively in the literature. Hutchison and Forder (1967), however, described a general network description program where a flowsheet would be one example of a network. Sargent (1967) mentioned this in his review article but stated

"... a personal view is that ... extensive direct interaction between designer and computer is an intermediate phase, providing a steppingstone from traditional design methods to almost completely automatic procedures, with human judgement entering only at a relatively high level in the over-all assessment."

The claim in this dissertation is that allowing fast easy manipulation of the process topology is allowing human judgment to enter the design process at a very high level.

A more recent application of graphics technology to computer aided process design is a system implemented by Singh (1976). His system differs from the one proposed here in that the flowsheet is not drawn automatically. Instead, the user defines process unit symbols and positions them on the screen of the graphics terminal himself. Process streams are drawn in one of three ways:
FIGURE 2: Revised System Structure
Proposal

1. by a simple algorithm that allows streams to cross unit symbols,
2. by a time consuming algorithm that will not cross symbols though streams can still cross each other,
3. as positioned by the user.

The system being proposed in this dissertation positions unit symbols and streams automatically with no stream crossings whenever that is mathematically possible.

Automatic flowsheet drawing is important because the definition and placement of symbols can be a complicated task if a good looking flowsheet is desired. Also, for the exploration of alternatives, the final form of the flowsheet is not known ahead of time. Thus early units may be placed in locations that will later prove awkward. Modification of the flowsheet to fit in an additional unit may be difficult, for example. The net result is that the engineer may spend too much of his time and creativity on engineering graphics instead of chemical engineering. In the worst case, an alternative may go unexplored because of purely graphical trouble fitting it into the flowsheet.

It might be expected that an automatic flowsheet drawing system could use techniques similar to those used in commercial printed circuit board layout systems. Information on those systems is not available in the open literature, however. Also, the orientation of those systems
is apparently different than the one being proposed here. For example, user interaction is usually required to complete the layout of a reasonably complex printed circuit board. The flowsheet drawing system in this dissertation is completely automatic.
CHAPTER II

DATA STRUCTURES

Computer aided design systems are faced with the problem of organizing and managing enormous amounts of data. Included are definitions of the process units, information about their interconnections, parameters such as sizes, user information such as unit names, thermodynamic data for use in computations, raw material stream data and so on. It is the nature of man's ability to deal with such quantities of data that motivates the proposal outlined in the last chapter. Previous computer aided design systems have handled the data pretty much on the computer's terms. The proposed system would shift the emphasis to handling data in ways better suited to a human's capabilities. To a large extent, it is this large amount of data that makes these systems both difficult to design and to use.

It is the opportunity to take advantage of the structure of the data that makes the modular approach to simulation shown in Figure 1 attractive. The system structure in Figure 2 is an attempt to further segregate parts of the process data base.
The process data base can be divided into two parts: that which has predictable structure that can be known to the executive ahead of time, and that which has structure dependent on factors which are best left unknown to the executive. Figure 2 shows this split as being made between data structures that are specific to an individual process unit and data structures that are not.

The executive must know the process topology, the identity of the process units and their interconnections, so that it can manage the calculations and display the flowsheet. It must also know the state of the process streams so that the results of simulations can be shown on the flowsheet. All of this information can be fit into a regular structure, so that the executive can use it with relative ease. The parameters associated with each unit do not yield so easily however. These parameters can be almost anything: heat transfer coefficients, number of trays, feed tray location; and there can be any number of them.

The suggestion, then, is to isolate the process unit specific data so that only routines written specifically for that unit type need know its format. Data common to all units would then be kept in a standard format accessible to the executive. Those functions of a computer aided design system that depend on knowledge of the specific unit type
Data Structures

would be handled by calls to the special routines. These include routines to initialize the unit data by interaction with the user, the drawing of the flowsheet symbol, performance of the calculations for the user and a report on the state of the unit to the user. Note that in the case of design oriented calculations, the output streams would be fixed with the real results of the calculations being hidden from the executive but available to the user through the unit report. There is no reason why the report could not be of standard format so that the executive could provide a uniform presentation of the results.

The proposed system is oriented toward being able to interactively explore modifications to a flowsheet. It follows then that its internal data structures should have a similar orientation; it should be easy to add, delete and modify elements of those data structures. In particular the data structures must be able to grow dynamically as the user develops his flowsheet. The data structures that best meet these requirements are list structures. Data structures are routinely studied by computer scientists; see Knuth (1968) or Berztiss (1975) for example.

In a list structure a list element, or node, is created to hold data about the item of concern. For example, a list could be created to hold information about the process units
in a flowsheet. A list element could be created for each unit with two of the pieces of data being unit name and unit type. The list elements would be linked together by the use of pointers. Thus one piece of data in each list element would be a pointer to the next element in the list. Adding and deleting elements (units in the flowsheet) is then a matter of modifying pointers.

The use of pointers is not limited to one per list element. If two pointers are included in each element, one to the next element and one to the previous element, flexibility is gained in the ability to delete an element. If pointers to sub-lists are included, more complicated data structures such as trees and graphs can be represented. For example, the list element for a process unit could include a pointer to a list of elements representing its input streams and a pointer to a similar list representing its output streams.

Associated with the notion of a list element as described above is the notion of a record or structure. For example the list element for a process unit could contain a unit name represented by a character string, a unit type represented by an integer code as well as pointers and perhaps data of other types. A mechanism is needed for manipulating such collections of variables with possibly
dissimilar data types. Such a collection of variables is known as a structure in some programming languages, such as PL/I, and a record in others, such as PASCAL. FORTRAN has no such facility; the programmer must hand craft mechanisms such as using the same index for several arrays with different data types.

The use of record variables and list structures is intended to simplify the management of computer storage. Since the size of the problem being worked on will be changing during a session with the user, it would be desirable to dynamically allocate memory for these data structures only when needed. Older computer aided design systems use straightforward data structures, typically FORTRAN arrays, that always provide enough space for the largest possible job. As a result, even very small jobs require large amounts of memory in the computer.

The use of record variables and list structures simplifies the design of a computer aided design system by organizing the computer's internal data in the same fashion that the flowsheet is organized. Thus when the user modifies the flowsheet, the computer modifies its data structures in a similar way.
Another advantage of using these data structures is that it eases the derivation of formal mathematical models of the process. One such application is used in the following chapters of this dissertation to solve the automatic flowsheet drawing problem.

A flowsheet can be modeled as a digraph (a graph with directed edges) where the vertices of the graph correspond to process units and the directed edges correspond to the process stream. The data structures outlined above are those normally used to represent a graph in a computer. Graph theory is an actively studied area of mathematics (see Harary (1969) for example) and efficient algorithms exist for answering many interesting questions about them. One problem of interest in computer aided design is that of isolating minimum sets of units that must have their calculations done simultaneously because of the existence of recycle. These sets correspond to what graph theory calls the strongly connected components of the digraph. Algorithms for solving this problem as well as answering other questions about graphs are found in books such as Aho, Hopcroft and Ullman (1974) and Reingold, Nievergelt and Deo (1977).
Alternative data structures for use in chemical engineering computer aided design have appeared in the literature recently. Evans, Joseph and Seider (1977) present what they term a plex data structure. It appears to be a well developed form of the general list structures described above. They use this to achieve modularity and flexibility for their proposed system for computer aided design. Their system is not oriented toward interactive computation, however. Instead, they propose the use of a problem oriented language or problem oriented calling programs to manipulate the data structures.

Singh (1976) uses a set-theoretic representation of a flowsheet in his interactive graphics based computer aided design system. The goal of flexibility in use appears to be largely achieved but the requirements that sets have predetermined maximum sizes and must use contiguous memory locations cause inefficiencies and runtime restrictions that would be nice to avoid.

The data structuring techniques discussed in this chapter are in general well known to modern computer science. Their use, in fact, is recommended and well supported by some of the modern programming languages such as PASCAL (Jensen and Wirth, 1974). It would appear that their use and resultant benefits would flow naturally from
the selection of a modern language for the implementation of a new computer aided design system. The use of FORTRAN means that effective data structures are difficult to implement and appear more as new inventions than as straightforward applications of modern programming technique. While it is certainly recognized that there are persuasive reasons for using a well standardized, universally available language such as FORTRAN, computer aided design seems to be one area where its use has slowed progress. An effective compromise might be to program the executive portion of Figure 2 in a modern language but to provide links for unit computations to be written by engineers using standard FORTRAN.
In order to program a computer to automatically draw a flowsheet, it is necessary to have a formal statement of what is desired. The basic requirement is that a symbol be drawn for each unit in the process and that the proper locations in these symbols be connected together by lines to represent the process streams. It is also required that the units have an identifying label and that the streams have room for the display of the value of a parameter (temperature or concentration of a component for example). The unit symbols should not overlay one-another and the streams should not cross through the symbols. Beyond these basics, however, the requirements get a little fuzzier; the flowsheet must be legible and easy to read. In a conventional drawing of a flowsheet this is achieved by the artistic skill of the person who does the drawing. While it does not seem likely that the computer can be turned into a true artist, it is possible to come up with some general rules, heuristics, that seem to lead to usable flowsheets.

The major heuristic to be used is to require that, in addition to not crossing through unit symbols, the streams not cross each other (Figure 3). This requirement will do
FIGURE 3: Avoiding Stream Crossings
Automatic Flowsheet Drawing

quite a bit to restrict the placement of the unit symbols — not just the streams. Other (much easier to satisfy) heuristics are to allow only right angle bends in streams, to minimize the number of such bends (Figure 4), and to avoid zig-zag in the flowsheet (Figure 5). Additional heuristics are certainly possible but those listed here seem to give a reasonable result. Note that although there are no conflicts so far, the addition of other heuristics would soon make the resolution of conflicts an important consideration.

Of the flowsheet drawing heuristics mentioned, only avoiding stream crossings seems difficult. The plan is to model the flowsheet by a graph (in the mathematical sense), derive information about a planar representation for that abstract graph, and then construct a drawing of the flowsheet based on that information.

A graph consists of a set of vertices and a set of edges. Each of the edges has associated with it a pair of vertices which the edge is said to be incident upon. If the pairs of vertices are ordered, the graph is said to be a directed graph or digraph. If the vertices are numbered, the graph is said to be a labeled graph. Graphs are normally drawn by representing the vertices as dots, circles, squares, et cetera and the edges as lines between
FIGURE 4: Minimizing Number of Bends
FIGURE 5: Avoiding Zig-Zag
them. If it is possible to draw a representation of the graph on a plane with no edge crossings, the graph is said to be planar. Note that it is certainly possible to draw a planar graph with edges that cross. Knowledge that a graph has the property of planarity does not imply knowledge about how to draw a planar representation of that graph. Figure 3 shows both a planar and a non-planar representation of what is mathematically the same (planar) graph. Figure 6 shows two non-planar graphs. A flowsheet will be drawn even if its corresponding abstract graph turns out to be non-planar. In this case there will be some localized stream crossings.

1.0 DERIVING A GRAPH FROM A FLOWSHEET

The first step in the quest for a planar representation of a flowsheet is the derivation of a corresponding abstract graph. The obvious choice is to model each process unit as a vertex and each process stream as a directed edge of a digraph. This model has the appealing quality of looking very much like the flowsheet. It is also a proper model for answering computational questions about the flowsheet such as finding strongly connected components as mentioned in the previous chapter. For the present purpose, however, the resultant graph would have a property that would be hard for graph theory to deal with. That is that the set of edges
FIGURE 6: Non-Planar Graphs
incident upon a vertex would be an ordered set. If a distillation column were being modeled, for example, the set of edges would be either (tops, bottoms, feed) or (tops, feed, bottoms) depending on whether the symbol faced right or left, but in any case the order would be important.

It is better, then, to model each intersection of a process unit and process stream as a vertex. As before, each process stream is modeled as an edge but this time, an undirected edge. For the purpose of finding a planar representation of the flowsheet, the direction of flow for the streams is unimportant. At this point, the graph would simply be a collection of unconnected edges representing the process streams. To complete the graph, edges are added to connect the vertices of each process unit, thereby forming a cycle and causing the model to reflect the fact that those vertices belong to the same process unit. See Figure 7.

2.0 PLANAR REPRESENTATIONS

Now that the flowsheet has been modeled by a graph, the next step is to learn how to draw a planar representation of that graph. It is hoped that this knowledge will lead to an ability to draw a planar representation of the flowsheet itself.
FIGURE 7: Derivation of a Graph from a Flowsheet

Vertices model unit-stream connections. Edges cut by dotted lines model streams. Edges inside dotted lines model units.
The form of the information on drawing a planar representation for a graph will be a sequence of paths along with the order in which to draw those paths and the direction in which to close any cycles. A path can be specified by a list of vertices where there exists an edge between vertices adjacent in the list. Table 1 gives an example of information sufficient to describe a planar representation of the graph in Figure 7. The clockwise versus counter-clockwise designations for the complete cycles could be replaced by bottom versus top, or right versus left, referring to the sides of the graph on which the closing edges return.

Information such as that in Table 1 will lead fairly directly to a planar representation for the flowsheet. Two issues should be brought up at this time, however.

First, each of the process units in the flowsheet will be represented by a cycle in the graph. That cycle in turn will correspond to a path that has its last two vertices from the unit. Allowing the computer to choose either a clockwise or counter-clockwise representation for the cycle corresponds to allowing the computer to specify a left or right handed symbol for the process unit. Such a capability will thus have to be built into the program that constructs the flowsheet from the graph.
TABLE 1: Drawing Information for Graph in Figure 7

<table>
<thead>
<tr>
<th>Path</th>
<th>Vertices</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 3, 4, 5, 6</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>5, 2</td>
<td>clockwise</td>
</tr>
<tr>
<td>3</td>
<td>4, 7, 8, 9, 10, 3</td>
<td>counter-clockwise</td>
</tr>
<tr>
<td>4</td>
<td>10, 11, 12</td>
<td>none</td>
</tr>
<tr>
<td>5</td>
<td>11, 9</td>
<td>clockwise</td>
</tr>
</tbody>
</table>
Automatic Flowsheet Drawing

Second, in the effort to find a planar representation for the graph, the computer may embed paths inside cycles. If the computer were to place paths inside a cycle that represented a process unit, the result would be unrealizable in the sense that a reasonable corresponding representation of the flowsheet could not be constructed. Figure 8 shows two planar representations for the graph in Figure 7 which have this problem. Table 2 shows the corresponding drawing information. Fortunately, it will be possible to have the computer avoid this problem nearly all the time. When it cannot, the flowsheet will be drawn anyway but will have at least one stream crossing. The process unit involved must have at least four stream connections before the problem is even possible (most units have only two or three). Figure 9, where vertices 1 through 4 represent a process unit, shows the structure necessary to have the problem occur. Graph Part 2 could not be moved outside the cycle without destroying the planarity of the representation.

3.0 PLANARITY ALGORITHM

Drawing information such as found in Tables 1 and 2 will be developed by extending a planarity testing algorithm published by Hopcroft and Tarjan (1974). Their algorithm efficiently tests an arbitrary graph to see if it is planar.
FIGURE 8: Alternate Representations for the Graph in Figure 7
TABLE 2: Drawing Information for Graph in Figure 8

<table>
<thead>
<tr>
<th>Path</th>
<th>Vertices</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 3, 10, 11, 12</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>11, 9, 10</td>
<td>clockwise</td>
</tr>
<tr>
<td>3</td>
<td>9, 8, 7, 4, 3</td>
<td>counter-clockwise</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 6</td>
<td>none</td>
</tr>
<tr>
<td>5</td>
<td>5, 2</td>
<td>clockwise</td>
</tr>
</tbody>
</table>
FIGURE 9: Graph Corresponding to Unrealizable Flowsheet

Vertices 1 to 4 represent a process unit. Graph parts 1 and 2 represent flowsheet segments of any size.
Automatic Flowsheet Drawing

or not. The efficiency is needed in this application because of the desire for fast interaction and because the job is intended to be done many times during the course of a session with the user.

Hopcroft and Tarjan's algorithm only tests for the property of planarity; it does not construct a planar representation. The conclusion of their paper suggests an approach for accomplishing such a construction, but perhaps overstates the ease with which it may be done. In any case the algorithm will be extended here to continue with the processing of the graph, even if it is found to be non-planar and to construct a new graph, called the dependency graph, to represent how the positions of cycles in the original graph depend upon each other. Additional algorithms will then be developed to use this information to produce drawing information and to use that information, in turn, to produce drawings of both the abstract graph and the flowsheet itself. The drawing information is never explicitly represented as in Tables 1 and 2 but rather is implicitly embedded in a digraph which is derived from the original abstract graph.

The representation of the process topology passes through a series of steps as shown in Figure 10. The derivation of an abstract graph from the process topology
FIGURE 10: Representations of the Process Topology
Automatic Flowsheet Drawing

has already been explained. Each of the other representations and associated transformations will be explained in the following sections.

3.1 Preliminaries

As Hopcroft and Tarjan (1974) so aptly state, "Graph theory is an endless source of easily stated yet very hard problems." Planarity is an easy concept to grasp but Hopcroft and Tarjan's algorithm to test for it is complicated and can be hard to understand. The description of the algorithm to follow is designed to informally show how it works. A more formal and complete development, including proofs, can be found in Reingold, Nievergelt and Deo (1977) as well as in Hopcroft and Tarjan (1974). A complete implementation is included in the program listings of Appendix C.

The data structure used to represent a graph in the computer is a set of adjacency lists called an adjacency structure. Each vertex in the graph has an adjacency list and each list element represents an adjacent vertex (there exists an edge between the two vertices). Figure 11 shows a graph and its set of adjacency lists.
FIGURE 11: Example Graph and Adjacency Lists
(From Reingold, Nievergelt, and Deo, 1977)
For the most part, the algorithms to follow take the form of a series of traversals, or searches, through various graphs. The method used to traverse the graphs is known as depth-first search, or backtracking. A depth-first search from any vertex follows an edge to a new vertex. If that vertex has not yet been considered, the process is repeated. If the vertex has been visited before, another edge from the original vertex is selected. When all edges from a vertex have been explored, the search backs up the edge that was followed to get to the current vertex and resumes the search from its predecessor. The entire graph is thus explored by probing as deeply into the graph as possible at each opportunity - just as the name depth-first search implies.

Some of the algorithms to be developed will be outlined using pseudo-code - programming language syntax to express flow of control but with English language statements to provide the semantics. Since the implementations of those algorithms in Appendix C are in RATFOR (a language designed to be translated into FORTRAN by a preprocessor), the pseudo-code will use RATFOR's syntax. A summary of RATFOR appears in Appendix A but the pseudo-code should be understandable without its study being necessary. For example, the depth-first search procedure is shown as Algorithm 1.
ALGORITHM 1: Depth-First Search

SUBROUTINE DEPTH_FIRST_SEARCH (V)
# Traverse a graph using depth-first search.
# Start from vertex V.
WHILE (There exists an unexplored edge, (V, W))
    IF (W not visited before)
        CALL DEPTH_FIRST_SEARCH (W)
RETURN
END
The depth-first search procedure is recursive, that is it is defined in terms of itself. Recursive procedures (subroutines) cannot be directly programmed in FORTRAN but can be in languages such as ALGOL, PL/I and PASCAL. The fact that many graph algorithms are best formulated as recursive procedures is another reason for choosing a language other than FORTRAN for implementation of a computer aided design system.

3.2 Conversion To A Palm Tree

Hopcroft and Tarjan's algorithm tests for planarity by generating paths through the graph as in Tables 1 and 2 and using mathematical tests to see if they can be embedded in a plane without interfering with each other. A path is a sequence of vertices connected by edges. In the presentation to follow a path means specifically a sequence of vertices visited by a depth-first search and terminated by a vertex that the search has already visited. Thus the depth-first search can be thought of as generating a sequence of paths that correspond to cycles in the graph. Efficiency is achieved in Hopcroft and Tarjan's algorithm by sorting the graph (in some sense) so that the number of possible embeddings that must be considered for the generated paths is restricted.
A depth-first search can be used to convert an undirected graph into a digraph that consists of a directed, rooted, spanning tree plus back edges. A directed, rooted, spanning tree is an acyclic graph that contains all the vertices of the original graph with one edge entering each vertex, and that has one vertex, the root, that has no entering edges. The spanning tree thus consists of all vertices and edges that when traversed by the depth-first search lead to a new vertex. The edges that lead to a vertex that has already been visited fall into the set of back edges. Adjacency list elements for edges directed against the direction imposed by the search are deleted. Hopcroft and Tarjan call the digraph resulting from the depth-first search a palm tree where the edges of the spanning tree are called tree arcs and the back edges are called fronds.

It is also useful to relabel the graph according to the order in which the vertices are visited by the depth-first search. Figure 12 shows the graph of Figure 11 after relabeling and conversion to a palm tree. Note that for all edges \((v, w)\) of the spanning tree, \(v\) is less than \(w\) and that for all back edges, \(w\) is less than \(v\).
Spanning Tree plus Back Edges (Palm Tree)

Labels

<table>
<thead>
<tr>
<th></th>
<th>new</th>
<th>old</th>
<th></th>
<th>new</th>
<th>old</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td>7</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td></td>
<td>8</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>e</td>
<td></td>
<td>9</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td></td>
<td>10</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>g</td>
<td></td>
<td>11</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 12: Palm Tree Derived from Graph in Figure 11
The course of the depth-first search is determined by the order in which edges of the graph are explored. This, in turn, is determined by the order in which the edges appear in the adjacency lists. The course of future depth-first searches can thus be determined by sorting the adjacency lists. In particular, it will be desirable to select, at each point in the search, the edge that will result in a path which returns to the vertex with the lowest possible number. Each path will thus return as far back toward the root of the spanning tree as possible at that point in the depth-first search. Other paths may return closer to the root when the search is returning from levels of recursion. If two edges lead equally far back in the graph, it will be necessary to break the tie by considering which edge leads to the next lowest numbered vertex.

In Figure 12 vertex 6 has vertices 7, 8 and 3 in its adjacency list (vertex 5 is no longer adjacent since the graph was converted to a digraph). The lowest numbered vertex that vertex 7 can lead to is vertex 2. Vertex 8 can lead to vertex 1 (via vertex 9) and vertex 3 terminates a path itself. Therefore the desired order for the adjacency list for vertex 6 is 8, 7, 3. The sorted adjacency lists for the graph in Figure 12 and the paths that result from the application of a depth-first search using those lists are shown in Table 3. A tracing of the graph, using the paths
### TABLE 3: Sorted Adjacency Lists for Graph in Figure 12 and Paths as Generated by a Depth-first Search

#### Sorted Adjacency Lists

<table>
<thead>
<tr>
<th>Node</th>
<th>Adjacent Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>8, 7, 3</td>
</tr>
<tr>
<td>7</td>
<td>2, 3</td>
</tr>
<tr>
<td>8</td>
<td>9, 5</td>
</tr>
<tr>
<td>9</td>
<td>1, 10</td>
</tr>
<tr>
<td>10</td>
<td>11, 8</td>
</tr>
<tr>
<td>11</td>
<td>4, 5, 9</td>
</tr>
</tbody>
</table>

#### Paths Generated Using Above Lists

1: 1, 2, 3, 4, 5, 6, 8, 9, 1
2: 9, 10, 11, 4
3: 11, 5
4: 11, 9
5: 10, 8
6: 8, 5
7: 6, 7, 2
8: 7, 3
9: 6, 3
in order, gives a hint as to how the algorithm can proceed. The possible conflicts between paths during the search have been localized somewhat.

The conversion to a relabeled palm tree and calculation of the lowest and next lowest vertices that can be reached from each vertex can all be done during the same depth-first search. See Algorithm 2.

3.3 Embedding The Graph In A Plane

The remainder of Hopcroft and Tarjan's algorithm is accomplished by traversing the palm tree with another depth-first search - this time using the ordered adjacency lists. The algorithm generates a cycle in the graph (the largest possible at that point in the depth-first search) and then embeds future paths on either the inside or the outside of that cycle. Simple mathematical tests are done to determine if the path embeddings interfere with each other. Paths longer than a single back edge are used to define a new cycle for the embedding of future paths. The algorithm is thus recursive in another sense besides that of the depth-first search being used to generate the paths.
ALGORITHM 2: Conversion of a Graph to a Palm Tree

SUBROUTINE PALM (V, U)
# Convert a graph to a palm tree, relabel the vertices
# and calculate the lowest and next lowest vertices
# reachable from each vertex. Start search at V.
# U is vertex that lead to V.
N = N + 1
NUMBER(V) = N
LOW(V) = N
NEXT = N
WHILE (There exists an unexplored edge, (V, W))
  $(
    IF (NUMBER(V) unassigned)
    $(
      # W is a new vertex
      CALL PALM (W, V)
      IF (LOW(W) < LOW(V))
        $(
          NEXT(V) = MIN(LOW(V), NEXT(W))
          LOW(V) = LOW(W)
          $
        )
      ELSE IF (LOW(W) == LOW(V))
        NEXT(V) = MIN(NEXT(V), NEXT(W))
      ELSE
        NEXT(V) = MIN(NEXT(V), LOW(W))
    )
    ELSE IF (NUMBER(W) < NUMBER(V) & W \= U)
    $(
      # (V, W) is a back edge
      IF (NUMBER(W) < LOW(V))
        $(
          NEXT(V) = LOW(V)
          LOW(V) = NUMBER(W)
          $
        )
      ELSE IF (NUMBER(W) > LOW(V))
        NEXT(V) = MIN(NEXT(V), NUMBER(W))
    )
  ELSE
    # Edge is not part of the spanning
    # tree and is not a back edge.
    DELETE from the graph
  )
RETURN
END
The data structures that are used to keep track of the information being generated are very important for efficiency considerations. These data structures are built out of stacks and linked lists. Linked lists were described earlier; stacks are last in, first out structures. When an element is added to a stack it is said to be pushed onto the stack. When it is removed, it is said to be popped off the stack.

Two stacks, called the inside and outside stacks, are maintained to hold the end vertices of paths embedded on the inside and the outside of their respective cycles. A vertex is pushed onto a stack when the path that it represents is embedded. A vertex is popped off the stack when the path that it represents could not possibly interfere with any path left to be generated. Because of the ordering of the adjacency lists, the two stacks will also be ordered. The highest numbered vertices will be on the tops of the stacks.

A third stack, called the bundle stack, is used to hold pairs of pointers to elements on the inside and outside stacks, thereby partitioning those stacks into bundles. A bundle is a maximal set of stack entries such that the embeddings of the paths that they represent are mutually dependent. That is that embedding any path represented in the bundle determines the embeddings of all other paths.
represented in the bundle. The ordering of the adjacency lists guarantees that the bundles will consist of only adjacent entries on the inside and outside stacks.

During the course of the algorithm it will sometimes be necessary to swap sides of a bundle. So that this can be done efficiently, the inside and outside stacks are implemented as linked lists. The pointers on the bundle stack point to the bottom list elements on the inside and outside stacks that belong to that bundle. See Figure 13. Swapping sides of a bundle is then a matter of swapping two sets of pointers. The bundle stack need not be implemented as a linked list since all operations on it will be done by pushing and popping the pairs of pointers.

To illustrate the basic operation of the algorithm consider the simple graph in Figure 14. The first path generated is used to define a cycle; paths 2 and 3 must then be embedded on the inside or outside of that cycle. When path 2 is generated all stacks are empty so vertex 3 (the finish vertex of path 2) is pushed onto the inside stack and a bundle is created that points to the new entry on the inside and has a null pointer for the outside. When path 3 is generated, an attempt is made to embed it on the inside. However, its finish vertex is number 2 which is less than the top entry of the inside stack (which is 3).
FIGURE 13: Data Structures for the Planarity Algorithm

Empty bundle side is indicated by a circle. The area inside the dotted line represents a typical bundle.
Ordered Adjacency Lists  

1: 2  
2: 3  
3: 4  
4: 5, 2  
5: 1, 3  

Resultant Paths  

1: 1, 2, 3, 4, 5, 1  
2: 5, 3  
3: 4, 2  

FIGURE 14: Graph with No Branches
Automatic Flowsheet Drawing

This test ($2 < 3$) indicates that the two paths involved would interfere. Figure 15 demonstrates the situation. The algorithm handles the situation by swapping the sides of bundles as necessary to allow embedding the current path on the inside. Figure 16 shows the result.

So far elements have been added to the inside stack and moved to the outside stack when necessary. It is also necessary to remove stack entries when they can no longer interfere with paths left to be generated. If this were not done, the stacks would not remain properly ordered.

When the depth-first search returns from a level of recursion, it is backing up an edge of the spanning tree to a lower numbered vertex. The entire subtree of the spanning tree with that edge as its first edge has been completely explored. Thus any entries on either stack that are greater than or equal to the number of the vertex being returned to can be eliminated. Consider Figure 14 for an example. When the depth-first search returns to vertex 3 after embedding paths 2 and 3, the algorithm must still consider the possibility of back edges from 3 to 2, 2 to 1, et cetera. Path 2 (from 5 to 3) could not possibly interfere, so its stack entry can be deleted (popped off the stack). Path 3 (4 to 2), however, could still be important.
FIGURE 15: Conflict Between Paths 2 and 3 of the Graph in Figure 14

To embed a path on the inside, its finish vertex must be greater than or equal to the top entry on the Inside Stack. The dotted line illustrates the situation when that is not so.
FIGURE 16: Final Embedding of Graph from Figure 14

The bundle documents the inter-dependence of paths 2 and 3.
One major complication still remains. When a path consists of more than a single back edge, future paths will have their start vertex on that path rather than on the original cycle. For this reason, the algorithm proceeds by using the new path to define a new cycle and to embed future paths with respect to that new cycle. The new cycle is formed by the path itself plus that part of the original cycle that extends from the finish vertex of the path to the start vertex. Figure 17 shows a graph that requires such recursive application of the algorithm. As usual the first cycle is defined by the first path. The second cycle is defined by path 2 plus the subpath (2, 3, 4) from the first cycle.

The recursion defined above is not implemented explicitly in the algorithm. Instead, when embedding a path that is more than a single back edge, an end of stack indicator is placed on the outside stack. When the depth-first search returns to the first vertex of the path while returning from its levels of recursion, the end of stack indicator is popped off the stack. It is at this time that paths that have been embedded with respect to the new cycle are reconciled with the old cycle. Any stack entries that are above the bundle containing the end of stack must be embedded on the inside. Figure 18 shows why. Basically, any back edges embedded on the outside of the new cycle
Ordered Adjacency Lists

1: 2
2: 3
3: 4
4: 1, 5
5: 6, 3
6: 2

Resultant Paths

1: 1, 2, 3, 4, 1
2: 4, 5, 6, 2
3: 5, 3

FIGURE 17: Graph Requiring Recursive Application of the Algorithm
FIGURE 18: Returning from Recursive Application of the Planarity Algorithm

All paths must be embedded on the inside. The back edge from 5 to 3 must be embedded inside cycle 2 regardless of how cycle 2 is embedded with respect to cycle 1.
would interfere with the original cycle. Therefore, the sides of any bundles that have outside entries are swapped.

The complete algorithm is shown in Algorithm 3. Two changes have been made to Hopcroft and Tarjan's algorithm.

1. Rather than testing for planarity at the places indicated by comments in Algorithm 3, this version charges on anyway. A flowsheet will be drawn even if it turns out to be mathematically non-planar.

2. Hopcroft and Tarjan test to avoid adding a path's entry to the inside stack if it cannot possibly interfere with future paths. Algorithm 3 always makes the entry so that it will be available for inclusion in the dependency graph as described in the next section.

Algorithm 3 deletes stack entries when they are no longer important for the embedding of future paths. The algorithm will be modified in the next section to save what has been learned about the embeddings of the paths involved by constructing a dependency graph.

Details of the stack operations have not been shown in Algorithm 3 since their implementations may be influenced by the language used. Specific complete implementations are
ALGORITHM 3: Planarity Algorithm of Hopcroft and Tarjan

SUBROUTINE PLANAR
# Initialize EMBED's variables
INCLUDE COMMONS
# Set up a stack implemented as a linked list to hold
# paths embedded on each side of the current cycle.
INSIDE = empty stack
OUTSIDE = empty stack
# Set up a stack to hold pairs of pointers to partition
# the INSIDE and OUTSIDE stacks into bundles.
BUNDLE = empty stack
# Initialize a flag to indicate ready for new path.
# Normally set to number of start vertex of current path.
START = 0
# Initialize current path number.
THIS_PATH = 0
# Start Hopcroft and Tarjan's algorithm at vertex 1
CALL EMBED (1)
RETURN
END

SUBROUTINE EMBED (V)
INCLUDE COMMONS
# Basic structure is a depth-first search starting at V.
WHILE (There exists another edge, (V, W))
  $(
    IF (START == 0)
    $(
      # New path.
      START = V
      THIS_PATH = THIS_PATH + 1
    )
  )
  IF (V < W)
    $(
      # (V, W) is a spanning tree edge.
      FIRST_PATH(W) = THIS_PATH
      CALL EMBED (W)
      DELETE all stack entries >= V and all
      bundles which contain only such entries
      (See Algorithm 4).
  )
IF (FIRST_PATH(W) \= FIRST_PATH(V))
$(
  # Reconcile stack entries based on
  # cycle formed by FIRST_PATH(W)
  # with cycle formed by FIRST_PATH(V).
  REPEAT
  $(
    POP top bundle off BUNDLE stack
    IF (Top entry of outside of bundle
        \= END_OF_STACK)
      & outside of bundle not empty)
    SWAP sides of bundle
    # (If inside of bundle not empty either,
    # graph is nonplanar)
  )
  UNTIL (Top entry of outside of bundle
          ==END_OF_STACK)
  POP END_OF_STACK off OUTSIDE stack
  PUSH bundle onto BUNDLE stack
)$(
ELSE
$(
  # (V >= W) ==> (V, W) is a back edge.
  FINISH(THIS_PATH) = W
  # Move bundles as necessary so that THIS_PATH
  # can be embedded on inside.
  WHILE (Top entry of either side of top bundle > W)
  $(
    POP top bundle off BUNDLE stack.
    IF (Top entry of inside of bundle > W)
      SWAP sides of bundle
    # (If top entry of outside of bundle also > W,
    # graph is nonplanar)
  )
  PUSH W onto INSIDE stack.
  IF (V \= START)
    # New cycle formed by THIS_PATH
    PUSH END_OF_STACK onto OUTSIDE stack.
    # Ready for new path.
    START = 0
  )$
RETURN
END
available in Hopcroft and Tarjan (1974) and Reingold, Nievergelt and Deo (1977) as well as in Appendix C of this dissertation. The remainder of this section will be devoted to an example that should make the stack operations clear.

Figure 12 shows the example graph from Reingold, Nievergelt and Deo after conversion to a spanning tree with back edges. Table 3 shows the paths that are generated as the algorithm proceeds. At the end of the first path, the end vertex (1) is pushed onto the inside stack. Since path 1 is longer than a single back edge (as determined by the test \( V = \text{START} \)), an END-OF-STACK is pushed onto the outside stack. A bundle is then pushed onto the bundle stack to leave the stacks as shown in Figure 19. The algorithm as specified by Hopcroft and Tarjan would not push the end vertex of the first path onto the inside stack. This part of the algorithm has been modified to provide more information for the algorithms to be developed in the next section. When path 2 is generated by the depth-first search, vertex 4 and another END-OF-STACK are pushed onto the inside and outside stacks respectively, and another bundle is pushed onto the bundle stack. The algorithm will now be embedding paths with respect to the cycle \((4, 5, 6, 8, 9, 4)\). The situation at this time is also shown in Figure 19. The embeddings of paths 3 and 4 are also done without any need to swap sides of bundles. Since both paths
FIGURE 19: Contents of Stacks at Various Points in the Progress of the Planarity Algorithm

An empty bundle side is indicated by a circle. EOS stands for an end-of-stack marker.
FIGURE 19 (continued)

After embedding path 5

After backing up edge (9, 10)
FIGURE 19 (continued)

After embedding path 6

After embedding path 7
FIGURE 19 (continued)

After backing up edge (6, 7)

After embedding path 9
Automatic Flowsheet Drawing

are single back edges, END-OF-STACK's are not needed; the algorithm is still using the cycle formed by path 2. Figure 19 shows how two more bundles have been added by the two paths.

Since the depth-first search has now explored all edges leading from vertex 11, it must return from a level of recursion before embedding path 5. Since there are no stack entries greater than or equal to the vertex being returned to (10), and since the edge (10, 11) was not the first edge of a new cycle, however, no adjustments to the stacks need be made.

Path 5 is the first path to interfere with an existing stack entry. Since its end vertex (8) is less than the top entry of the inside stack, the top bundle is popped from the bundle stack, and the sides of the bundle are swapped. The top entry of the next bundle does not interfere, so vertex 8 (the end vertex of path 5) is pushed onto the inside stack and the resultant combined bundle is pushed onto the bundle stack. Notice that the bundle was combined to reflect how the embeddings of the paths involved depend on each other. Figure 19 shows the resultant structure of the stacks.

Since all edges from vertex 10 have now been explored, the depth-first search must return from recursion to vertex 9. Since the top element on the outside stack (9) is
greater than or equal to the vertex being returned to (9), that element is popped from the stack. Since other elements in that bundle are less than the new vertex, however, the bundle is not deleted. Also, since the edge (9, 10) was the first edge of a new cycle (formed by path 2), bundles are popped off the stack until an END-OF-STACK is reached. Since none of these bundles have any outside entries, however, it is not necessary to do any swapping. The END-OF-STACK is popped off the outside stack and the combined bundle is pushed back onto the bundle stack. Note once again that the bundles are combined to show that the paths involved are mutually dependent; they all must be embedded on the inside of the cycle defined by path 2. See Figure 19.

All edges from vertex 9 have also been explored, so the search backs up the edge (8, 9) to vertex 8. The top entry on the inside stack is greater than or equal to the new vertex number and is therefore deleted. The edge (8, 9) was not the first edge of a new cycle, so the algorithm can proceed with path 6. The top entry of neither the inside nor the outside stack is greater than the end vertex of the path, so a new bundle with that end vertex on the inside stack is added with no problem. See Figure 19 for the current situation.
All edges from vertex 8 have now been explored, so the algorithm backs up to vertex 6. No stack entries are greater than or equal to 6, and the edge (6, 8) was not the first edge of a new cycle, so the stacks are not modified. Path 7 ends with vertex 2, however, which is less than the top element on the inside of the top two bundles. Both bundles are therefore popped from the bundle stack and their sides are swapped. Vertex 2 is pushed onto the inside stack to represent path 7 and since the path is more than a single back edge, an END-OF-STACK is pushed onto the outside stack (future paths will be embedded with respect to a cycle defined by path 7). The combined bundle is pushed onto the bundle stack to give the structure shown in Figure 19.

The embedding of path 8 is straightforward and adds another bundle consisting of path 8's entry on the inside and nothing on the outside. The edge (6, 7) is the first edge of a new cycle, so before path 9 can be embedded the stacks must be manipulated to account for the change in cycles. The top bundle is popped off the bundle stack but since the outside is empty, no swap takes place. The next bundle is popped off the bundle stack, but since the top element on the outside is an END-OF-STACK no swap takes place. The END-OF-STACK is popped from the outside stack and the combined bundle is pushed back on the bundle stack. The situation is shown in Figure 19.
The embedding of path 9 illustrates one final point. The end vertex is 3 which is greater than the top entry on the inside stack, so there is no interference there. The top entry on the outside stack, however, is greater than 3, so the bundle is popped from the bundle stack before pushing path 9's entry on the inside stack. When the bundle is pushed back on the stack, then, the fact that path 9 must be embedded opposite path 5 has been documented.

The final set of stacks after all paths have been embedded is shown in Figure 19. As the depth-first search backs up the spanning tree to vertex 1, each entry is deleted.

3.4 Labeling The Back Edges

During the course of Algorithm 3, dependencies are discovered between various paths. Since there is a one-to-one correspondence between paths and back edges, the dependencies can be thought of as being between the back edges. Algorithm 3 documents these dependencies by making an entry on a stack for each back edge and then maintaining bundles of mutually dependent stack entries. When a stack entry can no longer interfere with future stack entries, it is deleted. It can still end up dependent on a future entry, however, by being dependent on an entry that will
become dependent on that future entry as the algorithm proceeds.

The information being generated by Algorithm 3 can be converted to a more useful form by creating a new graph, called the dependency graph. The vertices of the dependency graph will represent the paths (or back edges) from Algorithm 3. The edges will represent dependencies and will be labeled according to whether the back edges involved should return on the same side of the spanning tree or on opposite sides. The stacks of Algorithm 3 hold entries that are embedded on the inside or the outside of some reference cycle. That cycle can be thought of as returning on the left or right side of the spanning tree (with respect to, say, an observer standing at the root looking toward the tree). All entries on the inside stack would then also return on the left while entries on the outside stack would return on the right.

The dependency graph can be constructed by taking each stack entry as Algorithm 3 deletes it, and linking it into the dependency graph. If there is another entry on the same side of its bundle, it is linked to it, and the (undirected) edge is labeled SAME. If it is the last entry on its side of the bundle and the other side is not empty, it is linked to the bottom entry of that side with an edge labeled
If it is the last entry of the bundle, no new edge is added to the dependency graph. Note, however, that other links to the back edge may already have been made when previous entries were deleted. Algorithm 4 details the process and Figure 20 shows the dependency graph for the example of the last section.

The usefulness of the dependency graph is that the labeling of its edges can be transformed into a labeling of its vertices, which correspond to the back edges of the original graph. In particular, each back edge can be labeled with information that tells how to draw it with respect to the spanning tree. This information could be left versus right, top versus bottom, or clockwise versus counter-clockwise, depending on the frame of reference chosen.

The conversion of the dependency graph from edge labeling to vertex labeling can be done easily by any algorithm that traverses the graph. Algorithm 5 uses a depth-first search. Figure 21 shows the result of converting the dependency graph in Figure 20, and Figure 22 shows the corresponding planar representation for the original graph.
ALGORITHM 4: Creating the Dependency Graph.

# Replace section indicated in Algorithm 3.
REPEAT
  $(
P OP a bundle from the bundle stack.
  FOR (Each side of the bundle)
    WHILE (Top entry >= V)
      $(
        # Can't interfere with future paths;
        # delete entry and link to dependency graph
        POP (Top entry from stack)
        IF (This side of bundle not empty)
          LINK to next entry with SAME edge.
        ELSE IF (Other side of bundle not empty)
          LINK to bottom entry of other side
          with OPPOSITE edge.
      )
  )
UNTIL (Bundle is not empty)
PUSH a bundle back onto the BUNDLE stack.
FIGURE 20: Dependency Graph for Graph from Figure 12 with Paths from Table 3
ALGORITHM 5: Converting the Dependency Graph from Edge to Vertex Labeling.

FOR (Each vertex, V)
   LABEL(V) = NONE

FOR (Each vertex, V)
   # Catch all connected components.
   IF (LABEL(V) == NONE & V has one or more edges)
      $(
      LABEL(V) = RIGHT
      CALL COLOR(V)
      )$

SUBROUTINE COLOR (V)
WHILE (There exists another edge (V, W))
   $(
   IF (LABEL(W) == NONE)
      $(
      IF (Edge (V, W) is SAME)
         LABEL(W) = LABEL(V)
      ELSE
      $(
      IF (LABEL(V) == RIGHT)
         LABEL(W) = LEFT
      ELSE
         LABEL(W) = RIGHT
      )
      CALL COLOR (W)
   )$
RETURN
END
FIGURE 21: Vertex Labeled Version of the Dependency Graph in Figure 20
FIGURE 22: Planar Representation for the Graph in Figures 11 and 12

Drawing was produced by the prototype system of Chapter IV.
4.0 RECONSTRUCTING THE FLOWSHEET

The combination of the breaking of a graph into paths by Hopcroft and Tarjan's algorithm, and the labeling of those paths via the use of the dependency graph, provides enough information to specify a particular planar representation for the graph. The information is sufficient, in fact, to make drawing the graph by hand easy. Computers are not as creative as people, however, and even more explicit information is needed for their use. The following sections define the remaining problems and outline solutions.

4.1 Reordering The Palm Tree

Hopcroft and Tarjan's algorithm defines cycles in a graph and embeds paths either inside or outside those cycles. If the graph is drawn using the same set of paths, some of those paths must be drawn inside existing cycles. Thus the cycles must be drawn before their required size is known.

In the development of Algorithm 3 it was pointed out that the identity of the paths to be generated was determined by the order of the edges on the adjacency lists used to represent the graph. The goal now is to sort those
adjacency lists so that the next depth-first search will result in a set of paths that correspond to drawing the graph from the inside out. That is that a path may encircle parts of the graph that have already been drawn, but will never have to be drawn inside a cycle that already exists. The way to do this is to order the edges on the adjacency list in descending order of the path number assigned in Hopcroft and Tarjan's algorithm (Algorithm 3). Thus any edge that was part of a path that had to be embedded inside a cycle will be drawn before the edge that was part of the path that defined that cycle. Table 4 shows the new set of adjacency lists for the graph in Figures 11, 12, and 22 along with the paths that result from a depth-first search using those adjacency lists. A tracing of the new paths, using the previously developed information on how to return the back edges, demonstrates how a planar representation for the graph can be drawn in the desired "inside-out" fashion.

4.2 Drawing The Abstract Graph

The basic plan for drawing a graph is to start with the root of the spanning tree at the left and always draw new vertices to the right of their immediate ancestor in the spanning tree. The basic shape of the graph will then be that of the spanning tree, with tree edges being drawn
### TABLE 4: Resorted Adjacency Lists for Graph in Figures 11, 12, 22 and Paths as Generated by a Depth-first Search

**Resorted Adjacency Lists**

<table>
<thead>
<tr>
<th>Node</th>
<th>Adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3, 7, 8</td>
</tr>
<tr>
<td>7</td>
<td>3, 2</td>
</tr>
<tr>
<td>8</td>
<td>5, 9</td>
</tr>
<tr>
<td>9</td>
<td>10, 1</td>
</tr>
<tr>
<td>10</td>
<td>8, 11</td>
</tr>
<tr>
<td>11</td>
<td>9, 5, 4</td>
</tr>
</tbody>
</table>

**Paths Generated Using Above Lists**

<table>
<thead>
<tr>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 1, 2, 3, 4, 5, 6, 3</td>
</tr>
<tr>
<td>2: 6, 7, 3</td>
</tr>
<tr>
<td>3: 7, 2</td>
</tr>
<tr>
<td>4: 6, 8, 5</td>
</tr>
<tr>
<td>5: 8, 9, 10, 8</td>
</tr>
<tr>
<td>6: 10, 11, 9</td>
</tr>
<tr>
<td>7: 11, 5</td>
</tr>
<tr>
<td>8: 11, 4</td>
</tr>
<tr>
<td>9: 9, 1</td>
</tr>
</tbody>
</table>
Automatic Flowsheet Drawing

horizontally. Back edges will consist of vertical lines from each of the vertices involved and a horizontal line connecting them. The vertical lines will be drawn long enough to allow the horizontal line to avoid all obstacles.

Only one problem stands in the way of automating the drawing of an abstract graph. When a path from Table 4 is drawn, all except the first and last vertices of the path are new—they have not yet been drawn. The last vertex is there to define the back edge that terminates the path. The first vertex connects the current path with the part of the spanning tree that has already been drawn. If the first edge of the path is also the first spanning tree edge to be drawn from that vertex, the current branch of the spanning tree is simply extended in a straightforward way. See Figure 23 which shows how path 2 from Table 4 simply extends the branch of the spanning tree started by path 1. If, however, a spanning tree edge has already been drawn from the first vertex of the path, a new branch of the spanning tree must be started.

Two things must be known before the first vertex of a spanning tree branch can be drawn. First, a spanning tree branch clearly has a side associated with it—in the same sense that the back edges have sides associated with them. That side must be determined. Second, the separation of the
FIGURE 23: Relationship Between Paths and Branches

Path 2 from Table 4 simply extends branch started by Path 1. Path 4 starts new branch. Path 3 was a single back edge.
new branch from the existing branch must be determined. The necessary separation will be determined by the number of back edges that must be drawn between the two branches.

The side associated with a new branch has already been determined, but care must be taken to remember the answer. As Algorithm 3 runs its course, a path number can be assigned to each edge of the graph. When the vertices of the dependency graph are labeled with a side, it is really the path from Algorithm 3 that is being labeled. Thus when a branch of the spanning tree is encountered as described in this section, its side is really the same as the side assigned to the path of which the branches first edge was a part when processed by Algorithm 3. It is important, therefore, to distinguish between paths generated by Algorithm 3 and paths generated from the reordered adjacency lists during the drawing process.

The separation required between the new and old branches of the spanning tree is a little harder to determine. Since a depth-first search is being used to do the drawing, all of the original branch has been drawn by the time the new branch is started. The size of the new branch, however, will not be known until the depth-first search, returning from levels of recursion, returns to the branch point. For this reason two passes are apparently
Automatic Flowsheet Drawing

required to do the drawing. On the first pass vertices can be positioned relative to the first vertex of their branch (branches can, of course, be nested arbitrarily deep). The first vertex of a branch can be positioned relative to the first vertex of the branch it is branching from. On the second pass of a depth-first search, then, absolute positions can be calculated for all vertices. The details of such a two pass process are very dependent on particular decisions about the representation of a vertex and are dominated by book-keeping chores associated with vertex positions and spacings. An example implementation can be found in Appendix C.

The ideas outlined above and demonstrated in Appendix C are sufficient to draw a graph with reasonably good results. Future work could provide a definite improvement in two areas, however.

When a back edge is being positioned, the system in Appendix C always draws the vertical lines long enough to clear anything in the current branch of the spanning tree. This is actually necessary when the back edges on that side of the spanning tree are all nested—a new back edge must encircle all of the ones already drawn. If this is not the case, and the system were smart enough to determine what actually was between the two vertices involved, a better
graph could be drawn. The problem is worse when a back edge is being drawn after branches have already been drawn from the branch associated with the back edge. If Figure 22 had a back edge from vertex 3 to vertex 1, it would not be drawn until the depth-first search was backing down the main branch having already drawn the rest of the graph. The result would be the graph shown in Figure 24.

The second area that could be improved is the way that branches are merged. The system in Appendix C, when returning from recursion and backing up from a branch to its predecessor, positions the new branch so that it avoids all contact with the earlier branch. That is, the system treats branches as rectangular shapes in this situation just as when it positions back edges. The same problem results. Though the resulting graph is acceptable, it could be better. Figure 25 demonstrates the situation.

4.3 Drawing The Flowsheet

A flowsheet can be drawn in basically the same way as an abstract graph. Several complications must be overcome, however. First, a process unit symbol is represented by several vertices and edges connecting them. This set of vertices and edges must be combined as the drawing algorithms proceed. Second, the order in which the
FIGURE 24: Graph in Figure 22 with an Added Edge (3, 1)
System in Appendix C does this:

\[ \text{3rd Branch} \quad \text{2nd Branch} \quad \text{1st Branch} \]

This would be better:

\[ \text{3rd Branch} \quad \text{2nd Branch} \quad \text{1st Branch} \]

Must be able to do this:

\[ \text{3rd Branch} \quad \text{2nd Branch} \quad \text{1st Branch} \]

FIGURE 25: Branch Placement Problem
Automatic Flowsheet Drawing

algorithms consider the vertices that make up the symbol, and the side on which any back edge involved falls, determine whether the unit symbol should be left or right handed. Third, the connections between the symbol and the process streams (that is, the vertices of the graph and those edges which model process streams) have a direction associated with them. This is not a direction in the sense of the directed edges of a digraph, or in the sense of material flow, but rather a geometrical direction associated with the design of the symbol. For example, Figure 26 shows a left and a right handed column and a graph that models them. In the right handed version the feed stream is directed to the left and, in the left handed version, to the right.

The first problem, combining the vertices and edges of a unit symbol, is easy. The depth-first search of each pass of the drawing algorithm will continue to process the abstract graph. When each vertex is encountered a check will be made to see if its process unit symbol has been drawn yet. If not, the symbol will be drawn; otherwise, the depth-first search will continue with no action taken. When a back edge is processed, a check will be made to see if both vertices involved are from the same unit. If they are not, a back edge (representing a process stream) will be drawn; otherwise, the search will continue.
FIGURE 26: Handedness of Units
Figure 26 shows how the hand of a unit symbol is determined by a combination of the order in which the graph is traversed and the side on which the back edge involved returns. The point is that the entire symbol must be traversed before its hand can be determined. This causes a problem with the method proposed above for combining the vertices and edges of a single symbol. Apparently a third pass is necessary to determine the hand of each symbol. The plan now is for pass one to assign a hand to each process symbol, pass two to position each symbol relative to the first symbol of its branch, and pass three to do the actual drawing after calculating absolute positions for the symbols.

The third problem, that of the unit symbol-process stream intersections having directions associated with them, causes a big programming headache. To understand the problem, refer to Figure 27. If the drawing program uses a convention of positioning a new unit symbol to the right of the branch as drawn so far, the new edge (process stream) will be coming into the new unit from the left. If symbol-stream intersections are restricted to having only four possible directions, a convention can be chosen for the route that the edge will take to get to each one as shown in Figure 27. For the branch to continue with another symbol, a process stream must leave the symbol and travel to the
New units can be drawn four ways:

![Diagram](image)

Branches can be continued twelve ways:

![Diagram](image)

**FIGURE 27:** Combinations of Entering and Exiting Spanning Tree Edges for a Symbol
right. The route that it takes depends on the route that the stream entering the unit took. In fact, Figure 27 shows the twelve possible combinations. Fortunately, if the convention shown in Figure 27 for the entering edge is strictly followed, only four of the possible combinations involve the exiting edge not traveling directly to the right. These are vertices 6 through 9. The programmer's troubles are not over yet, however. It is also possible that recycle streams will have to be connected to each of the twelve combinations in Figure 27. Once again, however, if the conventions outlined so far are followed strictly, only vertices 5 through 9 can force a recycle stream to take a route other than straight to where it is going. The above discussion has ignored two special cases. The first vertex, the root of the spanning tree, has no edge entering from the left and the last vertex of each branch has no edge exiting to the right. It is thus possible that a back edge will have to take a special route to get where it is going.

Additional work could possibly produce a set of drawing conventions that would ease the programming job described here. Even these ideas make the job manageable, though certainly not fun. Appendix C provides an example implementation.
One additional point can be made that concerns the directional properties of the symbol-stream intersections. It turns out to be a fairly common occurrence that the algorithms presented in this dissertation will not require a particular hand for a particular unit symbol. In this case additional heuristics can be used to choose the hand on artistic grounds. For example, the selection of a particular hand for the heat exchanger in Figure 28 eliminates a zig-zag effect.

5.0 ORDERING THE ORIGINAL GRAPH

This chapter has described how to model the topology of a flowsheet with an abstract graph. The first step after obtaining that graph was to break it down into a spanning tree (acyclic) and a set of back edges that identified the cycles in the graph. The remainder of the processing sorted the adjacency lists for the graph several times to achieve various goals. The identity of the spanning tree, and therefore the ultimate branch structure of the flowsheet, however, remained fixed due to the labeling of the vertices during the very first depth-first search. Two factors predetermined the identity of that spanning tree and the corresponding structure of the flowsheet: the start vertex of the first depth-first search (which became the root of
FIGURE 28: Choosing Hand of Unit to Avoid Zig-Zag
the spanning tree) and the order of the edges on the adjacency lists of the original graph.

Both of the above factors have an impact on the artistic merit of the flowsheet. Although they were not used in the prototype system presented in the next chapter, methods will be presented here which manipulate these factors to improve the flowsheets produced.

A flowsheet produced by the methods of this chapter will have a tree shaped outline with the root on the left. Ideally, it would be nice to think of the raw materials entering on the left with material flowing generally to the right and recycle streams returning from right to left. If a computer program blindly models the flowsheet with a graph, however, this is not likely to happen. Two things can be done to help the program prefer such a representation, however.

First the user could be allowed to select the unit that should be used for the root of the spanning tree. It would not be necessary to have the computer ask every time the flowsheet were to be redrawn. Instead, selecting the first unit of the flowsheet could be made an option that the user would use whenever necessary. Normally the user would specify the source unit of the major raw material for the process - a unit that would not be expected to change often.
Second, immediately after deriving the graph that models the flowsheet, before any further processing, the computer could sort the adjacency lists in the following order: (1) edges representing streams leaving the unit, (2) edges internal to the unit symbol, (3) edges representing streams entering the unit. Thus, when the spanning tree was derived, the algorithm would, in effect, try to make the material flow from left to right, up the spanning tree. Material would flow right to left only as a last resort.
CHAPTER IV
THE PROTOTYPE SYSTEM

A prototype system was built to illustrate how automatic flowsheet drawing could be used in a computer aided design system. The system does not do simulations but does allow the user to interactively build and modify flowsheets in the same way that the proposed computer aided design system would. A Digital Equipment Corporation PDP-15 system, with a VT-15 graphics processor, run by the Chemical Engineering Department of the Ohio State University, was used for the development of the system. Appendix B gives a description of the graphics hardware and software. Appendix C provides listings.

1.0 USER INFORMATION

When the prototype system is first activated the user sees the screen image shown in Figure 29. The large blank area will contain the flowsheet as the user works. The bottom line will be used for messages to the user and titles. It initially identifies the system as the GRASP system – for Graphics Assisted Synthesis of Processes. The right hand column will always contain a menu listing all operations available to the user at the time. The menu
FIGURE 29: Initial Screen Contents
The Prototype System

driven operation of the system is a key feature; it allows a system that is much easier to use and understand than one based on fixed input formats or problem oriented languages.

The way a menu is used depends on the type of graphics hardware used. The graphics unit used for this system is a fully refreshed random scan unit that has a light pen. This allows the user to identify an item on the screen by pointing at it. With other types of graphics units, the user would position crosshairs with a joystick or perhaps enter an item number at a keyboard. For this particular system a utility subprogram (MENU) was written that accepts up to 25 text strings as input parameters and returns the index of the one selected by the user.

The top two menu items in Figure 29 allow a flowsheet to be saved in, and restored from, a named disk file. This allows the user to work on a flowsheet in as many separate sessions as desired. Naturally, a commercial system would save all the process data (unit parameters, et cetera), not just that necessary for the display. Also, files for existing processes would be saved permanently so that later process improvement studies could be done without the overhead of the initial process specification.
The next four items on the menu allow development and modification of flowsheets by adding and deleting units and streams.

The next item, DRAW FLOWSHEET, causes the system to do just that. Experience with the system indicates that it is convenient for the user to make a number of related changes before redrawing the flowsheet. For example, the user will probably want to add a new unit and its stream connections at the same time.

DRAW GRAPH draws the abstract graph used by the algorithms of the previous chapter. This function would not be needed by a commercial system but is invaluable for debugging purposes.

ADD TITLE allows the user to specify a text string to be displayed at the bottom of the screen. This feature can be used to label plots before they are made.

MAKE COPY produces a hard copy plot of whatever is on the screen when the selection is made.

EXIT does just what it says - terminates the session with the user.
When ADD UNIT is selected, the menu is immediately replaced by one that identifies the types of process units known to the system (pump, valve, et cetera). When the user selects one of the unit types, the symbol for that unit is displayed on the left side of the screen, immediately above the existing flowsheet. The user is then asked to supply a name for the unit. This name is used to identify the unit from then on and is always displayed with the symbol. A commercial system would also collect unit parameters at this time. Figure 30 shows the situation after a flash drum has been added and named, and ADD UNIT selected once again.

When ADD STREAM is selected, the system draws a light pen button at each possible symbol-stream intersection that has not yet been connected. A light pen button is a part of the display that is drawn while the light pen is enabled—that is, the computer will respond when the user points the light pen at that part of the display. For the ADD STREAM function a small triangle is used for the light pen button that is clear on the screen but only a black dot on the hard copy plots. The user is then prompted via messages at the bottom of the screen to select two of the buttons to specify the end points of a stream. Figure 31 shows the situation after a flash drum and a pump have been added, a stream added from the bottom of the flash drum to the input of the pump, and ADD STREAM selected again. There are no light pen
FIGURE 30: After Selecting ADD UNIT
buttons at the bottom of the flash drum and the pump input since they have already been connected.

Figure 32 shows the result of selecting DRAW FLOWSHEET after connecting the output of the pump to the input of the flash drum in Figure 31. The flowsheet is centered left to right but is placed near the bottom of the display to leave room for units added by the ADD UNIT function.

Figure 33 shows the result of selecting DRAW GRAPH for the flowsheet of Figure 32. In this case the planarity algorithm had no interfering cycles to worry about, so the hands of the units were not important. The system did have to do all the book-keeping associated with spacing the units and lining up the streams, however.

During the operation of the system the user is continually prompted and supplied with information. For example, the message "FLOWSHEET IS NOT UP-TO-DATE" as seen in Figure 32 is used whenever changes have been made and DRAW FLOWSHEET has not yet been selected.

Figure 34 shows a more complicated situation. The flowsheet is still a single cycle but now a pump is going to be added between the bottom of the reactor and the tee. When DRAW FLOWSHEET is selected, the system produces the flowsheet shown in Figure 35. This time the planarity
FIGURE 32: After Selecting DRAW FLOWSHEET
FIGURE 33: After Selecting DRAW GRAPH
FIGURE 34: Adding a Unit to an Existing Flowsheet
FIGURE 35: After Adding Streams to Flowsheet in Figure 34
The Prototype System

algorithm has to do some work as demonstrated by Figure 36. Vertices 1 and 2 are modeling the heat exchanger. The graph shows that the hand for this unit is not important for the planar representation—a fact that can be verified by considering the flowsheet. The same thing is true for the pump which is modeled by vertices 6 and 7. Note, however, that drawing the pump facing the other way would have produced an undesirable zig-zag effect. The reactor is modeled by vertices 3, 4 and 5 and the tee by vertices 8, 9 and 10. The graph in Figure 36 shows that the hands chosen for those units are required for the sake of planarity.

The stream from the output of the tee to the input of the reactor in Figure 36 was drawn as a back edge of the abstract graph. The extra 90 degree bend was caused by the drawing algorithm ensuring that it missed all possible obstructions by moving to the outside of the rectangular area defined by the limits of the flowsheet. A more sophisticated drawing program that would avoid the extra bend seems quite feasible.

Figure 37 shows the result of selecting DELETE UNIT. The system draws a light pen button for each unit—this time, a line under the unit name. Figure 38 shows the result of deleting the pump and selecting DRAW FLOWSHEET again. When deleting a unit, the system must also delete
FIGURE 36: Graph for Flowsheet in Figure 35
FIGURE 37: After Selecting DELETE UNIT
FIGURE 38: After Deleting Pump from Figure 37
any streams connected to that unit.

The DELETE STREAM function operates very much like the ADD STREAM function. For delete, a light pen button is drawn at every symbol-stream intersection that is connected to another symbol. The user can select either end of the stream with the light pen.

During development of the graph drawing algorithms, the abstract graphs produced by the DRAW GRAPH function are easier to interpret than flowsheets. Even so, most of the same considerations are present. For this reason, the prototype system allows the drawing of graphs generated by a separate program as well as those derived from a flowsheet. When DRAW GRAPH is selected a menu appears with the choices RESTORE GRAPH and THIS FLOWSHEET. Figure 39 shows the example graphs from Hopcroft and Tarjan (1974) and Reingold, Nievergelt and Deo (1977).

2.0 PROGRAM STRUCTURE

An outline of the subprogram structure of the GRASP system is given in Table 5. The main program initializes the data structures and displays an initial message and a menu of the functions that the GRASP system can perform. It then calls a subroutine determined by which of the functions
From Hopcroft and Tarjan (1974)

From Reingold, Nievergelt and Deo (1977)

FIGURE 39: Example Abstract Graphs
TABLE 5: GRASP System Structure

GRASP - Initialize and display menu for next action
UPINI - Initialize display files for unit symbols
MSG - Put initial message at bottom of screen
MENU - Have user select next action
OLNK1 - Following are in an overlay together
SAVEF - Save flowsheet
RSTRF - Restore flowsheet
ADDU - Add a process unit
DELU - Delete a process unit
  ISLCT - Have user select which unit
  MSG - Flowsheet not up-to-date
ADDS - Add a process stream
  ISLCT - Have user select connections
  MSG - Flowsheet not up-to-date
DELS - Delete a process stream
  ISLCT - Have user select which one
  MSG - Flowsheet not up-to-date
ADDT - Add a title to bottom of screen
  MSG - Display that title
DRAWF - For Draw Flowsheet or Draw Graph
VTSIM - Make hard copy

UPINI - Initialize display files for unit symbols
IPMP - For pump
  IDRAW - Produce display file
ICLMN - For column
  IDRAW
  
ICMP - For compressor
  IDRAW

VTSIM - Copy VT-15 screen to HP plotter
VTSUBS - Assembly language for absolute addresses
BVECS - Basic vectors
  VTPLT - VT-15 to HP
  HPLT - Plotter driver
PCHAR - Characters
  BVECS - Build from vectors
  VTPLT
  HPLT
VTPLT
  HPLT
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAWF</td>
<td>For Draw Flowsheet or Draw Graph</td>
</tr>
<tr>
<td>MENU</td>
<td>RESTORE GRAPH or THIS FLOWSHEET</td>
</tr>
<tr>
<td>DRPRP</td>
<td>Do planarity algorithm</td>
</tr>
<tr>
<td>RSTRA</td>
<td>If RESTORE GRAPH, do so</td>
</tr>
<tr>
<td>MAKEF</td>
<td>If THIS FLOWSHEET, derive graph</td>
</tr>
<tr>
<td>LINKA</td>
<td>Add an edge to the graph</td>
</tr>
<tr>
<td>ORDRA</td>
<td>Convert to palm tree and sort</td>
</tr>
<tr>
<td>EMBED</td>
<td>Create dependency graph</td>
</tr>
<tr>
<td>PSHAB</td>
<td>Push a bundle</td>
</tr>
<tr>
<td>POPAB</td>
<td>Pop a bundle</td>
</tr>
<tr>
<td>SWPAB</td>
<td>Swap sides of a bundle</td>
</tr>
<tr>
<td>DPEND</td>
<td>Add to dependency graph</td>
</tr>
<tr>
<td>DPLNK</td>
<td>Add an edge</td>
</tr>
<tr>
<td>POPAB</td>
<td>Pop a bundle</td>
</tr>
<tr>
<td>PSHAB</td>
<td>Push a bundle</td>
</tr>
<tr>
<td>COLOR</td>
<td>Convert to vertex labeling</td>
</tr>
<tr>
<td>ORDRB</td>
<td>Sort graph for drawing</td>
</tr>
<tr>
<td>DRGPH</td>
<td>If DRAW GRAPH, do so</td>
</tr>
<tr>
<td>ADRW1</td>
<td>Pass 1 to position vertices</td>
</tr>
<tr>
<td>ADRW2</td>
<td>Pass 2 to do drawing</td>
</tr>
<tr>
<td>PNT</td>
<td>Synonym for POINT</td>
</tr>
<tr>
<td>ABVRT</td>
<td>Draw a numbered vertex</td>
</tr>
<tr>
<td>DRFLW</td>
<td>If DRAW FLOWSHEET, do so</td>
</tr>
<tr>
<td>FDRWO</td>
<td>First pass to assign hands</td>
</tr>
<tr>
<td>FDRW1</td>
<td>Second pass to position units</td>
</tr>
<tr>
<td>BRPOS</td>
<td>First unit of a branch</td>
</tr>
<tr>
<td>UNPOS</td>
<td>Regular unit</td>
</tr>
<tr>
<td>FDRW2</td>
<td>Third pass to draw flowsheet</td>
</tr>
<tr>
<td>BRDRW</td>
<td>First unit of a branch</td>
</tr>
<tr>
<td>UNDRW</td>
<td>Regular unit</td>
</tr>
</tbody>
</table>
The Prototype System

was selected by the user. The call to OLNK1 is necessary only because of a quirk of the computer system used that makes it convenient to have only a single external symbol in an overlay.

Several utility subprograms were written specifically for this system. MENU was already mentioned - it provides a convenient way for a programmer to supply menu items and receive the one selected by the user. ISLCT is a similar subprogram that allows the programmer to draw his own light pen buttons and have ISLCT return a number indicating which one was chosen by the user. MSG provides a standard way of communicating with the user in an area of the screen dedicated to that purpose. VTSIM copies the contents of the screen onto a digital plotter. It does this by actually simulating the VT-15 graphics processor. Although VTSIM is very hardware specific, listings of it and the subprograms it calls are provided in Appendix C to illustrate the function of a graphics system at the lowest level.

In addition to the subprograms shown in Table 5, several routines are called to do the actual drawing on the graphics unit. These routines are part of the graphics software described in Appendix B.
CHAPTER V
CONCLUSIONS AND RECOMMENDATIONS

This dissertation has argued that the key to effective interactive computer aided design in chemical engineering is the ability to automatically draw flowsheets. The automatic flowsheet drawing problem was formalized and solved by resorting to graph theory. The key to this solution was that the flowsheet was modeled by a special abstract graph that included two types of edges: process streams and "artificial edges" to hold the unit symbols together. An algorithm was found in the literature that tested a graph for the mathematical property of planarity. This algorithm was extended to produce information about how to close each cycle in the graph to achieve a planar representation for the graph. Additional algorithms were then developed that allowed the graph to be drawn from the inside out with appropriate spacings. Reconstruction of the flowsheet required the development of methods to translate the abstract graph back to a flowsheet and to account for the directional properties of the unit-stream intersections.

A prototype system (described in the last chapter) was written which demonstrates the feasibility of using automatic flowsheet drawing for completely interactive
Conclusions and Recommendations

computer aided design in chemical engineering. The flowsheets are produced legibly and quickly. The use of interactive techniques, particularly the use of menus, combined with the display of the flowsheet, allowed the design of a system that can be used virtually without a manual. The benefits, when applied to a complete computer aided design system with the orientation and structure described at the beginning of this dissertation, should be great.

1.0 FUTURE WORK

The prototype system is just that— a prototype. It was built as a learning tool and was used to try out new ideas. As a result, some of its characteristics would be inappropriate for a commercial system. As a model computer program, it is showing the strain of the many revisions. It is recommended, therefore, that future work should start with a new version of the programs in Appendix C. Chapter III of this dissertation has the benefit of being written after the programs and naturally reflects a better understanding of the algorithms involved. Appendix C is valuable, however, for the extra detail and concrete examples that the programs provide. They should be studied together by one who hopes to continue the work.
The obvious goal for future work is to use the ideas of this dissertation in a computer aided design system. A practical first step might be to build a preprocessor that would create input files for an existing system. The following paragraphs identify some specific weaknesses of the prototype system that should receive attention during future development efforts.

The graph algorithms generally assume that the graph under consideration is biconnected, (there are at least two paths between every pair of vertices). This is the same as assuming that all streams in a flowsheet are part of some recycle loop. This restriction could be avoided by breaking the graph into biconnected pieces (algorithms for doing this are available in Reingold, Nievergelt and Deo (1977) for example), but that is probably not necessary. What is necessary is to guarantee that the planarity algorithm correctly handles the dead end paths that may result from a non-biconnected graph. An attempt was made to do this in the prototype system but was unfortunately not carried through to completion.

The implications of the directional property of the symbol-stream intersections discussed in Chapter III were realised only late in the development of the prototype system. As a result some "bugs" remain in this area.
Effort should be made to further formalize the ideas presented in this dissertation on establishing conventions for connecting streams to symbols. The object would be to minimize the number of combinations of connections that must be considered.

The ideas from Chapter III on having the user specify a "root" unit and having the system sort the original abstract graph to try to force material to flow from left to right were not implemented in the prototype system. Their inclusion would be straightforward, however, and should be done.

Only a small flowsheet can actually fit on the screen of a graphics terminal all at once. A commercial system would have to "draw" the flowsheet on a large abstract surface and display only part of it at a time. This technique is known as windowing. See Newman and Sproull (1979). The user would need to be able to easily move the window to view the whole flowsheet at will. This facility is best provided at the graphics system level and was not supported by the system used for this work. It should be considered a requirement for future work.

The drawing algorithms used by the prototype system always consider the flowsheet to have a rectangular shape with dimensions determined by the maximum horizontal and
Conclusions and Recommendations

vertical dimensions of the flowsheet. The result is very conservative routing of streams and placement of branches. This situation was discussed in Chapter III and illustrated in Figures 24 and 25. It seems likely that data structures could be devised to efficiently store information about the actual shape of the flowsheet. Of the recommendations for future work discussed here, this is the biggest job. Since satisfactory flowsheets can be produced without this refinement, it should be given the lowest priority.

2.0 OTHER APPLICATIONS

The graph drawing methods developed in this dissertation could conceivably find use in any application that uses graphs. Areas that come to mind immediately are laying out printed circuit boards, laying out integrated circuit masks, and drawing activity networks for PERT (Program Evaluation and Review Technique) or CPM (Critical Path Method) applications. Dynamic simulation programs and simulation programs from fields other than chemical engineering are other possibilities.

Many possible application areas provide their own unique set of constraints. Consider the printed circuit layout problem for example. If devices such as integrated circuits are modeled with symbols analogous to process units
Conclusions and Recommendations

and printed circuit track modeled analogous to process streams, the fact that all components must be mounted on the same side of the board amounts to constraining all symbols to have the same hand. Also, in the chemical engineering problem, streams are always connected between the output of one unit and the input of another. On a printed circuit board the output of one component may be connected to the input of several (fanout). It would be desirable for an automatic layout system to create and position the necessary track intersections automatically.

Another problem with laying out a printed circuit board would be that of keeping the layout inside the dimensions of the board. It would also be nice to improve the chemical engineering system to draw flowsheets within the confines of standard size engineering drawing paper. The issue is not critical for the chemical engineering system because of the hypothesized ability to interactively view the flowsheet through a movable window—a capability that has no impact on being able to lay out a printed circuit board. An intriguing idea would be to use the methods of this dissertation to provide input information to a conventional layout system.
Conclusions and Recommendations

The application of the ideas of this research to activity networks is interesting because of the interactive nature of the use to which these networks are put. The fact that the edges of these networks are directed and that the networks themselves are commonly laid out on a time scale adds an interesting twist to the problem.


Harary, F., *Graph Theory*, Addison-Wesley (1969)

List of References


Kernighan, B. W. and P. J. Plauger, Software Tools, Addison-Wesley (1976)


List of References


APPENDIX A
THE RATFOR PROGRAMMING LANGUAGE

The programs in this dissertation are written in RATFOR, an extension of FORTRAN. The rationale for its use is that it provides modern facilities that make the programming task easier and allow the production of programs that are easier to read and understand.

RATFOR is described in "RATFOR - A Preprocessor for a Rational FORTRAN" (Kernighan, 1975) and Software Tools (Kernighan and Plauger, 1976). Software Tools presents a RATFOR to FORTRAN translator as one of its tools; a production version of the translator is available on magnetic tape from the publisher, Addison-Wesley.

The following summary of RATFOR follows closely that given in the above references.
RATFOR provides four types of services to aid the programmer:

1. Modern control structures
2. A simple macro facility
3. A file inclusion facility
4. Input formatting

Aside from these services the language is FORTRAN.

1.0 CONTROL STRUCTURES

RATFOR's control structures allow a compound statement anywhere a single statement is legal. A compound statement is formed by enclosing it in braces. The version of RATFOR used in this dissertation uses $( and $) for braces. Thus

\[
\text{IF (condition)} \\
\quad \$( \\
\quad \quad X = 1 \\
\quad \quad Y = 2 \\
\quad \}$
\]

will cause both X and Y to be assigned values if the condition is true. The braces are not necessary if there is only one statement.
RATFOR provides seven control statements:

1. IF - ELSE
2. DO
3. WHILE
4. REPEAT - UNTIL
5. FOR
6. BREAK
7. NEXT

1.1 The IF - ELSE Statement

The following will execute statement 1 if the condition is true, statement 2 if the condition is false:

```
IF (condition)
  statement 1
ELSE
  statement 2
```

The ELSE construction is optional and is bound to the most recent IF that does not already have an ELSE.

The CASE statement of some languages can be simulated using ELSE IF. Thus

```
IF (condition 1)
  statement 1
ELSE IF (condition 2)
  statement 2
ELSE IF (condition 3)
  statement 3
```

will cause one and only one of the three statements to be
1.2 The DO Statement

The DO statement is the same as in FORTRAN except the availability of compound statements eliminates the need for a statement number at the bottom of the loop:

```
DO limits
    statement
```

Limits, for example could be I = 1, 10. The DO statement is appropriate when the number of iterations is known before the loop is entered (i.e. does not depend on calculations inside the loop).

1.3 The WHILE Statement

The WHILE statement sets up a loop with the test at the top:

```
WHILE (condition)
    statement
```

Thus if the condition is false to begin with, the statement will not be executed at all. Otherwise it will be executed repeatedly until the condition is false.
1.4 The REPEAT - UNTIL Statement

This statement sets up a loop with the test at the bottom:

\begin{verbatim}
REPEAT  statement  
UNTIL (condition)  .
\end{verbatim}

Thus the statement will always be executed at least once. If the condition is true, the statement will be executed again, and so on until the condition is false.

1.5 The FOR Statement

This statement sets up a loop with the test at the top and provides for an initialization and a reinitialization statement. Thus

\begin{verbatim}
FOR (initialize; condition; reinitialize)
statement
\end{verbatim}

is equivalent in most cases to

\begin{verbatim}
initialize
WHILE (condition)
$(
statement
reinitialize$
$)  .
\end{verbatim}

If the condition is omitted it is taken to be true.
1.6 The BREAK Statement

The BREAK statement causes an exit from the innermost loop that contains it. Control is transferred to the first statement after the loop. Thus

```
DO I = 1, 10
  IF (I .EQ. 5)
    BREAK
```

will exit with I equal to 5.

1.7 The NEXT Statement

The NEXT statement causes the innermost loop that contains it to go to the next iteration. Control is transferred to the condition test part of a DO, WHILE or REPEAT - UNTIL; the top of a REPEAT with no UNTIL; and the reinitialize of a FOR. Thus

```
J = 0
DO I = 1, 10
  $(
    IF (I .eq. 5)
      NEXT
    J = J + 1
  )$
```

exits with J equal to 9 (since the count at I equal to 5 was skipped).
2.0 MACRO FACILITY

The DEFINE statement allows simple alphanumeric string replacement - most typically for definition of symbolic constants. Thus

```
DEFINE(SIZE, 50)
DIMENSION X(SIZE), Y(SIZE, 3)
```

will yield the same results as

```
DIMENSION X(50), Y(50, 3)
```

The version of RATFOR used for this dissertation was modified to use the keyword MACRO in place of DEFINE (DEFINE was the name of a system directive on the machine used). The result is the same as if

```
DEFINE(MACRO,DEFINE)
```

were at the top of each file.

3.0 FILE INCLUSION

The INCLUDE statement allows the contents of a second file to be processed as if it were inserted in place of the INCLUDE. This is useful if two or more programs use duplicate sections of code - particularly COMMON declarations. Thus if a file named COMMONS exists and contains

```
COMMON /COM1/ X, Y, Z
```

the following subprogram will access X, Y and Z through
4.0 INPUT FORMATTING

RATFOR statements may be placed anywhere on a line. The translator will start FORTRAN statements on column 7, statement numbers on column 1, and so on. Each line will be assumed to contain one statement but multiple statements can be separated by semicolons. Comments begin with a # and can start anywhere on a line. Marginal comments are thus allowed.

The following table shows RATFOR symbols which may be used instead of the normal FORTRAN relational operators.

```
==  .EQ.
\=  .NE.
\  .OR.
&   .AND.
<   .LT.
<=  .LE.
>   .GT.
>=  .GE.
```

The symbol for .OR. is often printed as | or /

The RATFOR translator does not enforce any standards for indentation and placement of braces. In this dissertation, however, statements are always indented from
their controlling statements in a systematic fashion. Combined with the absence of any GO TO statements, this means that control always flows down and to the right. The sole exception to this is that ELSE IF's are always lined up to indicate their equality (one and only one will be executed).
APPENDIX B
THE GRAPHICS SYSTEM

The prototype system described in this dissertation was developed on a Digital Equipment Corporation PDP-15 computer with a VT-15 graphics processor. Support for interactive graphics was provided by the Graphics-15 set of FORTRAN callable routines running under the RSX-Plus III operating system. A utility subroutine for copying the contents of the graphics screen to a Hewlett Packard 7225A plotter was written during the development of the prototype system and is included with the listings in Appendix C. The applicable manuals from Digital Equipment Corporation are included in the List of References. The purpose of this appendix is to provide enough information about the graphics system to make the listings in Appendix C readable. For enough information to actually write programs, see the Digital Equipment Corporation manuals.

The VT-15 graphics processor operates by fetching instructions directly from the host PDP-15's memory, decoding them and generating analog signals to drive an x, y, z oscilloscope. The instructions consist of parameter setting, beam positioning, vector drawing, text drawing and
flow of control instructions. A steady image on the oscilloscope is obtained by the VT-15 looping through the same set of instructions 60 times per second. The instructions are placed in the PDP-15 memory and the VT-15 processor started and stopped by the execution of subroutines called by the user's FORTRAN program.

An area of PDP-15 memory used by the VT-15 processor is called a display file. The user must supply INTEGER mode arrays in the FORTRAN program for any display files used and initialize them by setting the first elements to zero. All other manipulations of the display files are done via subroutine calls.

The VT-15 processor is started with

CALL DINIT (DFILE)

where DFILE is the name of the integer array being used for the display file. The VT-15 processor is stopped with

CALL DCLOSE

Only three subroutines make entries in the display file that actually cause an image to appear on the screen. The beam is positioned to an absolute location on the screen by

CALL POINT (IX, IY, INT, DFILE, IEDIT)

The screen area can be thought of as an addressable area of 1024 by 1024 points with the origin at the lower left corner. In the call to subroutine POINT, the beam will be
The Graphics System

positioned to point \((IX, IY)\). The last three arguments are optional. If \(INT\) is used and set equal to 1, the point will be intensified. Otherwise, it will be invisible. \(POINT\) returns with \(EDIT\) set to a value that can be used later as an edit address for modifying the part of the display file that was just set. \(DFILE\) tells \(POINT\) which display file to use for this operation. If it is omitted, \(POINT\) uses the last display file that was specified.

Lines can be drawn on the screen by

\[
\text{CALL LINE (IX, IY, INT, DFILE, EDIT)}
\]

where, once again, only \(IX\) and \(IY\) are required. For this call, \(IX\) and \(IY\) are treated as relative movements in the \(x\) and \(y\) directions. The line will not be intensified only if \(INT\) is supplied and is equal to zero. \(DFILE\) and \(EDIT\) have the same uses as with subroutine \(POINT\).

Text can be displayed on the screen with

\[
\text{CALL TEXT (STRING, NCHAR, DFILE, EDIT)}
\]

where only the first two arguments are required. \(STRING\) must be a \(REAL\) array that holds the text string to be displayed. \(NCHAR\) specifies how many characters should be displayed. \(DFILE\) and \(EDIT\) have the same functions as for the other subroutines.
Once a display file has been built, it can be used as a kind of "sub-picture" for another display file:

```
CALL COPY (ISAVE, SUB, DFILE)
```

In this call, VT-15 code is added to DFILE to call SUB as a subroutine. If ISAVE does not equal zero, the VT-15 parameters will be saved around the call.

Parameters for the VT-15 processor are set by

```
CALL PRAMTR (OPTION, VALUE, DFILE, IEDIT)
```

In the listings of Appendix C, mneumonics are always used for the various options, so the same mneumonics will be used here. Also, only those options actually used will be discussed.

The light pen can be turned on and off by

```
CALL PRAMTR (LIGHTPEN, ON)
```

and

```
CALL PRAMTR (LIGHTPEN, OFF)
```

Vectors and text about to be drawn can be made to blink with

```
CALL PRAMTR (BLINK, ON)
```

Text can be written vertically with

```
CALL PRAMTR (ROTATE, ON)
```

Of course, both BLINK and ROTATE can also be turned off. The menus in the prototype system were written by using

```
CALL PRAMTR (OFFSET, ON)
```

which tells the VT-15 processor to use the extreme right
The Graphics System

side of the screen instead of the main area. A value can be associated with the image about to be drawn by using

\[ \text{CALL PRAMTR (NAMEREG, VALUE)} \]

A program can be made to wait for the user to select part of the screen image with the light pen by

\[ \text{CALL LTORPB (IX, IY, NAMER, BTNS, IWHICH, WAIT).} \]

The important argument is NAMER. When LTORPB returns, NAMER is set to the value of NAMEREG that was set when the item selected by the user was drawn. Thus the programmer can create light pen buttons by using subroutine PRAMTR to enable the light pen and assign values for NAMEREG and then use subroutine LTORPB to see which light pen button has been selected.
APPENDIX C

COMPUTER PROGRAM LISTINGS

The programs in this appendix were written for the prototype system described in Chapter IV. The language used is RATFOR which is described in Appendix A. Subroutines not included in the listings are generally part of the graphics system described in Appendix B. Files specified in an INCLUDE statement are listed after their first mention.
# PROGRAM GRASP - # GRAPHICS ASSISTED SYNTHESIS OF PROCESSES

# INCLUDED FILES:
# GRDEF, DFDCI, MSDCI, LPDCI, UPDCI, DFCOM, MNCOM, MSCOM, LPCOM, UPCOM,
# PDCI, PRCOM, FDPCl, FWCOM

# CALLED SUBPROGRAMS:
# GRAPHICS PACKAGE, UPINI, MENU, MSG, DRAWF, OLNK1

INCLUDE GRDEF  # GRAPHICS MEANOUNICS
INCLUDE DFDCI  # MAIN DISPLAY FILE
INCLUDE MSDCI  # MESSAGE COMMON VARIABLES
INCLUDE LPDCI  # LIGHT PEN COMMON VARIABLES
INCLUDE UPDCI  # UNIT PICTURE VARIABLES
INCLUDE PDCI  # PROCESS COMMON VARIABLES
INCLUDE FDPCl  # FLOWSHEET DISPLAY FILE

INTEGER ICMD  # USER COMMAND

MACRO(NITEMS,11)
REAL ITEMS(3, NITEMS)  # MENU SELECTIONS
REAL HELLO(10)  # BEGINNING MESSAGE

INCLUDE DFDCI  # MAIN DISPLAY FILE
INCLUDE MNCOM  # MENU DISPLAY FILE
INCLUDE MSCOM  # MESSAGE DISPLAY FILE
INCLUDE LPCOM  # LIGHT PEN BUTTON DISPLAY FILES
INCLUDE UPCOM  # UNIT PICTURE DISPLAY FILES (FORCE INTO ROOT SEGMENT)
INCLUDE PRCOM  # PROCESS TOPOLOGY (FORCE INTO ROOT SEGMENT)
INCLUDE FWCOM  # FLOWSHEET DISPLAY FILE

DATA ITEMS/ 'SAVE', 'FLOWS', 'HEET',
'RSTR', 'FLOWS', 'HEET',
'ADD U', 'NIT', '
'DELET', 'E UNI', 'T',
'DELET', 'T REA', '
'DELET', 'STR', '
'DRAW', 'FLOWS', 'HEET',
'DRAW', 'GRAPH', ',
'ADD T', 'ITE', ',
'MAKE', 'COPY', ',
'EXIT', ',
DATA HELLO/ 'GRASP', 'S A V E ', 'F L O W S ', 'S Y N T ',
'S T R E A M ', 'H E E T ',
'P U T ', 'D E L E T ', 'D R A W ',
'M A K E ', 'C O P Y ', 'E X I T ',

# GET DISPLAY GOING
DFILE(1) = 0  # INITIALIZE MAIN FILE
MENDSP(1) = 0  # INITIALIZE MENU FILE
MSGDSP(1) = 0  # INITIALIZE MESSAGE FILE
SBNDSPI(1) = 0  # INITIALIZE STREAM BUTTON DISPLAY FILE
LPBDSPI(1) = 0  # INITIALIZE LIGHT PEN BUTTON DISP, FILE
FWDSPI(1) = 0  # INITIALIZE FLOWSHEET DISPLAY FILE
CALL PRAMTR (OFFSET, ON, MENDSP)  # PUT MENU IN OFFSET AREA
CALL COPY (0, MENDSP, DFILE)  # ADD TO DISPLAY
CALL PRAMTR (OFFSET, OFF, MSGDSP)  # PUT MESSAGE IN MAIN AREA
CALL COPY (0, MSGDSP, DFILE)  # ADD TO DISPLAY
CALL PRAMTR (OFFSET, OFF, LPBDSPI)  # PUT LP BUTTONS IN MAIN AREA
CALL COPY (0, LPBDSPI, DFILE)  # ADD TO DISPLAY
CALL PRAMTR (OFFSET, OFF, FWDSPI)  # PUT FLOWSHEET IN MAIN AREA
CALL COPY (0, FWDSPI, DFILE)
CALL QINIT (DFILE)  # START THE DISPLAY

# INITIALIZE STREAM LIGHT PEN BUTTON
CALL UPINI  # INITIALIZE THE UNIT PICTURE DATA

# INITIALIZE PROCESS DATA STRUCTURES
NEXTH = 1
FX = 0
FY = 0
DO I = 1, MAXUNIT
   DO J = 1, 6
      DO K = 1, 2
         PUNIT(I, J, K) = 0
      END DO
   END DO
END DO

CALL MSG(HELLO)  # ONE TIME MESSAGE TO IDENTIFY SYSTEM

REPEAT
   ICMD = MENU (NITEMS, ITEMS)
   IF (ICMD == 7)
      CALL DRAWF (0)  # DRAW FLOWSHEET
   ELSE IF (ICMD == 8)
      CALL DRAWF (1)  # DRAW GRAPH
   ELSE IF (ICMD == 10)
      CALL VTSIM (DFILE, 1)  # MAKE COPY (1)
   ELSE IF (ICMD == 11)
      CALL EXIT  # EXIT
   ELSE  # OVERLAY LINK 1
      CALL OLNK1 (ICMD)
   END IF
END REPEAT
END
# FILE NAME: GRDEF RAT

# DEFINITIONS FOR USE WITH THE GRAPHICS-15 SUBROUTINE CALLS

MACRO(DEFINE, MACRO)
DEFINE(SCALE, 1)
DEFINE(INTENSITY, 2)
DEFINE(LIGHTPEN, 4)
DEFINE(BLINK, 8)
DEFINE(DASH, 16)
DEFINE(OFFSET, 32)
DEFINE(ROTATE, 64)
DEFINE(NAMEREG, 128)
DEFINE(ON, 1)
DEFINE(OFF, 0)
DEFINE(WAIT, 1)
DEFINE(WAITOFF, 0)
DEFINE(NEWCOPY, 0)
DEFINE(NEWLINE, 1)
DEFINE(NEWPRAMTR, 2)
DEFINE(NEWTEXT, 3)
DEFINE(NEWPOINT, 4)
DEFINE(NEWGRAPH, 5)
DEFINE(NEWANY, 6)

# FILE NAME: DFCOM RAT -
# MAIN DISPLAY FILE FOR GRASP

INTEGER DFILE(16)

# FILE NAME: DFCOM RAT -
# MAIN DISPLAY FILE

COMMON /DFCOM/ DFILE
FILE NAME MSOCL RAT
# VARIABLES FOR MSOCL
REAL MSTUFF(10)
INTEGER MSGDSP(9)

FILE NAME MSOCL RAT
# COMMON FOR MESSAGE AT BOTTOM OF DISPLAY
COMMON /MSCOM/ MSGDSP, MSTUFF

FILE NAME LPDCL
# DISPLAY FILES FOR LIGHT PEN BUTTONS
INTEGER SBNDSP(7) # TRIANGLE FOR STREAM BUTTONS
INTEGER LPBDSP(904) # DISPLAY FILE TO HOLD ALL LIGHT PEN BUTTONS

FILE NAME LPCOM
# LIGHT PEN BUTTONS
COMMON /LPCOM/ SBNDSP, LPBDSP
# FILE NAME UpDcl

# DECLARATIONS FOR UNIT PICTURE COMMON

INTEGER UNIT  # NUMBER OF CURRENT UNIT PICTURE
INTEGER h    # 1 FOR RIGHT HAND UNIT; -1 FOR LEFT HAND
MACRO(NUNITS, 7)  # NUMBER OF UNIT PICTURES
MACRO(NSTRMS, 4)  # MAXIMUM NUMBER OF STREAMS PER UNIT
MACRO(DFLSZ, 50)  # MAXIMUM DISPLAY FILE SIZE

MACRO(X, 1)
MACRO(Y, 2)
MACRO(STRDIR, 3)  # STREAM DIRECTION
MACRO(STREXT, 4)  # STREAM EXTENSION
MACRO(XDATA, 5)
MACRO(YDATA, 6)
MACRO(LUNIT, 1)   # LEFT HAND UNIT
MACRO(RUNIT, 2)   # RIGHT HAND UNIT

INTEGER USIZE(NUNITS, 2)  # SIZE (N, X) = X DIMENSION OF UNIT PICTURE N
                           # SIZE (N, Y) = Y DIMENSION OF UNIT PICTURE N
INTEGER UTITLE(NUNITS, 2)
                           # TITLE(NUNITS, X) = X COORDINATE OF START OF TITLE FOR UNIT N
                           # TITLE(NUNITS, Y) = Y COORDINATE OF START OF TITLE FOR UNIT N
INTEGER USTRM(NUNITS, NSTRMS, 6)
                           # USTRM(N, M, X) = X ADDRESS OF STREAM M, UNIT N
                           # USTRM(N, M, Y) = Y ADDRESS OF STREAM M, UNIT N
                           # USTRM(N, M, STRDIR) = DIRECTION OF STREAM:
                           # 0 FOR 0 DEGREES  # 1 FOR 0 DEGREES
                           # 2 FOR 90 DEGREES  # 2 FOR 90 DEGREES
                           # 3 FOR 180 DEGREES  # 3 FOR 180 DEGREES
                           # 4 FOR 270 DEGREES  # 4 FOR 270 DEGREES
                           # USTRM(N, M, STREXT) = NON-ZERO IF STREAM NEEDS EXTENSION -
                           # USED WHEN MORE THAN ONE STREAM PER UNIT
                           # HAS THE SAME DIRECTION CODE
                           # USTRM(N, M, XDATA) = X ADDRESS OF STREAMS DATA FIELD
                           # USTRM(N, M, YDATA) = Y ADDRESS OF STREAMS DATA FIELD

INTEGER UDFILE(DFLSZ, 2, NUNITS)
                           # UDFILE(1, LUNIT, N) = START OF DISPLAY FILE FOR LEFT HAND UNIT N
                           # UDFILE(1, RUNIT, N) = START OF DISPLAY FILE FOR RIGHT HAND UNIT N

# FILE NAME UpCom

# UNIT PICTURE COMMON
COMMON /UpCom/ USIZE, UTITLE, USTRM, UDFILE
DECLARATIONS FOR PROCESS FLOWSHEET DATA

MACRO(\(\text{MAX-UNIT,15}\))
INTEGER NEXTU # NEXT AVAILABLE UNIT SPACE
INTEGER FX, FY # SIZE OF ENTIRE FLOWSHEET
REAL PRNAME(2, MAX-UNIT) # NAME OF EACH UNIT
INTEGER PRUNIT(MAX-UNIT, 6, 2)
MACRO(STRM1,1)
MACRO(STRM2,2)
MACRO(STRM3,3)
MACRO(STRM4,4)
MACRO(NUNIT,1)
MACRO(NSTRM,2)
MACRO(Coord,5)
MACRO(XLOC,1)
MACRO(YLOC,2)
MACRO(PDATA,6)
MACRO(SYM,1)
MACRO(HAND,2)

# THUS, FOR EACH PROCESS UNIT THERE ARE THE FOLLOWING ENTRIES:

# UNIT N
# STRM1 NUNIT NSTRM
# STRM2 NUNIT NSTRM
# STRM3 NUNIT NSTRM
# STRM4 NUNIT NSTRM
# COORD XLOC YLOC
# DATA SYMB HAND
# PRNAME(2)

# COMMON FOR PROCESS FLOWSHEET DATA
COMMON /PRCOM/ PRUNIT, PRNAME, NEXTU, FX, FY
# FILE NAME FWCLI
# FLOWSHEET DISPLAY FILE

INTEGER FWDSP(2000)

# FILE NAME FWCOMI
# FLOWSHEET DISPLAY FILE COMMON

COMMON /FWCOMI/ FWDSP

# FILE NAME MNCOM RAT
# MENU DISPLAY FILE
COMMON /MNCOM/ MENDSP(190)
SUBROUTINE UPINI

# INITIALIZE THE UNIT PICTURE DATA STRUCTURES

#include UPDCL
#include UPCOM

DO I = 1, NUNITS
  $!
  DO J = 1, 2
    $(
      USIZE(I, J) = 0
      UITLE(I, J) = 0
    )$
  DO J = 1, NSTRMS
    DO K = 1, 6
      USTRM(I, J, K) = 0
    $)
  CALL IPMP (1) # PUMP
  CALL ICLMN (2) # COLUMN
  CALL IVLV (3) # VALVE
  CALL IDRM (4) # DRUM
  CALL IXHR (5) # HEAT EXCHANGER
  CALL IMXR (6) # MIXER
  CALL ICMP (7) # COMPRESSOR

# TO ADD A NEW UNIT PICTURE:
# 1. ADD 1 TO ABOVE CALL'S ARGUMENT
# 2. WRITE A NEW SUBROUTINE USING ABOVE AS MODELS TO
# 3. INITIALIZE THE PICTURE'S DATA STRUCTURES
# 3. INSPECT UPDCL AND CHANGE THE MACRO'S FOR NUNITS, NSTRMS
#      AND DFLSZ IF NECESSARY

RETURN
END
SUBROUTINE IPMP (UNIT)

# INITIALIZE UNIT PICTURE DATA STRUCTURES FOR THE PUMP

MACRO(NVECS, 15)
INTEGER IVECS(3, NVECS)

INCLUDE UPDCL
INCLUDE UPCOM

DATA IVECS / 28, 42, 0, -14, -14, 1, 56, 0, 1, -14, 14, 1, 119, 35, 0, -140, 0, 1, -14, -14, 1, 0, -14, 1, 14, -14, 1, 14, 0, 1, 14, 14, 1, 0, 14, 1, -14, 14, 1, -49, -21, 0, 42, 0, 1/

USIZE(UNIT, X) = 175
USIZE(UNIT, Y) = 105

UTITLE(UNIT, X) = 14
UTITLE(UNIT, Y) = 7

USTRM(UNIT, 1, X) = 175
USTRM(UNIT, 1, Y) = 77
USTRM(UNIT, 1, STRDIR) = 1
USTRM(UNIT, 1, STREXT) = 0
USTRM(UNIT, 1, XDATA) = 35
USTRM(UNIT, 1, YDATA) = 84

USTRM(UNIT, 2, X) = 0
USTRM(UNIT, 2, Y) = 56
USTRM(UNIT, 2, STRDIR) = 3
USTRM(UNIT, 2, STREXT) = 0

CALL IDRAW (NVECS, IVECS, UNIT)

RETURN

END
SUBROUTINE ICMN (UNIT)

# INITIALIZE UNIT PICTURE DATA STRUCTURES FOR THE COLUMN

MACRO(NVECS,17)
INTEGER IVECS(3, NVECS)

INCLUDE UPDCL
INCLUDE UPCOM

DATA IVECS / 28, 7, 0,
14, 0, 1,
14, 14, 1,
0, 140, 1,
-14, 14, 1,
-14, 0, 1,
-14, -14, 1,
0, -140, 1,
14, -14, 1,
-28, 84, 0,
14, 0, 1,
42, 63, 0,
14, 0, 1,
0, 28, 1,
0, -182, 0,
0, 28, 1,
-14, 0, 1/

USIZE(UNIT, X) = 224
USIZE(UNIT, Y) = 182
UTITLE(UNIT, X) = 70
UTITLE(UNIT, Y) = 91

USTRM(UNIT, 1, X) = 70
USTRM(UNIT, 1, Y) = 182
USTRM(UNIT, 1, STRDIR) = 2
USTRM(UNIT, 1, STREXT) = 0
USTRM(UNIT, 1, XDATA) = 84
USTRM(UNIT, 1, YDATA) = 147

USTRM(UNIT, 2, X) = 0
USTRM(UNIT, 2, Y) = 91
USTRM(UNIT, 2, STRDIR) = 3
USTRM(UNIT, 2, STREXT) = 0

USTRM(UNIT, 3, X) = 70
USTRM(UNIT, 3, Y) = 0
USTRM(UNIT, 3, STRDIR) = 4
USTRM(UNIT, 3, STREXT) = 0
USTRM(UNIT, 3, XDATA) = 84
USTRM(UNIT, 3, YDATA) = 14

CALL IDRAW (NVECS, IVECS, UNIT)

RETURN
END
SUBROUTINE IVLV (UNIT)

# INITIALIZE UNIT PICTURE DATA STRUCTURES FOR THE VALVE

MACRO(NVECS,16)
INTEGER IVECS(3, NVECS)

INCLUDE UPOCL
INCLUDE UPDCOM

DATA IVECS / 0, 35, 0,
                   14, 0, 1,
                   0, 7, 0,
                   28, -14, 1,
                   0, 14, 1,
                   -28, -14, 1,
                   0, 14, 1,
                   7, 7, 1,
                   14, 0, 1,
                   7, -7, 1,
                   -28, 0, 1,
                   14, 0, 0,
                   0, -14, 1,
                   14, 0, 0,
                   154, 0, 1/

USIZE(UNIT, X) = 196
USIZE(UNIT, Y) = 63

UTITLE(UNIT, X) = 14
UTITLE(UNIT, Y) = 7

USTRM(UNIT, 1, X) = 196
USTRM(UNIT, 1, Y) = 35
USTRM(UNIT, 1, STRDIR) = 1
USTRM(UNIT, 1, STREXT) = 0
USTRM(UNIT, 1, XDATA) = 56
USTRM(UNIT, 1, YDATA) = 42

USTRM(UNIT, 2, X) = 0
USTRM(UNIT, 2, Y) = 35
USTRM(UNIT, 2, STRDIR) = 3
USTRM(UNIT, 2, STREXT) = 0

CALL IDRAW(NVECS, IVECS, UNIT)

RETURN
END
SUBROUTINE IDRM (UNIT)

# INITIALIZE THE UNIT PICTURE DATA STRUCTURES FOR THE DRUM

MACRO(NVECS, L8)
INTEGER IVECS(3, NVECS)

INCLUDE UPDCL
INCLUDE UPCOM

DATA IVECS / 14, 42, 0, 0, 70, 1, 14, 14, 1, 28, 0, 1, 14, -14, 1, 0, -70, 1, -14, -14, 1, -28, 0, 1, -14, 14, 1, 56, 0, 1, 0, 70, 0, -56, 0, 1, 0, -35, 0, -14, 0, 1, 42, 77, 0, 0, -26, 1, 0, -98, 0, 0, -28, 1/

USIZE(UNIT, X) = 224
USIZE(UNIT, Y) = 154
UTITLE(UNIT, X) = 84
UTITLE(UNIT, Y) = 70

USTRM(UNIT, 1, X) = 42
USTRM(UNIT, 1, Y) = 154
USTRM(UNIT, 1, STRDIR) = 2
USTRM(UNIT, 1, STREXT) = 0
USTRM(UNIT, 1, XDATA) = 56
USTRM(UNIT, 1, YDATA) = 133

USTRM(UNIT, 2, X) = 0
USTRM(UNIT, 2, Y) = 77
USTRM(UNIT, 2, STRDIR) = 3
USTRM(UNIT, 2, STREXT) = 0

USTRM(UNIT, 3, X) = 42
USTRM(UNIT, 3, Y) = 0
USTRM(UNIT, 3, STRDIR) = 4
USTRM(UNIT, 3, STREXT) = 0
USTRM(UNIT, 3, XDATA) = 56
USTRM(UNIT, 3, YDATA) = 7

CALL IDRAW (NVECS, IVECS, UNIT)

RETURN
END
SUBROUTINE IXNCR (UNIT)

# INITIALIZE THE UNIT PICTURE DATA STRUCTURES FOR THE HEAT EXCHANGER

MACRO(NVECS,17)
INTEGER IVECS(3,NVECS)

INCLUDE UPDCL
INCLUDE UPCOM

DATA IVECS / 14, 28, 0,
0, 14, 1,
14, 14, 1,
14, 0, 1,
14, -14, 1,
0, -14, 1,
-14, -14, 1,
-14, 0, 1,
-14, 14, 1,
14, -28, 0,
0, 42, 1,
14, -14, 1,
0, 63, 1,
-42, -56, 0,
14, 0, 1,
42, 0, 0,
154, 0, 1/

USIZE(UNIT, X) = 210
USIZE(UNIT, Y) = 91
TITLE(UNIT, X) = 63
TITLE(UNIT, Y) = 7

USTRM(UNIT, 1, X) = 210
USTRM(UNIT, 1, Y) = 35
USTRM(UNIT, 1, XDATA) = 70
USTRM(UNIT, 1, YDATA) = 42

USTRM(UNIT, 2, X) = 42
USTRM(UNIT, 2, Y) = 91
USTRM(UNIT, 2, XDATA) = 56
USTRM(UNIT, 2, YDATA) = 70

USTRM(UNIT, 3, X) = 3
USTRM(UNIT, 3, Y) = 35
USTRM(UNIT, 3, XDATA) = 3
USTRM(UNIT, 3, YDATA) = 0

USTRM(UNIT, 4, X) = 28
USTRM(UNIT, 4, Y) = 0
USTRM(UNIT, 4, XDATA) = 4
USTRM(UNIT, 4, YDATA) = 0

CALL IDRAW (NVECS, IVECS, UNIT)

RETURN
END
SUBROUTINE IMXR (UNIT)

# INITIALIZE THE UNIT PICTURE DATA STRUCTURES FOR THE MIXER

*MACRO(NVECS,4)
INTEGER IVECS(3, NVECS)
INCLUDE UPDCL
INCLUDE UPCOM

DATA IVECS / 14, 0, 0, 
0, 56, 1, 
0, -28, 0, 
154, 0, 1/

USIZE(UNIT, X) = 168
USIZE(UNIT, Y) = 56

UTITLE(UNIT, X) = 28
UTITLE(UNIT, Y) = 7

USTRM(UNIT, 1, X) = 168
USTRM(UNIT, 1, Y) = 28
USTRM(UNIT, 1, STRDIR) = 1
USTRM(UNIT, 1, STREXT) = 0
USTRM(UNIT, 1, XDATA) = 28
USTRM(UNIT, 1, YDATA) = 35

USTRM(UNIT, 2, X) = 14
USTRM(UNIT, 2, Y) = 56
USTRM(UNIT, 2, STRDIR) = 2
USTRM(UNIT, 2, STREXT) = 0

USTRM(UNIT, 3, X) = 14
USTRM(UNIT, 3, Y) = 0
USTRM(UNIT, 3, STRDIR) = 4
USTRM(UNIT, 3, STREXT) = 0

CALL IDRAW (NVECS, IVECS, UNIT)

RETURN
END
SUBROUTINE ICMP (UNIT)

# INITIALIZE THE UNIT PICTURE DATA STRUCTURES FOR THE COMPRESSOR

MACRO(NVECS, 8)
INTEGER IVECS(3, NVECS)

INCLUDE UPDCL
INCLUDE UPCOM

DATA IVECS / 14, 7, 0,
0, 42, 1,
42, -21, 1,
-42, -21, 1,
0, 21, 0,
-14, 0, 1,
56, 0, 0,
147, 0, 1/

USIZF(UNIT, X) = 203
USIZF(UNIT, Y) = 56

UTITLE(UNIT, X) = 63
UTITLE(UNIT, Y) = 7

USTRM(UNIT, 1, X) = 203
USTRM(UNIT, 1, Y) = 28
USTRM(UNIT, 1, STRDIR) = 1
USTRM(UNIT, 1, STREXT) = 0
USTRM(UNIT, 1, XDATA) = 63
USTRM(UNIT, 1, YDATA) = 35

USTRM(UNIT, 2, X) = 0
USTRM(UNIT, 2, Y) = 28
USTRM(UNIT, 2, STRDIR) = 3
USTRM(UNIT, 2, STREXT) = 0

CALL IDRAW (NVECS, IVECS, UNIT)

RETURN
END
SUBROUTINE IDRAW (NVECS, IVECS, UNIT)

# CREATE THE DISPLAY FILES IN UPCOM FOR UNIT FROM IVECS.

INTEGER IVECS(3, NVECS)
INCLUDE UPDCL
INCLUDE UPCOM

DO I = 1, 2
$(
  IF (I == RUNIT)
    # DO THE RIGHT HAND DISPLAY FILE
    H = 1
  ELSE
    # DO THE LEFT HAND DISPLAY FILE
    H = -1
  UDFILE(1, I, UNIT) = 0
  IF (I == LUNIT)
    # DRAW FROM RIGHT HAND CORNER TO LEFT
    CALL LINE (USIZE(UNIT, X), 0, 0, UDFILE(1, I, UNIT))
  DO J = 1, NVECS
    CALL LINE (IVECS(1, J) * H, IVECS(2, J), IVECS(3, J),
    UDFILE(1, I, UNIT))
$(
RETURN
END
SUBROUTINE MSG (STUFF)
# DISPLAY A MESSAGE AT THE BOTTOM OF THE SCREEN

INCLUDE GRDEF            # GRAPHICS MNEUMONICS
INCLUDE MSGCL            # MESSAGE COMMON
REAL STUFF(I0)           # 50 CHARACTERS WORTH
INCLUDE MSGM

MSGDSP(I) = 0            # START OVER
CALL PRA MTR (OFFSET, OFF, MSGDSP)
CALL POINT (140, 0)

DO I = 1, 10
   MSTUFF(I) = STUFF(I)  # COPY TO COMMON
END DO

CALL TEXT (MSTUFF, 49)
RETURN
END
INTEGER FUNCTION MENU (NUMB, ITEMS)

# FUNCTION TO DISPLAY A MENU OF ITEMS ON THE OFFSET AREA OF THE SCOPE
# AND RETURN THE NUMBER OF THE ITEM SELECTED BY LIGHT PEN.

# NUMB ITEMS WILL BE DISPLAYED FROM THE ARRAY ITEMS (3, NUMB) WHERE THE
# FIRST INDEX REFERS TO THE THREE WORDS OF A TEXT STRING OF 15 CHARACTERS,
# THE FIRST 14 WILL BE DISPLAYED AND THE LAST CONVERTED TO AN "ALT MODE",
# NUMB MUST BE >= 1 AND <= 25.

# MENU WILL WAIT FOR A LIGHT PEN HIT ON ONE OF THE ITEMS. THE ITEM WILL THEN
# BLINK UNTIL THE USER OK'S IT WITH PUSH BUTTON 6 OR CLEARS IT WITH PUSH BUTTON
# 5. RETURN WILL BE WITH MENU = NUMBER OF THE ITEM SELECTED.

#include GRDEF

REAL ITEMS (3, NUMB) # THE MENU ITEMS
INTEGER NUMB  # NUMBER OF ITEMS TO BE DISPLAYED
INTEGER IEDT (25)  # EDIT ADDRESSES FOR THE ITEMS
INTEGER LPEDT  # EDIT ADDRESS FOR LIGHT PEN ENABLE
INTEGER NAME  # EDIT ADDRESS FOR NAME STRING
INTEGER LNAME  # CONTENTS OF NAME STRING AFTER LP HIT
LOGICAL BTN51 (6), BTN52 (6)  # CONTENTS OF NAME STRING AFTER LP HIT
INTEGER INTHICH  # KIND OF HIT (LP OR PB) FROM LTORPB
REAL SELECT (3)  # 'SELECT ITEM'
REAL CLRIT (3)  # 'PB5: CLEAR'
REAL OKIT (3)  # 'PB6: ENTER'
INTEGER MSGLP (8)  # DISPLAY FILE FOR LP INSTRUCTIONS
INTEGER MSGPR (13)  # DISPLAY FILE FOR PB INSTRUCTIONS

#include MNCOM

DATA SELECT /'SELECT', 'ITEM', 'M' /
DATA CLRIT /'PB5', 'CLEAR', 'R' /
DATA OKIT /'PB6', 'ENTER', 'R' /

IF ((NUMB < 1) + (NUMB > 25))
  $  
  WRITE (25, 1)
  1 FORMAT ('*** MENU: NUMBER OF ITEMS OUT OF RANGE')
  CALL EXIT
  $

# INITIALIZE INSTRUCTION FILES

MSGLP (1) = 0
CALL POINT (0, 0, OFF, MSGLP)
CALL TEXT (SELECT, 14)
MSGPR (1) = 0
CALL POINT (0, 0, OFF, MSGPR)
CALL TEXT (CLRIT, 14)
CALL POINT (0, 36)
CALL TEXT (OKIT, 14)

# NOW BUILD THE MENU

MENDSP (1) = 0
CALL PRAMTR (LIGHTPEN, OFFSET, ON, ON, MENDSP, LPEDT)  # (MENDSP IN COMMON)

# DISPLAY MENU

DO I = 1, NUMB
  $
  CALL POINT (0, 1023-3601)
  CALL PRAMTR (BLINK, NAME, OFF, I, MENDSP, IEDT(I))
  CALL TEXT (ITEMS(1, I), 14)
  $

$
ADD INSTRUCTIONS
CALL PRAMTR (BLINK, OFF)
CALL COPY (0, MSGLP)
CALL COPY (0, MSGPB)
CALL PRAMTR (OFFSET, OFF)

SELECT ITEM WITH LIGHT PEN AND OK WITH PUSH BUTTONS
REPEAT

$(
CALL BLANK (MSGPB)  # SUPPRESS PB MESSAGE
CALL UNBLNK (MSGLP)  # TURN ON LP MESSAGE
CALL GETPSH (BTNS2)
REPEAT

$(
CALL LTORPB (X, Y, NAMER, BTNS1, IWHICH, WAIT)
IF (BTNS1(1) \= BTNS2(1))
%
CALL REPLOT (NEWPRAMTR, LIGHTPEN+OFFSET, OFF, ON, LPEDT)
CALL HCOPY
CALL REPLOT (NEWPRAMTR, LIGHTPEN+OFFSET, ON, ON, LPEDT)
BTNS2(1) = BTNS1(1)
%)
$
UNTIL ((IWICH \= 1) \& (NAMER \geq 1) \& (NAMER \leq NUMB))

# BLINK ITEM AND TURN LIGHT PEN OFF
CALL REPLOT (NEWPRAMTR, BLINK+NAMEREG, ON, NAMER, IEDIT(NAMER))
CALL REPLOT (NEWPRAMTR, LIGHTPEN+OFFSET, OFF, ON, LPEDT)
CALL BLANK (MSGLP)  # SUPPRESS LP MESSAGE
CALL UNBLNK (MSGPB)  # TURN ON PB MESSAGE
REPEAT

$(
CALL GETPSH (BTNS1)
CALL LTORPB (X, Y, I, BTNS2, IWHICH, WAIT)
$
UNTIL ((BTNS1(5) \leq BTNS2(5)) \& (BTNS1(6) \leq BTNS2(6)))

IF (BTNS1(5) \leq BTNS2(5))
$
CALL REPLOT (NEWPRAMTR, BLINK+NAMEREG, OFF, NAMER, IEDIT(NAMER))
CALL REPLOT (NEWPRAMTR, LIGHTPEN+OFFSET, ON, ON, LPEDT)
$
$
UNTIL (BTNS1(6) \leq BTNS2(6))
# RETURN ITEM NUMBER
MENU = NAMER
RETURN
END
SUBROUTINE HCOPY

# PRODUCE HARD COPY OF ENTIRE SCREEN

INCLUDE DFDCCL       # DISPLAY FILE
INCLUDE DFCOM

CALL VTSIM (DFILE, 0)
RETURN
END
SUBROUTINE OLNI1 (ICMD)

# HANDLE ALL COMMANDS FROM GRASP THAT WILL REQUIRE OVERLAY LINK 1
# THIS SUBROUTINE IS JUST A DEVICE TO ALLOW ITS CALLED ROUTINES
# TO COME FROM A LIBRARY. ON ANOTHER MACHINE, THE FOLLOWING
# ENTRIES COULD BE INSERTED INTO THE "ELSE IF" CHAIN IN GRASP.

INTEGER ICMD

IF (ICMD == 1)
    CALL SAVEF # SAVE FLOWSHEET
ELSE IF (ICMD == 2)
    CALL RSTRF # RESTORE FLOWSHEET
ELSE IF (ICMD == 3)
    CALL ADDU # ADD UNIT
ELSE IF (ICMD == 4)
    CALL DELU # DELETE UNIT
ELSE IF (ICMD == 5)
    CALL ADDS # ADD STREAM
ELSE IF (ICMD == 6)
    CALL DELS # DELETE STREAM
ELSE IF (ICMD == 9)
    CALL ADDT # ADD TITLE

# ELSE BAD COMMAND, TRY AGAIN
RETURN
FND
SUBROUTINE SAVEF

' SAVE CONTENTS OF PRCOM IN DISK FILE

INCLUDE LUNS       # LOGICAL UNIT NUMBERS FOR GRASP
INCLUDE PRDCL      # VARIABLES FOR PRCOM
REAL FNAME         # FILE NAME (FROM USER)
INCLUDE PRCOM      # PROCESS COMMON

WRITE (USER, 1)
   1 FORMAT (' SAVEF - ENTER FILE NAME: UP TO 5 LETTERS.')
READ (USER, 5) FNAME
   5 FORMAT (A5)

CALL ENTER (DISK, FNAME, 'PRO', IEV)
CALL WAITFR (IEV)
IF (IEV <= 0)
   $(
       WRITE (USER, 2) IEV, FNAME
       2 FORMAT (' *** SAVEF - ENTER ERROR', 14, ' ON ', A5, ' PRO', /,
                  ' FLOWSHEET NOT SAVED.')
       RETURN
   )
I = MAX*UNIT
WRITE (DISK, 100, ERR=10) I, NEXTU
   100 FORMAT (IX, 15, IX, 15)
DO I = 1, MAX*UNIT
   WRITE (DISK, 101, ERR=10) PRNAME(1, I), PRNAME(2, I),
                           ((PRUNIT(I, J; K), K = 1, 2), J = 1, 6)
   101 FORMAT (IX, A5, A4, 12(1X, 15))
CALL CLOSE (DISK)
WRITE (USER, 3) FNAME
   3 FORMAT (' SAVEF - FLOWSHEET SAVED IN ', A5, ' PRO')
RETURN

# WRITE ERROR
10 WRITE (USER, 4)
   4 FORMAT (' *** SAVEF - WRITE ERROR: FLOWSHEET NOT SAVED.')
CALL CLOSE (DISK)
RETURN
END
# FILE NAME: LUNS RAT

# MACROS FOR LOGICAL UNIT NUMBER ASSIGNMENTS
MACRO(USER,25) # NORMALLY TT1
MACRO(DISK,17) # COULD CHANGE TO A DEC TAPE NUMBER
MACRO(LSTLUN,16) # NORMALLY LINE PRINTER
SUBROUTINE RSTRF

# RESTORE CONTENTS OF PRCOM FROM FLOWSHEET

#include LUNS  # LOGICAL UNIT NUMBERS FOR GRASP
#include PRCCL  # VARIABLES FOR PRCOM
REAL FNAME  # FILE NAME (FROM USER)
#include PRCOM  # PROCESS COMMON

WRITE (USER, 1)
   1 FORMAT (' RSTRF - ENTER FILE NAME: UP TO 5 LETTERS."
READ (USER, 5) FNAME
   5 FORMAT (A5)

CALL SEEK (DISK, FNAME, 'PRO', IEV)
CALL WAITFR (IEV)
IF (IEV <= 0) THEN
   WRITE (USER, 2) IEV, FNAME
      2 FORMAT ('*** RSTRF - SEEK ERROR', I4, ' ON ', A5, ' PRO', /
               ' FLOWSHEET NOT RESTORED.'
   RETURN
endif

READ (DISK, 100, ERR=10) I, NEXTU
100 FORMAT (1X, I5, 1X, I5)
DO 1 = 1, MAX*UNIT
   READ (DISK, 101, ERR=10) PRNAME(1, I), PRNAME(2, I), /
      (PRUNIT(I, J, K), K = 1, 2), J = 1, 6
101 FORMAT (1X, A5, A4, 12(1X, I5))
CALL CLOSE (DISK)
WRITE (USER, 3) FNAME
   3 FORMAT (' RSTRF - FLOWSHEET RESTORED FROM ', A5, ' PRO')
RETURN

# READ ERROR
10 WRITE (USER, 4)
   4 FORMAT ('*** RSTRF - READ ERROR: FLOWSHEET NOT RESTORED.')
CALL CLOSE (DISK)
RETURN
END
SUBROUTINE ADDU

# ADD A PROCESS UNIT TO PRUNIT

INCLUDE LUNS  # LOGICAL UNIT NUMBERS FOR GRASP
INCLUDE GROEF  # GRAPHICS MNEUMONICS
INCLUDE PROCL  # PROCESS COMMON
INCLUDE UPDCL  # UNIT PICTURE COMMON
INCLUDE DFCL  # DISPLAY COMMON
INCLUDE FNDCL  # FLOWSHEET DISPLAY COMMON

MACRO(NSYMBS,7)
REAL SYMBS(3, NSYMBS)
INTEGER IX, IY  # COORDINATES FOR SYMBOL

INCLUDE PRCOM
INCLUDE UPCOM
INCLUDE DFCOM
INCLUDE FCOM

DATA SYMBS /
'PUMP', 'COLUM', 'N', 'VALVE', 'FLASH', 'DRUM', 'HEAT', 'EXCHA', 'EXCHER', 'MIXER', 'COMPR', 'COMPRESOR', /

IF (NEXTU == -1)
%%
WRITE (USER, 1)
1 FORMAT ('*** ADDU - TOO MANY UNITS: UNIT NOT ADDED,' )
RETURN
%

IX = 100
IY = FY + 33
UNIT = MENU (NSYMBS, SYMBS)
CALL POINT (IX, IY, 0, FNDSP)
CALL COPY (0, UDFILE(1, RUNIT, UNIT))
FY = FY * 10 + USIZE(UNIT, Y)
PRUNIT(NEXTU, PDATA, SYMQR) = UNIT
PRUNIT(NEXTU, PDATA, HAND) = UNIT
PRUNIT(NEXTU, COORD, XLOC) = IX
PRUNIT(NEXTU, COORD, YLOC) = IY
WRITE (USER, 2)
2 FORMAT ('ADDU - ENTER UNIT NAME: UP TO 9 LETTERS,' )
READ (USER, 3) (PRNAME(I, NEXTU), I = 1, 2)
3 FORMAT (2A5)
IX = IX + UTITLE(UNIT, X)
IY = IY + UTITLE(UNIT, Y)
CALL POINT (IX, IY)
CALL TEXT (PRNAME(1, NEXTU), 9)
REPEAT
%
IF (NEXTU == MAX-UNIT)
%
NEXTU = -1
RETURN
%
NEXTU = NEXTU + 1
%
UNITL (PRUNIT(NEXTU, PDATA, SYMQR) == 0)
RETURN
END
SUBROUTINE DELU

# DELETE A UNIT FROM THE FLOWSHEET

# INCLUDED FILES:
# GRDEF, PROCL, LPDCL, UPDCL, PRCOM, LPCOM, UPCOM

# CALLED SUBPROGRAMS:
# GRAPHICS PACKAGE, ISLCT

INCLUDE GRDEF              # GRAPHICS MNEUMONICS
INCLUDE PROCL              # PROCESS TOPOLOGY
INCLUDE LPDCL              # LIGHT PEN DISPLAY FILE
INCLUDE UPDCL              # UNIT PICTURE VARIABLES

INTEGER LPEDT              # EDIT ADDRESS FOR TURNING LIGHT PEN ON AND OFF
INTEGER IEDT(128)          # EDIT ADDRESSES FOR IDENTIFYING THE BUTTONS
INTEGER IX, IY             # FOR CALCULATING COORDINATES
INTEGER IUNIT              # UNIT BEING WORKED ON
INTEGER ISTRM              # STREAM BEING WORKED ON
INTEGER JUNIT              # USED FOR CONNECTED UNIT
INTEGER JSTRM              # USED FOR CONNECTED STREAM
REAL INCOR(10)             # MESSAGE THAT FLOWSHEET IS NOT UP-TO-DATE
LOGICAL NONE               # TRUE IF NO LIGHT PEN BUTTONS DRAWN

INCLUDE PRCOM              # PROCESS COMMON
INCLUDE LPCOM              # LIGHT PEN COMMON
INCLUDE UPCOM              # UNIT PICTURE COMMON

DATA INCOR '/GET', 'FLOWS', 'HEET', 'IS NO', 'T UP-', 'TO-DA',
'te', 4 o'/

# DRAW LIGHT PEN BUTTONS:
# A LINE UNDER THE NAME OF EVERY UNIT THAT EXISTS.
LPBSDP(1) = 0
CALL PRACTM (LIGHTPEN + OFFSET, ON, OFF, LPBSDP, LPEDT)
NONE = .TRUE.
DO I = 1, MAX-UNIT
  IF (PRUNIT(I, PDATA, SYMB) \= 0)
    IF (PRUNIT(I, PDATA, SYMB) \= 0)
      CALL PRACTM (BLINK+NAMEREG, OFF, I, LPBSDP, IEDT(I))
      UNIT = PRUNIT(I, PDATA, SYMB)
      H = PRUNIT(I, PDATA, HAND)
      IX = PRUNIT(I, COORD, XLOC)
      IY = PRUNIT(I, COORD, YLOC)
      IY = IY + UTITLE(UNIT, Y)
      IF (H == LUNIT)
        IX = IX * USIZE(UNIT, X) - UTITLE(UNIT, X) - 126
      ELSE
        IX = IX + UTITLE(UNIT, X)
      CALL POINT (IX, IY)
      CALL LINE (126, 0)
      NONE = .FALSE.
    IF (NONE)
      LPBSDP(1) = 0
      CALL PRACTM (OFFSET, OFF)
      RETURN
$)
CALL PRACTM (LIGHTPEN + BLINK, OFF, OFF)
IUNIT = ISLCT (LPEDT, IEDT)  # HAVE USER SELECT UNIT

# CLEAR ITS ENTRY
DO 1STRM = 1, 4

$# CLEAR CONNECTIONS TO OTHER UNITS
1F (PRUNIT(IUNIT, ISTRM, NUNIT) /= 0)

$JUNIT = PRUNIT(IUNIT, ISTRM, NUNIT)
JSTRM = PRUNIT(IUNIT, ISTRM, NSTRM)
PRUNIT(JUNIT, JSTRM, NUNIT) = Ø
PRUNIT(JUNIT, JSTRM, NSTRM) = Ø
PRUNIT(IUNIT, ISTRM, NUNIT) = Ø
PRUNIT(IUNIT, ISTRM, NSTRM) = Ø

$

PRUNIT(IUNIT, PDATA, SYMB) = Ø

# ERASE BUTTONS
LPBOSP(1) = Ø
CALL PRAMTR(OFFSET, OFF, LPBOSP)

# TELL USER FLOWSHEET IS NOT UP-TO-DATE
CALL MSG (INCOR)
RETURN

END
INTEGER FUNCTION ISLCT (LPEDT, IEDT)

# SELECT A LIGHT PEN BUTTON HIT

# INCLUDED FILES:
# GROEF

# CALLED SUBROUTINES:
# GRAPHICS PACKAGE, MSG

INCLUDE GROEF  # GRAPhICS MNEUMONICS

INTEGER LPEDT  # EDIT ADDRESS FOR TURNING LP ON AND OFF
INTEGER IEDT(128)  # EDIT ADDRESSES FOR BLINKING THE BUTTONS
INTEGER DX, DY  # COORDINATE OF THE BEAM ON A LP HIT
INTEGER NAMER  # VALUE OF THE NAME REGISTER ON A LP HIT
LOGICAL BTNS1(6), BTNS2(6)  # STATE OF THE PB'S ON A PB HIT
INTEGER IWHICH  # CODE FOR LP OR PB HIT
REAL MSGLP(10)  # MESSAGE FOR WHILE WAITING FOR LP HIT
REAL MSGPB(10)  # MESSAGE FOR WHILE WAITING FOR PB HIT

DATA MSGLP / 'SELECT', 'T ITE', 'M HIT', 'H LIG', 'HT PE', 'N', '4 o', '/'
DATA MSGPB / 'PUSH', 'PUTTO', 'N 6 T', 'D ENT', 'ER: 0', 'R 9 T', 'O CLE', 'AR', '2 o', '/

REPEAT
%
# GET LIGHT PEN HIT
CALL GETPSH (MSGLP)
CALL GETPSH (BTNS2)
REPEAT
%
CALL LTORPB (DX, DY, NAMER, BTNS1, IWHICH, WAIT)
IF (BTNS1(1) /= BTNS2(1))
%
CALL REPLOT (NEWPRAMTR, LIGHTPEN+OFFSET, OFF, OFF, LPEDT)
CALL HCOPY
CALL REPLOT (NEWPRAMTR, LIGHTPEN+OFFSET, ON, OFF, LPEDT)
BTNS2(1) = BTNS1(1)
%
) UNTIL (IWHICH == 1)
IF (NAMER == 0)
  INAME = 128
ELSE
  INAME = NAMER
%
# BLINK BUTTON AND TURN LIGHT PEN OFF
CALL REPLOT (NEWPRAMTR, BLINK + NAMEREG, ON, NAMER, IEDT(NAMER))
CALL REPLOT (NEWPRAMTR, LIGHTPEN + OFFSET, OFF, OFF, LPEDT)
%
# VERIFY WITH PB
CALL MSG (MSGPB)
REPEAT
%
CALL GETPSH (BTNS1)
CALL LTORPB (DX, DY, I, BTNS2, IWHICH, WAIT)
%
) UNTIL ( (BTNS1(5) /= BTNS2(5)) * (BTNS1(6) /= BTNS2(6)) )
IF (BTNS1(5) /= BTNS2(5))
%
# CLEAR ENTRY
CALL REPLOT (NEWPRAMTR, BLINK+NAMEREG, OFF, NAMER, IEDT(NAMER))
CALL RELOT (NEWPRAMTR, LIGHTPEN*OFFSET, ON, OFF, LPEOT)

$)
UNTIL (BTNS1(6) \= BTNS2(6))

# RETURN VALUE
ISLCT = INAME
RETURN
END
SUBROUTINE ADDS

# ADD A STREAM TO THE FLOWSHEET

# INCLUDED FILES:
# GRDEF, PROCL, LPDCL, UPDCL, PRCOM, LPCOM, UPCOM

# CALLED SUBPROGRAMS:
# GRAPHICS PACKAGE, ISLCT

INCLUDE GRDEF       # GRAPHICS MNEUMONICS
INCLUDE PROCL       # PROCESS TOPOLOGY
INCLUDE LPDCL       # LIGHT PEN BUTTON VARIABLES
INCLUDE UPDCL       # UNIT PICTURE DISPLAY FILES

INTEGER LPEDT       # EDIT ADDRESS FOR TURNING LP ON AND OFF
INTEGER IEDT(128)   # EDIT ADDRESS FOR IDENTIFYING BUTTONS
INTEGER IX, IY      # FOR CALCULATING COORDINATES
INTEGER IUNIT(2)    # UNIT NUMBER CONNECTED BY ADDED STREAM
INTEGER ISTRM(2)    # STREAM NUMBERS BEING CONNECTED
INTEGER IENTRY(128, 2) # UNIT AND STREAM ASSOCIATED WITH EACH BUTTON
INTEGER IFREE       # NEXT AVAILABLE SPACE IN IENTRY
REAL INCOR          # MESSAGE THAT THE FLOWSHEET IS NOT UP-TO-DATE

INCLUDE PRCOM       # PROCESS COMMON
INCLUDE LPCOM       # LIGHT PEN COMMON
INCLUDE UPCOM       # UNIT PICTURE COMMON

DATA INCOR / 'FLOWS', 'HEET ', 'IS NO', 'T UP-', 'TO-DA', 'TE ', 4 0 ' /

# DRAW LIGHT PEN BUTTONS:
# A TRIANGLE AT EVERY STREAM INTERSECTION THAT EXISTS
# BUT IS NOT CURRENTLY CONNECTED.

LPBDSP(I) = 0
CALL PRAMTR (LIGHTPEN * OFFSET, ON, OFF, LPBDSP, LPEDT)
IFREE = 1
DO I = 1, MAX*UNIT
$(
  IF (PRUNIT(I, PDATA, SYMB) \= 0)
  $(
    # EXISTS, CHECK EACH STREAM
    UNIT = PRUNIT(I, PDATA, SYMB)
    H = PRUNIT(I, PDATA, HAND)
    DO J = 1, NSTRMS
      $(
        IF (USTRM(UNIT, J, STROIR) \= 0)
        BREAK # DOES NOT EXIST
      # STREAM EXISTS
      IF (PRUNIT(I, J, NUNIT) \= 0)
      $(
        # NOT CONNECTED, SO DRAW BUTTON
        CALL POINT (0, 0) # NEUTRAL CORNER
        CALL PRAMTR (BLINK*NAMEREG, OFF, IFREE, LPBDSP, IEDT(IFREE))
        IENTRY(IFREE, 1) = I
        IENTRY(IFREE, 2) = J
        IFREE = IFREE + 1
        IX = PRUNIT(I, COORD, XLOC)
        IY = PRUNIT(I, COORD, YLOC)
        IY = IY + USTRM(UNIT, J, Y)
        IF (H \= LUNIT)
          IX=IX*USIZE(UNIT,X)-USTRM(UNIT,J,X)
        ELSE
          IX = IX + USTRM(UNIT, J, X)
      )
    )
  )
$(
$)
CALL POINT (IX, IY)
CALL COPY (0, SBNDSP, LPRDSP)
$)
$)
$)
CALL PRAMTR (LIGHTPEN + BLINK, OFF, OFF)
IF (IFREE == 1)
    RETURN
IFREE = ISLCT (LPEDT, IEDT)
IUNIT(1) = IENTRY(IFREE, 1)
ISTRM(1) = IENTRY(IFREE, 2)
CALL REPLOT (NEWPRAMTR, LIGHTPEN + OFFSET, ON, OFF, LPEDT)
IFREE = ISLCT (LPEDT, IEDT)
IUNIT(2) = IENTRY(IFREE, 1)
ISTRM(2) = IENTRY(IFREE, 2)
# MAKE ENTRIES
PRUNIT(IUNIT(1), ISTRM(1), NUNIT) = IUNIT(2)
PRUNIT(IUNIT(1), ISTRM(1), NSTRM) = ISTRM(2)
PRUNIT(IUNIT(2), ISTRM(2), NUNIT) = IUNIT(1)
PRUNIT(IUNIT(2), ISTRM(2), NSTRM) = ISTRM(1)
# ERASE BUTTONS
LPBDSP(1) = 0
CALL PRAMTR (OFFSET, OFF, LPBDSP)
# TELL USER FLOWSHEET IS NOT UP-TO-DATE
CALL MSG (INCOR)
RETURN
END
SUBROUTINE DELS
  
# DELETE A STREAM FROM THE FLOWSHEET
  
# INCLUDED FILES:
#  GROEF, PROCL, LPDCL, UPDCL, PRCOM, LPCOM, UPCOM

# CALLED SUBPROGRAMS
#  GRAPHICS PACKAGE, ISLCT

INCLUDE GROEF  # GRAPHICS MNEUMONICS
INCLUDE PROCL  # PROCESS TOPOLOGY
INCLUDE LPDCL  # LIGHT PEN BUTTON VARIABLES
INCLUDE UPDCL  # UNIT PICTURE VARIABLES

INTEGER LPEDT  # EDIT ADDRESS FOR TURNING LP ON AND OFF
INTEGER IEDT(128)  # EDIT ADDRESS FOR IDENTIFYING BUTTONS
INTEGER IX, IY  # FOR CALCULATING COORDINATES
INTEGER IUNIT(2)  # UNIT NUMBERS CONNECTED
INTEGER ISTRM(2)  # STREAM NUMBERS CONNECTED
INTEGER IENTRY(128, 2)  # UNIT AND STREAM ASSOCIATED WITH EACH BUTTON
INTEGER IFREE  # NEXT AVAILABLE SPACE IN IENTRY
REAL INCOR(10)  # MESSAGE THAT THE FLOWSHEET IS NOT UP-TO-DATE

INCLUDE PRCOM  # PROCESS COMMON
INCLUDE LPCOM  # LIGHT PEN BUTTON DISPLAY FILE
INCLUDE UPCOM  # UNIT PICTURE DISPLAY FILE

DATA INCOR / 'FLOWS', 'HEET', 'IS NO', 'T UP-', 'TO-DA', 'TE', 'A', 'E' /

# DRAW LIGHT PEN BUTTONS:
#  A TRIANGLE AT EVERY STREAM INTERSECTION THAT EXISTS
#  AND IS CONNECTED.
I.PB DSP(1) = 0
CALL PRAMTR (LIGHTPEN * OFFSET, ON, OFF, LPBDSP, LPEDT)
IFREE = 1
DO I = 1, MAX-UNIT
  $( IF (PRUNIT(I, PDATA, SYMB) \= 0)
    $(
      # UNIT EXISTS, CHECK EACH STREAM
      UNIT = PRUNIT(I, PDATA, SYMB)
      H = PRUNIT(I, PDATA, HAND)
      DO J = 1, NSTRMS
        $(
          # STREAM EXISTS
          IF (PRUNIT(J, NUNIT) \= 0)
            # CONNECTED, DRAW BUTTON
            CALL POINT (0, 0)  # NEUTRAL CORNER
            CALL PRAMTR (BLINK * NAMEREG, OFF, IFREE, LPBDSP, IEDT(IFREE))
            IENTRY(IFREE, 1) = I
            IENTRY(IFREE, 2) = J
            IFREE = IFREE + 1
            IX = PRUNIT(I, COORD, XLOC)
            IY = PRUNIT(I, COORD, YLOC)
            IY = IY + USTRM(UNIT, J, Y)
            IF (H == LUNIT)
              IX = IX + USIZE(UNIT, X) - USTRM(UNIT, J, X)
            ELSE
              IX = IX + USTRM(UNIT, J, X)
          )
        )
      )
    )
  )
)
CALL POINT (IX, IY)
CALL COPY (0, SBNDSP, LPBOSP) $)
$)
CALL PRAMTR (LIGHTPEN * BLINK, OFF, OFF)
IF (IFREE == 1)
  RETURN
IFREE = ISLCT (LPEDT, IEDT)
IUNIT(1) = IENTRY(IFREE, 1)
ISTRM(1) = IENTRY(IFREE, 2)
IUNIT(2) = PRUNIT(IUNIT(1), ISTRM(1), NUNIT)
ISTRM(2) = PRUNIT(IUNIT(1), ISTRM(1), NSTRM)

# DELETE ENTRIES
DO I = 1, 2
  $%
  PRUNIT(IUNIT(I), ISTRM(I), NUNIT) = 0
  PRUNIT(IUNIT(I), ISTRM(I), NSTRM) = 0
  $%

# ERASE BUTTONS
LPBOSP(1) = 0
CALL PRAMTR (OFFSET, OFF, LPBOSP)

# TELL USER FLOWSHEET NOT UP-TO-DATE
CALL MSG (INCP)
RETURN
END
SUBROUTINE ADDT
# ADD USER DEFINED TITLE TO THE DISPLAY
# INCLUDED FILES:
# LUNS
# CALLED SUBROUTINES:
# MSG
INCLUDE LUNS
REAL TITLE(10)
WRITE (USER, 1)
   1 FORMAT ('ADDT - ENTER TITLE: UP TO 49 CHARACTERS.')
READ (USER, 2) TITLE
   2 FORMAT (10A5)
CALL MSG (TITLE)
RETURN
END
SUBROUTINE D R A W F ( I T Y P E )
# DRAW THE FLOWSHEET

INCLUDE MBDCL
INCLUDE PROCL
INCLUDE FLOCL

INTEGER ITYPE
MACRO(NGPH,2)
REAL GPH(3, NGPH)
INTEGER IGPH

INCLUDE MBCOM
INCLUDE PRCOM
INCLUDE FLCOM

DATA GPH / 'THIS ', 'FLOWS', 'HEET ', 'RESTO ', 'RE GR', 'APH '/

IF (ITYPE == 1)
$ (
# GRAPH
IGPH = MENU (NGPH, GPH)
IF (IGPH == 1)
CALL DRPRP (0) # THIS FLOWSHEET
ELSE
CALL DRPRP (1) # RESTORE FLOWSHEET
CALL DGRPH # DRAW THE FLOWSHEET
$)
ELSE
$ (CALL DRPRP (0) # THIS FLOWSHEET
CALL DRFLW # DRAW THE FLOWSHEET
$)

RETURN
END
# FILE NAME FLOGL RAT
# TRANSLATION BETWEEN FLOWSHEET AND GRAPH

INTEGER GPHNDE(MAX=UNIT, 4)  # VERTEX NUMBER IN ABSTRACT GRAPH
INTEGER FLWNDE(MAX=VERTEX, 2)  # UNIT AND STREAM IN FLOWSHEET
MACRO(FUNIT, 1)
MACRO(FSTRM, 2)

# FILE NAME FLCOM
# FLOWSHEET <=> GRAPH

COMMON /FLCOM/ GPHNDE, FLWNDE
SUBROUTINE ORPRP (TYPE)

# DO PLANARITY ALGORITHM WITH GRAPH INDICATED BY TYPE

INTEGER TYPE

IF (TYPE == 1)
    CALL RSTRA  # GET GRAPH OFF DISK
ELSE
    CALL MAKEF  # GET FROM CURRENT FLOWSHEET

CALL ORDRA  # CONVERT TO PALM TREE AND SORT ADJACENCY LISTS
CALL EMBED  # PRODUCE THE DEPENDENCY GRAPH
CALL COLOR  # COLOR SAID GRAPH
CALL ORDRB  # SORT ADJACENCY LISTS FOR DRAWING

RETURN
END
SUBROUTINE RSTRA

# RESTORE THE PLANARITY ALGORITHM DATA STRUCTURES FROM DISK THAT WERE PUT
# THERE BY SAVEA, USED FOR DEBUGGING.

DOUBLE INTEGER NAME  # FILE NAME
INCLUDE PLCL
INCLUDE PLCOM

MACRO(DISK, 17)
MACRO(USER, 25)

WRITE (USER, 1)
   1 FORMAT (' RSTRA - ENTER FILE NAME: UP TO 5 LETTERS IN QUOTES.')
READ (USER, ) NAME
CALL SEEK (DISK, NAME, 'GPH', IEV)
CALL WAITFR (IEV)
IF (IEV <= 0)
   %
   WRITE (USER, 2) IEV, NAME
   2 FORMAT (' *** RSTRA - SEEK ERROR', 14, ' ON ', A5, ' GPH')
   CALL EXIT
%
READ (DISK, 3) APOOL, A, ALIST
   3 FORMAT (20(I, I4))
CALL CLOSE (DISK)
RETURN
END
# FILE NAME PLOCL RAT

# MACROS FOR PLANARITY ALGORITHM AND DECLARATIONS FOR PLCOM

MACRO(LSTLUN,16) # LUN FOR LISTINGS
MACRO(ERRLUN,21) # LUN FOR ERROR MESSAGES
MACRO(FP,1) # MNEUMONIC FOR FORWARD POINTER
MACRO(BP,2) # MNEUMONIC FOR BACKWARD POINTER
MACRO(VRTX1,2) # VERTEX THAT NODE IN LINKED LIST REPRESENTS
MACRO(VRTX2,3) # EDGE GOES FROM VRTX1 TO VRTX2
MACRO(TYPE,4) # TYPE OF EDGE THAT NODE REPRESENTS
MACRO(ARC,1) # EDGES CAN BE TREE ARCS
MACRO(FROND,2) # OR FRONDS
MACRO(NULL,0) # OR NOT IN PALM TREE AT ALL.

# THE FOLLOWING MACROS ARE USED TO DIMENSION VARIOUS DATA STRUCTURES
# IN VARIOUS ROUTINES. THEY SHOULD ALL BE REVIEWED SIMULTANEOUSLY WHEN
# CHANGING THE NUMBER OF PROCESS UNITS TO BE HANDLED.

MACRO(MAX*VERTEX,49) # MAXIMUM NUMBER OF VERTICES ALLOWED
MACRO(MAX*EDGE,92) # MAKE 2 * MAX*VERTEX * 2.
MACRO(STKSIZE,94) # MAKE MAX*EDGE * 2
MACRO(BKTLIM,91) # MUST BE EQUAL TO 2 * MAX*VERTEX * 1
MACRO(FLIM,48) # MAKE MAX*EDGE * MAX*VERTEX * 1
MACRO(POOL*SIZE,135) # SIZE OF POOL OF AVAILABLE NODES FOR
# ADJACENCY LISTS (TYPICALLY MAKE 3 * MAX*VERTEX)

INTEGER A(MAX*VERTEX, 2) # A(VERTEX, FP) AND A(VERTEX, BP) CONSTITUTE
INTEGER APool # LIST HEAD FOR VERTEX'S ADJACENCY LIST
INTEGER ALIST(POOL*SIZE, 4) # ADJACENCY LIST NODES;
# SECOND INDEX -
# FP1: POINTER TO NEXT NODE IN LIST
# VRTXI: OWNER OF LIST
# VRTX2: EDGE FROM VRTX1 TO VRTX2
# TYPE1: ABOVE EDGE IS EITHER A TREE
# OR A FROND
INTEGER UNNUMB(MAX*VERTEX) # MAPPING FOR GETTING BACK TO ORIGINAL VERTEX
# NUMBERING.

# FILE NAME PLOCL RAT

# DATA STRUCTURES FOR PLANARITY ALGORITHM

COMMON /PLCOM/ A, APool, ALIST, UNNUMB
SUBROUTINE MAKEF

# CREATE AN ABSTRACT GRAPH IN PLCOM FROM THE FLOWSHEET DATA IN PRCOM,
# KEEP TRANSLATION DATA IN FLCOM.

# INCLUDED FILES:
# PRDCL, PRCOM, PLDCL, PLCOM, FLDCL, FLCOM

# CALLED SUBPROGRAMS:
# LINKA

INCLUDE PLOCL
INCLUDE PROCL
INCLUDE FLOCL

INTEGER V, W
INTEGER IUNIT, ISTRM
INTEGER IFIRST, ILAST
INTEGER ANEXT

INCLUDE PLCOM
INCLUDE PRCOM
INCLUDE FLCOM

# INITIALIZE ABSTRACT GRAPH DATA STRUCTURES
# ADJACENCY LIST-HEADS:
DO I = 1, MAX-VERTEX
   A(I, FP) = 0
   A(I, BP) = 0
END

# LINK POOL OF AVAILABLE NODES
4POOL = 1
LIMIT = POOL-SIZE - 1
DO I = 1, LIMIT
   ALIST(I, FP) = I + 1
   ALIST(Pool-SIZE, FP) = 0
   ANEXT = 1

# CHECK EACH POSSIBLE UNIT
DO IUNIT = 1, MAX-UNIT
   IF (PRUNIT(IUNIT, PDATA, SYMB) \= 0)
      # CHECK EACH STREAM
      IFIRST = 0
      ILAST = 0
      DO ISTRM = 1, 4
         IF (PRUNIT(IUNIT, ISTRM, NUNIT) \= 0)
            IF (IFIRST == 0)
               IFIRST = ISTRM
               GPHNDE(IUNIT, ISTRM) = ANEXT
               FLWDE(ANEXT, FUNIT) = IUNIT
               FLWDE(ANEXT, FSTRM) = ISTRM
               ANEXT = ANEXT + 1
       IF (ILAST \= 0)
          # MAKE LINK BETWEEN ILAST AND ISTRM
          V = GPHNDE(IUNIT, ILAST)
          W = GPHNDE(IUNIT, ISTRM)
          CALL LINKA (V, W)
          CALL LINKA (W, V)
   END
ILAST = ISTRM
$
$
IF (IFIRST \neq ILAST)
$
$
# LINK THEM
V = GPHNDE(IUNIT, IFIRST)
W = GPHNDE(IUNIT, ILAST)
CALL LINKA(V, W)
CALL LINKA(W, V)
$
$
IF (IFIRST == 0)
$
$
# PUT ISOLATED UNIT IN GRAPH
GPHNDE(IUNIT, 1) = ANEXT
FLWNDE(ANEXT, FUNIT) = IUNIT
FLWNDE(ANEXT, FSTRM) = 1
ANEXT = ANEXT + 1
$
$
$
# NOW DO THE INTERCONNECTIONS
DO IUNIT = 1, MAX*UNIT
$
$
IF (PRUNIT(IUNIT, PDATA, SYMB) \neq 0)
$
$
DO ISTRM = 1, 4
$
$
IF (PRUNIT(IUNIT, ISTRM, NUNIT) \neq 0)
$
$
V = GPHNDE(IUNIT, ISTRM)
W = GPHNDE(PRUNIT(IUNIT, ISTRM, NUNIT),
PRUNIT(IUNIT, ISTRM, NSTRM))
CALL LINKA(V, W)
$
$
$
RETURN
END
SUBROUTINE LINKA (VERTEX, EDGE)

# ADD EDGE TO THE ADJACENCY LIST FOR VERTEX

INTEGER VERTEX          # OWNER OF THE ADJACENCY LIST
INTEGER EDGE            # EDGE TO BE ADDED
INTEGER NEWNDE          # NEW NODE SELECTED FROM AVAILABLE POOL
INTEGER OLDNDE          # LAST NODE IN LIST (BEFORE ADDING NEWNDE)

INCLUDE PLDCL          # DECLARATIONS FOR PLANARITY ALGORITHM DATA STRUCTURES
INCLUDE PLCOM          # NAMED COMMON FOR THOSE DATA STRUCTURES

NEWNDE = APool          # IDENTIFY NEW NODE FROM AVAILABLE POOL
IF (NEWNDE == 0)
  $(
    WRITE (ERRLUN, 1)
    1 FORMAT (’ LINKA - ADJACENCY LIST POOL EMPTY’)
    CALL EXIT
  )
OLDNDE = A(VERTEX, BP)     # IDENTIFY LAST NODE ON LIST
APool = ALIST(NEWNDE, FP)  # REMOVE NEWNDE FROM AVAILABLE POOL
ALIST(NEWNDE, VRTX1) = VERTEX  # GIVE NEWNDE A VALUE
ALIST(NEWNDE, VRTX2) = EDGE

# LINK NEWNDE TO ADJACENCY LIST FOR VERTEX
IF (OLDNDE == 0)
  A(VERTEX, FP) = NEWNDE       # LINK DIRECTLY TO LIST HEAD
ELSE
  ALIST(OLDNDE, FP) = NEWNDE   # LINK TO END OF CURRENT LIST
  A(VERTEX, BP) = NEWNDE       # IN ANY CASE, THE LAST NODE GETS
                             # LINKED TO THE LIST HEAD
RETURN
END
SUBROUTINE ORDRA

# DO A DEPTH-FIRST-SEARCH TO TURN A GRAPH, G, DEFINED BY THE ADJACENCY LISTS
# IN PLOM, TO A PALM TREE, P, THEN RENUMBER THE VERTICES AND SORT THE
# ADJACENCY LISTS TO PREPARE FOR PATH FINDING.

MACRO(RECUR,1200)   # RECURSION ENTRY POINT
MACRO(RETRN,1201)   # RECURSION RETURN POINT

INCLUDE PLOM  # DECLARATIONS FOR PLANARITY ALGORITHM DATA STRUCTURES

INTEGER V       # START DEPTH FIRST SEARCH AT VERTEX V
INTEGER U       # V'S FATHER
INTEGER W       # CURRENT VERTEX
INTEGER POINT   # POINTER TO NEXT VERTEX TO VISIT
INTEGER NUMBER(MAX+VERTEX) # DFS RENUMBERS VERTICES IN THIS VECTOR
INTEGER N       # COUNTER USED TO ASSIGN NUMBERS TO NUMBER
INTEGER STACK(MAX+VERTEX, 5) # RECURSION STACK FOR SAVING EACH LEVEL'S
                                # LOCAL VARIABLES.
INTEGER STKPT   # STACK POINTER FOR STACK
INTEGER LOWPT1(MAX+VERTEX) # LOWPT1(V) = LOWEST VERTEX BELOW V REACHABLE
                                # BY A FROND FROM A DESCENDANT OF V
INTEGER LOWPT2(MAX+VERTEX) # LOWPT2(V) = SECOND LOWEST VERTEX BELOW V
                                # REACHABLE BY A FROND FROM A DESCENDANT OF V
INTEGER AKT(AKTLIM) # HEADERS FOR LISTS OF EDGES WITH EQUAL PHI'S
INTEGER PHI     # SORT VARIABLE FOR EDGES
INTEGER OLDPNT, NXTPNT # TEMPORARY VARIABLES FOR USE IN LIST PROCESSING

INCLUDE PLCOM  # NAMED COMMON FOR PLANARITY ALGORITHM DATA STRUCTURES

STKPT = 1       # INITIAL RECURSION STACK POINTER
N = m           # PREPARE TO RENUMBER THE VERTICES
DO I = 1, MAX-VERTEX
    NUMBER(I) = 0
V = 1
U = m          # V'S ANCESTOR
CALL DFS(V, U)  # EXPLORE G WITH A DEPTH-FIRST-SEARCH STARTING AT V = 1

# DFS (V, U)
RECUR
N = N + 1
NUMBER(V) = N
LOWPT1(V) = N
LOWPT2(V) = N
NXTPNT = A(V, FP)  # START AT HEADER OF V'S LIST
WHILE (NXTPNT /= 0)
    POINT = NXTPNT
    NXTPNT = ALIST(POINT, FP)
    W = ALIST(POINT, VRTX2)
    IF (NUMBER(W) == 0)
        IF W IS NEW VERTEX, MARK (V, W) AS AN ARC OF THE SPANNING TREE
            LIST(POINT, TYPE) = ARC
        # CALL DFS (W, V)
            # SAVE LOCAL VARIABLES ON STACK FOR RECURSIVE CALL
            STACK(STKPT, 1) = U
            STACK(STKPT, 2) = V
            STACK(STKPT, 3) = W
            STACK(STKPT, 4) = POINT
            STACK(STKPT, 5) = NXTPNT
            STKPT = STKPT + 1
            # BIND ARGUMENTS FOR THE CALL
        ENDIF
    ENDIF
U = V
V = W
GO TO RECUR

RETRN
# RETURNED FROM RECURSION
STKPT = STKPT - 1
U = STACK(STKPT, 1)
V = STACK(STKPT, 2)
W = STACK(STKPT, 3)
POINT = STACK(STKPT, 4)
NXTPNT = STACK(STKPT, 5)

IF (LOWPT1(W) < LOWPT1(V))
  \$
  LOWPT2(V) = \text{MIN0} (LOWPT1(V), LOWPT2(W))
  LOWPT1(V) = LOWPT1(W)
  \$
ELSE IF (LOWPT1(W) = LOWPT1(V))
  \$
  LOWPT2(V) = \text{MIN0} (LOWPT2(V), LOWPT2(W))
  LOWPT1(V) = LOWPT1(W)
  \$
ELSE IF (LOWPT1(W) > LOWPT1(V))
  \$
  LOWPT2(V) = \text{MIN0} (LOWPT2(V), LOWPT1(W))
  LOWPT1(V) = LOWPT1(W)
  \$
ELSE
  \$
  # EDGE NOT IN P
  ALIST(POINT, TYPE) = NULL
  \$
ENDIF
# CHECK FOR RECURSION AND RETURN TO CORRECT PLACE
IF (STKPT = 1)
  GO TO RETRN
# END OF DFS

# UNNUMB WILL MAP NEW VERTEX NUMBERS BACK TO OLD WHEN TIME COMES
DO I = 1, MAX-VERTEX
  UNNUMB(I) = 0
DO I = 1, MAX-VERTEX
  IF (NUMBER(I) = 0)
    UNNUMB(NUMBER(I)) = I
# RENUMBER VERTICES ACCORDING TO RESULTS OF DFS WHILE
# SORTING THE ADJACENCY LISTS,
DO I = 1, BKTLIM
  BKT(I) = 0
  # EMPTY LIST HEADER
DO I = 1, MAX-VERTEX
  # FOR EACH ADJACENCY LIST
  \$
  NXTPNT = A(I, FP)
  # START AT HEADER
  WHILE (NXTPNT = 0)
    \$
    POINT = NXTPNT
    NXTPNT = ALIST(POINT, FP)
    # REMOVE (POINT) FROM I'S LIST
    A(I, FP) = ALIST(POINT, FP)
    IF (ALIST(POINT, TYPE) = NULL)
      \$
# PUT NODE BACK IN POOL
ALIST(POINT, FP) = APOOL
APOOL = POINT
$

ELSE
$
# PUT NODE IN PROPER "BUCKET"
J = ALIST(POINT, VRTX2)
LOW1 = LOWPT1(J)
LOW2 = LOWPT2(J)
# RENUMBER THE VERTICES FOR THIS EDGE
V = NUMBER(I)
w = NUMBER(J)
ALIST(POINT, VRTX1) = V
ALIST(POINT, VRTX2) = W
# COMPUTE PHI
IF (ALIST(POINT, TYPE) == FRONT)
   PHI = 2 * W
ELSE  # TREE ARC
   $(
      PHI = 2 * LOW1
      IF (LOW2 < V)
         PHI = PHI + 1
   )$
# ADD (V, W) TO BKT(PHI)
ALIST(POINT, FP) = BKT(PHI)
BKT(PHI) = POINT
$
$
A(I, BP) = Ø  # SINCE EMPTIED BY LOOP
$
# ALL A'S NOW EMPTY, NODES MOVED TO BKT'S
# FILL A'S BACK UP IN ORDER OF NON-DECREASING PHI
DO I = 1, BKTLIM
   WHILE (BKT(I) \= Ø)
   $(
      # PICK NODE
      POINT = BKT(I)
      BKT(I) = ALIST(POINT, FP)
      # LINK TO PROPER ADJACENCY LIST
      V = ALIST(POINT, VRTX1)
      OLDPNT = A(V, BP)
      A(V, BP) = POINT
      ALIST(POINT, FP) = Ø
      IF (OLDPNT =Ø)
         A(V, FP) = POINT
      ELSE
         ALIST(OLDPNT, FP) = POINT
   )$
RETURN
END
SUBROUTINE EMBED
# SPECIFY A PLANAR EMBEDDING OF THE PALM TREE REPRESENTED BY THE PROPERLY
# ORDERED (BY ORDE) ADJACENCY LISTS IN PLCOM, DO THIS BY SPECIFYING ON WHICH
# SIDE (LEFT OR RIGHT) EACH FROND SHOULD BE PLACED.

INCLUDE MBODL       # DECLARATIONS AND
INCLUDE MBCOM       # NAMED COMMON FOR THIS ALGORITHMS VARIABLES,

START = 0           # READY TO START FIRST PATH
PATH = 0             # PREPARE FOR THAT PATH TO BE NO, 1
FIRSTP(1) = 1       # FIRST PATH TO PASS THROUGH VERTEX 1 WILL BE NO, 1
AVAIL = 1           # START WITH EMPTY STACK EXCEPT FOR HEADERS
MTLEFT = MAX-EDGE + 1 # THE STACK NODES AT MTLEFT AND MTRIGHT ARE USED
MTRIGHT = MAX-EDGE + 2 # AS STACK POINTERS, THEIR ENTRIES ARE 0 SO THAT
STACK(MTLEFT, ENTRY) = 0 # THEY WILL ALWAYS BE LESS THAN ANY OTHER
STACK(MTRIGHT, ENTRY) = 0 # STACK ENTRY.
STACK(MTLEFT, NEXT) = MTLEFT
STACK(MTRIGHT, NEXT) = MTRIGHT
STK = EMPTY         # BUNDLE STACK EMPTY ALSO

# INITIALIZE DEPENDENCY GRAPH
DO I = 1, STKSIZ
$<
   PHEAD(I) = EMPTY
   PSIDE(I) = EMPTY
$>

MACRO(RECUR,1080) # RECURSION ENTRY POINT
MACRO(RETRN,1001) # RECURSION RETURN POINT
RSPNT = EMPT     # START AT TOP LEVEL OF RECURSION

V = 1               # START ALGORITHM AT VERTEX 1
# CALL PLANAR (V)

# SUBROUTINE PLANAR (V)
# PLANAR EXPLORES A PALM TREE, P, WITH A DEPTH-FIRST-SEARCH AND
# EMBEDS IT IN THE PLANE - WITH NO CROSSED EDGES IF POSSIBLE.
# INTEGER V, W       # (V, W) IS THE EDGE BEING EXPLORED AT ANY ONE TIME
# INTEGER POINT, NXPNT # USED TO PROCESS V'S ADJACENCY LIST

RECUR
# START AT V'S LIST HEAD
NXPNT = A(V, FP)

WHILE (NXPNT \= EMPTY)
$<
   # STEP THROUGH ADJACENCY LIST
   POINT = NXPNT
   NXPNT = ALIST(POINT, FP)
   W = ALIST(POINT, VRTX2)
   IF (START == 0)
      $<
         # START NEW PATH
         START = V
         PATH = PATH + 1
      $>
   ALIST(POINT, PNUM) = PATH
   IF (V < W)
      $<
         # (V, W) IS AN ARC OF THE SPANNING TREE.
         # THEREFORE, WE KNOW THAT W HAS NOT BEEN SEEN BEFORE,
FIRSTP(W) = PATH

# CALL PLANAR (W)  # PROCEED WITH DEPTH-FIRST-SEARCH
# SAVE "LOCAL" VARIABLES
RSPNT = RSPNT + 1
RSTK(RSPNT, 1) = V
RSTK(RSPNT, 2) = W
RSTK(RSPNT, 3) = POINT
RSTK(RSPNT, 4) = NXTPNT
# "BIND" ARGUMENT FOR CALL
V = W
GO TO RECUR

RETRN
# RETURNED FROM "RECURSIVE" CALL
V = RSTK(RSPNT, 1)
W = RSTK(RSPNT, 2)
POINT = RSTK(RSPNT, 3)
NXTPNT = RSTK(RSPNT, 4)
RSPNT = RSPNT - 1

# THE SUB-SPANNING TREE WITH FIRST EDGE (V, W)
# HAS NOW BEEN COMPLETELY EXPLAINED.
# NOTE THAT NO FROND RETURNING TO A VERTEX >= V CAN
# INTERFERENCE WITH THOSE LEFT TO BE DISCOVERED SINCE
# A PALM TREE HAS NO CROSS-EDGES,
# THEREFORE, DELETE ALL STACK ENTRIES AND BUNDLES
# CORRESPONDING TO VERTICES >= V, LINK THE DELETED
# ENTRIES INTO THE DEPENDENCY GRAPH.
CALL OPEN

IF ( (FIRSTP(W) /= FIRSTP(V)) & (START /= V) )

( (FIRSTP(W) /= FIRSTP(V)) & (START /= V) )

# ALL OF THE SEGMENT WITH FIRST EDGE (V, W) HAS BEEN
# EMBEDDED, THIS SEGMENT IS WITH RESPECT TO A CYCLE
# FORMED BY THE FIRST PATH PASSING THROUGH W PLUS THE
# PATH FROM THE FINISH VERTEX OF THE FIRST PATH TO PASS
# THROUGH W, FINISH(FIRSTP(W)), TO V.
# THEREFORE, IT IS NOW NECESSARY TO BE ABLE TO EMBED
# ALL FRONDS ENTERING ABOVE FINISH(FIRSTP(W)) ON THE
# LEFT.
# WE WILL DO THIS TO THE EXTENT POSSIBLE, BUT RATHER
# THAN JUST DECLARE A GRAPH NON-PLANAR, WE WILL DRAW
# IT ANYWAY WITH SOME SORT OF SUB-OPTIMAL NUMBER OF
# EDGE CROSSINGS.
OLDX = MLEFT  # USE OLDX AND OLDY TO POINT TO THE
OLDY = MRIGHT # TOP OF THE BUNDLE BEING CONSIDERED.
REPEAT  # FOR EACH BUNDLE OF THE SEGMENT

# JUST EMBEDDED.
CALL POPAB  # X AND Y FROM THE BUNDLE STACK
IF (STACK(X, ENTRY) > FINISH(FIRSTP(W)))

# SINCE LEFT ENTRIES WOULD CONFLICT
# WITH THE CYCLE IF THEY WERE MOVED TO
# THE RIGHT SIDE, THERE'S NO POINT IN
# SWAPPING SIDES.
# IF THERE ARE ALSO ENTRIES ON THE
# RIGHT, THE GRAPH IS NON-PLANAR.
# SINCE WE'RE GOING TO DRAW IT ANYWAY,
# DON'T EVEN BOTHER TO CHECK.
IF (X \= MTLEFT)
  OLDX = X
IF (Y \= MTRGHT)
  OLDY = Y

ELSE IF ((STACK(Y, ENTRY) > FINISH(FIRSTP(W)))
  \& (STACK(STACK(OLDY, NEXT), ENTRY) \= EOS))
  # SWAP SIDES OF THE BUNDLE
  # (THE ENTRIES ON THE LEFT WILL NOT
  # INTERFERE)
  CALL SWPAB
ELSE
  # TIME TO END LOOP
  $(
    TEMP = OLDY
    IF (X \= MTLEFT)
      OLDX = X
    IF (Y \= MTRGHT)
      OLDY = Y
    BREAK
  )$

  # DELETE THE EOS FROM THE RIGHT STACK
  IF (OLDY == STACK(TEMP, NEXT))
    OLDY = TEMP
  STACK(TEMP, NEXT) = STACK(STACK(TEMP, NEXT), NEXT)

  # PUT THE COMBINED, REARRANGED BUNDLE BACK ON THE STACK
  X = OLP DX
  Y = OLDY
  CALL PSHAB

IF (START \= V)
  # BACKING UP A DEAD END PATH, SO START NEW PATH
  # (GRAPH IS NOT BICONNECTED)
  START = 0

ELSE
  $(
    # (V \> W)
    # (V, W) IS A FRONDE OF THE PALM TREE.
    # CURRENT PATH IS COMPLETE.
    FINISH(PATH) = W
    # SAVE POINTER TO THE ORIGINAL CYCLE
    # Embed path on left.
    # IF THE TOP ENTRY ON THE LEFT STACK INTERFERES WITH THIS,
    # WE CAN SWAP SIDES OF THE BUNDLE TO MAKE WAY, IF THE TOP
    # ENTRIES OF BOTH THE LEFT AND RIGHT STACKS INTERFERE, THE
    # GRAPH IS NON-PLANAR, BUT WE WILL DRAW IT ANYWAY. WE MUST
    # DO THIS PROCEDURE WITH ALL BUNDLES ON THE TOP OF THE BUNDLE
    # STACK THAT HAVE EITHER STACK WITH A TOP ENTRY \> W.
    OL DX = MTLEFT
    OLD Y = MTRGHT
    REPEAT
    CALL POPAB    # X, Y FROM BUNDLE STACK
    IF ((STACK(STACK(OLDY, NEXT), ENTRY) \> W)
      \& (Y \= MTRGHT))
      $(
        # RIGHT SIDE WOULD INTERFERE IF MOVED TO LEFT
        # SO LEAVE BUNDLE AS IS.
        IF (X \= MTLEFT)
          OLDX = X
        IF (Y \= MTRGHT)
          OLDY = Y.
    )$
ELSE IF ( (STACK(STACK(OLDX, NEXT), ENTRY) > W) & (X ≠ MTLEFT) )
  # ONLY LEFT SIDE INTERFERES SO
  CALL SWPAB
ELSE
  %(# PUT UNUSED BUNDLE BACK ON STACK AND END LOOP
  CALL PSHAB
  BREAK
  %)
  # NOTE: THE BUNDLE (OLDX, OLDY) IS CURRENTLY OFF THE STACK
  # IF THE CURRENT PATH IS MORE THAN A SINGLE FROND,
  # ADD AN EOS TO THE RIGHT STACK.
  IF (V ≠ START)
    $(
      STACK(AVAIL, ENTRY) = EOS
      STACK(AVAIL, NEXT) = STACK(MTRGHT, NEXT)
      STACK(MTRGHT, NEXT) = AVAIL
      IF (OLDY == MTRGHT)
        OLDY = AVAIL
      AVAIL = AVAIL + 1
      %)
  IF (W > FINISH(FIRSTP(START)))
    $(
      # ADD W TO THE LEFT STACK
      STACK(AVAIL, ENTRY) = W
      STACK(AVAIL, PNTR) = POINT
      STACK(AVAIL, NEXT) = STACK(MTLEFT, NEXT)
      STACK(MTLEFT, NEXT) = AVAIL
      IF (OLDX == MTLEFT)
        OLDX = AVAIL
      AVAIL = AVAIL + 1
      %)
    # PUT COMBINED, REARRANGED AND AUGMENTED BUNDLE ON THE
    # BUNDLE STACK.
    IF ( (OLDX ≠ MTLEFT) * (OLDY ≠ MTRGHT) )
      $(
        X = OLDX
        Y = OLDY
        CALL PSHAB
      %)
    # INDICATE READY TO START NEW PATH.
    START = 0
    %)
  %)
  # CHECK FOR RECURSION AND RETURN TO CORRECT PLACE.
  IF (RSPNT ≠ EMPTY)
    GO TO RETRNT
RETURN
END
# File Name M8OCL RAT - Declarations for Embed

**Include Ploci**

**Include PLocL**

```c
#include plocl
```

**Adjacency List Declarations**

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```

```c
#include plocl
```
MACRO(EDGE1, VTX1)  # FOR USE WITH ALIST NODES USED FOR THE DEPENDENCY
MACRO(EDGE2, VTX2)  # GRAPH,
MACRO(SAME, 1)  # TWO TYPES OF EDGES IN THE DEPENDENCY GRAPH
MACRO(OPPOSITE, 2)
INTEGER PHEAD(STKSIZ)  # ADJACENCY LIST HEADS FOR DEPENDENCY GRAPH
INTEGER PSIDE(STKSIZ)  # EMPTY, LEFT, OR RIGHT EMBEDDING

# FILE NAME MBCOM RAT - VARIABLES FOR EMBED
INCLUDE PLOOM
COMMON /MBCOM/ RSTK, RSPNT, START, FINISH, FIRSTP, PATH, STACK, AVAIL,
MTLEFT, MTRIGHT, BNDSTK, BSTKP, V, W, X, Y, OLDX, OLDY, NOMORE,
TEMP, PHEAD, PSIDE
SUBROUTINE PSHAB

# PUSH A BUNDLE

INCLUDE MBDCIL                  # USE EMBED'S VARIABLES
INCLUDE MBCOM

BSTOP = BSTKP + 1
RNDSTK(BSTKP, LEFT) = X
RNDSTK(BSTKP, RIGHT) = Y
RETURN

END
SUBROUTINE POPAB

# POP A BUNDLE

INCLUDE MBDCCL     # USE EMBED'S VARIABLES
INCLUDE MBCOM

IF (BSTKP /= EMPTY)
  \$
    X = RN DSTK (BSTKP, LEFT)
    Y = BNDSTK (BSTKP, RIGHT)
    B STKP = B STKP - 1
  \$
ELSE  # BUNDLE STACK EMPTY
  \$
    X = M TLEFT
    Y = M TRGHT
  \$
RETURN
END
SUBROUTINE SWPAB

# SWAP A BUNDLE

INCLUDE MBCL     # USE EMBED'S VARIABLES
INCLUDE MBCOM

# OLDX AND OLDY MUST POINT TO THE STACK ELEMENTS THAT POINT TO
# THE TOPS OF THE LEFT AND RIGHT SIDES OF THE BUNDLE.
# X AND Y MUST POINT TO THE BOTTOMS OF THE LEFT AND RIGHT SIDES
# OF THE BUNDLE. OLDX AND OLDY WILL BE UPDATED TO THE NEXT
# BUNDLE.

IF (X == MTLEFT)
  $(
    TEMP = STACK(OLDY, NEXT)
    STACK(OLDY, NEXT) = STACK(Y, NEXT)
    STACK(Y, NEXT) = STACK(OLDX, NEXT)
    STACK(OLDX, NEXT) = TEMP
    OLDX = Y
  )
ELSE IF (Y == MTRGHT)
  $(
    TEMP = STACK(OLDX, NEXT)
    STACK(OLDX, NEXT) = STACK(X, NEXT)
    STACK(X, NEXT) = STACK(OLDY, NEXT)
    STACK(OLDY, NEXT) = TEMP
    OLDY = X
  )
ELSE
  $(
    TEMP = STACK(OLDX, NEXT)
    STACK(OLDX, NEXT) = STACK(OLDY, NEXT)
    STACK(OLDY, NEXT) = TEMP
    TEMP = STACK(X, NEXT)
    STACK(X, NEXT) = STACK(Y, NEXT)
    STACK(Y, NEXT) = TEMP
    OLDX = Y
    OLDY = X
  )

RETURN
END
SURROUNTO DPEND

# DELETE BUNDLES AND STACK ENTRIES FROM EMBED'S DATA STRUCTURES
# THAT ARE NO LONGER IMPORTANT. LINK THE PATHS ASSOCIATED WITH
# DELETED ENTRIES INTO A DEPENDENCY GRAPH.

INCLUDE MBDCI  # USE EMBED'S VARIABLES
INTEGER PATH1, PATH2  # TEMPORARIES USED FOR CREATING LINKS IN THE
# DEPENDENCY GRAPH,
INTEGER BASE(2)  # SUBSCRIPTED VERSION OF (X, Y)
INTEGER MT(2)  # SUBSCRIPTED VERSION OF (MTLEFT, MTRIGHT)
INTEGER ODLTPMP  # USED FOR LAST STACK ENTRY CONSIDERED BEFORE TEMP,

INCLUDE MBCOM

MT(1) = MTLEFT
MT(2) = MTRIGHT

# CONSIDER BUNDLES FROM THE TOP OF THE STACK DOWN.
REPEAT
%( CALL POPAB
BASE(1) = X
BASE(2) = Y
# CREATE AN "OPPOSITE" LINK IFF
# 1. AT LEAST ONE OF THE BOTTOM ENTRIES OF THIS BUNDLE IS
# GOING TO BE DELETED (IT WILL BE UNABLE TO PROPAGATE
# A "SAME" LINK).
# 2. BOTH BOTTOM ENTRIES EXIST,
IF (( (STACK(X, ENTRY) >= V) + (STACK(Y, ENTRY) >= V) )
& (STACK(X, ENTRY) \= EOS) 
& (STACK(Y, ENTRY) \= EOS) )
% PATH1 = ALIST(STACK(X, PNTR), PNUM)
PATH2 = ALIST(STACK(Y, PNTR), PNUM)
CALL DPLNK (PATH1, PATH2, OPPOSITE)
%)

# NOW WORK THROUGH STACK ENTRIES IN THIS BUNDLE THAT ARE NO LONGER
# OF INTEREST.
DO I = 1, 2  # LEFT, RIGHT
%( ODLTPMP = MT(I)
TEMP = STACK(MT(I), NEXT)
WHILE (( ODLTPMP \= BASE(I))
& (STACK(TEMP, ENTRY) >= V) )
%( # DELETE TEMP'S ENTRY
ODLTPMP = TEMP
TEMP = STACK(TEMP, NEXT)
IF (( ODLTPMP \= MT(I))
& (ODLTPMP \= BASE(I))
& (STACK(TEMP, ENTRY) \= EOS) )
%( PATH1 = ALIST(STACK(ODLTPMP, PNTR), PNUM)
PATH2 = ALIST(STACK(TEMP, PNTR), PNUM)
CALL DPLNK (PATH1, PATH2, SAME)
%)
%)
STACK(MT(I), NEXT) = TEMP
IF (STACK(BASE(I), ENTRY) >= V)
BASE(I) = MT(I)
%)
X = BASE(1)
\[ \text{Y} = \text{BASE}(2) \]
\[ \text{UNTIL} \ (X \leq \text{MLEFT}) \land (Y \leq \text{MTRGHT}) \]

\text{CALL PSHAB}
\text{RETURN}
\text{END}
SUBROUTINE DPLNK (PATH1, PATH2, SIDE)

# LINK PATH1 AND PATH2 TOGETHER IN THE DEPENDENCY GRAPH WITH
# EDGES OF TYPE SIDE.

INCLUDE MBDCOL                  # USE EMBED'S VARIABLES

INTEGER PATH1, PATH2, SIDE       # LOCAL TEMPORARY

INCLUDE MBCOM

DO 1 = 1, 2     # DO EACH ADJACENCY LIST
  $(
  IF (I == 2)
    $(
      # DO OPPOSITE EDGE
      DPTMP = PATH1
      PATH1 = PATH2
      PATH2 = DPTMP
    )

  # PICK AN EMPTY NODE FROM POOL
  DPTMP = APOOL
  APOOL = ALIST(DPTMP, FP)
  # FILL WITH GOODIES
  ALIST(DPTMP, EDGE1) = PATH1
  ALIST(DPTMP, EDGE2) = PATH2
  ALIST(DPTMP, TYPE) = SIDE
  # ADD TO PATH1'S ADJACENCY LIST
  ALIST(DPTMP, FP) = PHEAD(PATH1)
  PHEAD(PATH1) = DPTMP
  $(

RETURN
END
SUBROUTINE COLOR

COLOR THE DEPENDENCY GRAPH PRODUCED BY EMBED.

THE DEPENDENCY GRAPH CONTAINS EDGES THAT LINK PATHS FOUND BY EMBED AS NEEDING TO BE EITHER ON THE SAME SIDE OR OPPOSITE SIDES OF SOME CYCLE. COLOR WILL ASSIGN EACH PATH AS BEING EMBEDDED ON THE LEFT OR ON THE RIGHT OR LEFT AS HAVING NO DEPENDENCIES.

NOTE THAT THE DEPENDENCY GRAPH IS NOT NECESSARILY CONNECTED.

INCLUDE MBDCDL

# USE EMBED'S VARIABLES

INTEGER PATH1, PATH2, IPATH

INCLUDE MBCOM

MACRO(RECUR,1000) # RECURSION ENTRY POINT
MACRO(RETRN,1001) # RECURSION RETURN POINT

RSPNT = EMPTY # START AT TOP LEVEL OF RECURSION

DO IPATH = 1, PATH # SUBROUTINE EMBED LEFT PATH EQUAL TO MAXIMUM USED,

IF (PHEAD(IPATH) /= EMPTY) # COLOR A CONNECTED COMPONENT OF THE DEPENDENCY GRAPH
CALL DFS(IPATH)
PATH1 = IPATH
PSIDE(PATH1) = LEFT

# DFS (PATH1)
RECUR
NXTPNT = PHEAD(PATH1)
WHILE (NXTPNT /= EMPTY)

POINT = NXTPNT
NXTPNT = ALIST(POINT, FP)
PATH2 = ALIST(POINT, EDGE2)
IF (PSIDE(PATH2) == EMPTY) # PATH2 HASN'T BEEN SEEN BEFORE
IF (ALIST(POINT, TYPE) == SAME)
PSIDE(PATH2) = PSIDE(PATH1)
ELSE # OPPOSITE
IF (PSIDE(PATH1) == LEFT)
PSIDE(PATH2) = RIGHT
ELSE
PSIDE(PATH2) = LEFT
END
# PROCEED WITH DEPTH FIRST SEARCH
CALL DFS(PATH2)
RSPNT = RSPNT + 1
RSTK(RSPNT, 1) = PATH1
RSTK(RSPNT, 2) = NXTPNT
RSTK(RSPNT, 3) = POINT
PATH1 = PATH2
GO TO RECUR

RETRN # RETURNED FROM RECURSION
PATH1 = RSTK(RSPNT, 1)
NXTPNT = RSTK(RSPNT, 2)
POINT = RSTK(RSPNT, 3)
DONE WITH PATH1'S ADJACENCY LIST, SO PUT BACK IN EMPTY POOL.
IF (PHEAD(PATH1) ≠ EMPTY)
$\textbf{ALIST(POINT, FP)} = \textbf{APOOL}\\
\textbf{APOOL} = \textbf{PHEAD(PATH1)}\\
\textbf{PHEAD(PATH1)} = \textbf{EMPTY}$
# CHECK FOR RECURATION AND RETURN TO RIGHT PLACE
IF (RSPNT ≠ EMPTY)
GO TO RETRN
# END OF DFS
$
\textbf{RSPNT} = \textbf{RSPNT} - 1$

# END OF DEPENDENCY GRAPH
RETURN
END
SUBROUTINE ORDRE

# SORT THE ADJACENCY LISTS IN ORDER OF DECREASING PATH NUMBER.
# USE PHEAD AS THE BUCKET HEADERS FOR A RADIX SORT.
# USE EMBED'S VARIABLES

INTEGER OLDPNT

DO I = 1, MAX-VERTEX   # EACH ADJACENCY LIST
   NXTPNT = A(I, FP)   # START AT HEADER
   WHILE (NXTPNT /= EMPTY)
      POINT = NXTPNT   # STEP THROUGH ADJACENCY LIST
      NXTPNT = ALIST(POINT, FP)   # REMOVE NODE FROM I'S LIST
      A(I, FP) = ALIST(POINT, FP)   # PLACE IN PATH NUMBER'S BUCKET
      TEMP = ALIST(POINT, PNUM)   # SINCE EMBTDED BY LOOP
      ALIST(POINT, FP) = PHEAD(TEMP)   # TEMP
      PHEAD(TEMP) = POINT
   $)
   A(I, BP) = 0   # SINCE EMBTDED BY LOOP
$)
# ALL NODES ARE NOW LINKED TO BUCKETS - A'S ARE EMPTY
DO I = 1, PATH
   TEMP = PATH - I + 1   # WORK THE PATHS IN REVERSE ORDER
   WHILE (PHEAD(TEMP) /= EMPTY)
      $)
      POINT = PHEAD(TEMP)
      PHEAD(TEMP) = ALIST(POINT, FP)   # LINK TO PROPER ADJACENCY LIST
      V = ALIST(POINT, VRTX1)
      OLDPNT = A(V, BP)
      A(V, BP) = POINT
      ALIST(POINT, FP) = 0
      IF (OLDPNT == 0)
      A(V, FP) = POINT
      ELSE
      ALIST(OLDPNT, FP) = POINT
   $)
RETURN
END
SUBROUTINE DRGPH

INCLUDE GRDEF  # GRAPHICS DEFINITIONS
INCLUDE ABDCL  # VARIABLES FOR ADRW1 AND ADRW2
INCLUDE PRDCL  # PROCESS INFORMATION
INCLUDE FWDCCL # FLOWSHEET DISPLAY FILE
INCLUDE ABCCOM
INCLUDE PRCCOM
INCLUDE FWCCOM

# DRAW ABSTRACT GRAPH

FWDSP(1) = 0  # START NEW FLOWSHEET
CALL PRAMTR (OFFSET, OFF, FWDSP)
CALL ADRW1    # POSITION VERTICES
CALL ADRW2    # DO THE DRAWING
FX = BX(1)
FY = BY(1)
RETURN
END
# FILE NAME ABDCL RAT
# VARIABLE DECLARATIONS FOR ADRW1

INCLUDE MBDCL      # EMBED'S VARIABLES

MACRO(MAX=BRANCH,23)  # MAKE MAX-VERTEX/2
MACRO(XSIZE,40)
MACRO(YSIZE,40)
MACRO(SPACEx10)

INTEGER BSTRT       # FLAG: 0 => START NEW BRANCH
INTEGER STRTV       # START VERTEX OF CURRENT BRANCH
INTEGER BSIDE(MAX=BRANCH)  # DIRECTION OF EACH BRANCH
INTEGER BFIRST(MAX=BRANCH)  # FIRST VERTEX OF BRANCH
INTEGER BNUM(MAX=VERTEX)  # BRANCH NUMBER OF EACH VERTEX
INTEGER POS(MAX=VERTEX,2)  # X AND Y COORDINATES OF EACH VERTEX
INTEGER BX(MAX=BRANCH), BY(MAX=BRANCH)  # X AND Y DIMENSIONS OF EACH BRANCH
INTEGER BRSTK(MAX=BRANCH,5)  # INFO ON NESTED BRANCHES
MACRO(BRNUMBER,1)
MACRO(L-HALF,2)
MACRO(R-HALF,3)
MACRO(T-HALF,4)
MACRO(B-HALF,5)
INTEGER TMPB       # ADDITIONAL TEMPORARY REGISTER
INTEGER OLDPNT

# FILE NAME ABCOM RAT
# COMMON FOR ADRW1

INCLUDE MBCOM          # FROM EMBED

COMMON /ABCOM/ BNUM, POS, BSIDE, BX, BY, STRTV, BFIRST
SUBROUTINE ADRW1

# PASS 1 OF A SERIES OF DEPTH-FIRST-SEARCHES TO PRODUCE A DRAWING
# OF THE ABSTRACT GRAPH SPECIFIED BY ALIST.

INCLUDE ABOCL
INTEGER TMPOS
INCLUDE ABCOM

MACRO(RECUR,1000)
MACRO(RETN,1001)
MACRO(X,1)
MACRO(Y,2)
RSPNT = EMPTY
RSTKP = EMPTY
RSTRT = 1
START = n
RSIDF(1) = LEFT
RFIRST(1) = 1
X(1) = XSIZE
Y(1) = YSIZE
STRTV = 1
V = 1
POS(1, X) = 0
POS(1, Y) = 0

# CALL DFS(1)
V = 1
RNUM(1) = 1

# DFS(V)
RECUR
NXTPN = A(V, FP)
WHILE (NXTPN \= EMPTY)
  
  POINT = NXTPN
  NXTPN = ALIST(POINT, FP)
  W = ALIST(POINT, VRTX2)
  IF (V < W)
  # (V, W) IS AN ARC OF THE SPANNING TREE
  IF (START == 0)
  # START NEW PATH
  START = V
  IF (BSTRT == 0)
  # START NEW BRANCH
  BSTKP = RSTKP + 1
  BRSTK(BSTKP, BR-NUMB) = BNUM(V)
  BRSTK(BSTKP, L-HALF) = POS(V, X) + POS(STRTV, X)
  BRSTK(BSTKP, R-HALF) = POS(V, Y) - POS(STRTV, Y)
  BSTRT = W
  N = N * 1
  BFIRST(N) = W
  IF (PSIDE(ALIST(OLDPNT, PNUM)) == LEFT)
  BSIDE(N) = RIGHT
  ELSE
  BSIDE(N) = LEFT
  POS(W, X) = XSIZE + SPACE
  POS(W, Y) = 0
  BX(W) = XSIZE + SPACE + XSIZE
  BY(W) = YSIZE
ELSE
#
# CONTINUE DOWN PRESENT BRANCH
POS(W, X) = XSIZE + SPACE
POS(W, Y) = 0
IF (V <= STRTV)
$
POS(W, X) = POS(W, X) + POS(V, X)
POS(W, Y) = POS(W, Y) + POS(V, Y)
$
BX(N) = BX(N) + SPACE + XSIZE
$
BNUM(W) = N
#
# PROCEED WITH DEPTH-FIRST-SEARCH
# CALL DFS(W)
RSPNT = RSPNT + 1
RSTK(RSPNT, 1) = NXTPNT
RSTK(RSPNT, 2) = POINT
RSTK(RSPNT, 3) = V
RSTK(RSPNT, 4) = W
V = W
GO TO RECUR
RETRN
#
# RETURNED FROM RECURSION
NXTPNT = RSTK(RSPNT, 1)
OLDPNT = RSTK(RSPNT, 2)
V = RSTK(RSPNT, 3)
W = RSTK(RSPNT, 4)
RSPNT = RSPNT - 1
IF (BNUM(V) <= BNUM(W))
$
# JUST BACKED UP THROUGH BRANCH BOUNDARY
STRTV = BFIRST(BNUM(V))
IF (BSIDE(BNUM(W)) == LEFT)
POS(W, Y) = BRSTK(BSTKP, T-HALF) + SPACE + POS(W, Y)
ELSE # (BRANCH GOES RIGHT)
$
POS(W, Y) = -(BY(BNUM(W)) - POS(W, Y))
POS(W, Y) = POS(W, Y) + BRSTK(BSTKP, B-HALF) - SPACE
POS(STRTV, Y) = POS(STRTV, Y) + SPACE + BY(BNUM(W))
$
POS(W, X) = POS(W, X) + POS(V, X)
POS(W, Y) = POS(W, Y) + POS(V, Y)
BY(BNUM(V)) = BY(BNUM(V)) + SPACE + BY(BNUM(W))
TEMP = POS(V, X) + POS(STRTV, X)
TEMP = BX(BNUM(V)) - TEMP
TEMP = BX(BNUM(W)) - TEMP
IF (TEMP > 0)
BSTKP = BSTKP - 1
$
BSTRT = 0
# READY FOR NEW BRANCH
IF (START == V)
# HACKING DOWN DEAD END PATH
START = 0
$
ELSE
#
# (W <= V)
$
# (V, W) IS A FROND OF THE PALM TREE
# IF THIS FROND HAS NO SIDE ASSIGNED YET, ASSIGN NOW
IF (PSIDE(ALIST(POINT, PNUM)) == EMPTY)
$
BSTRT == 0
$

# AVOID ENCIRCLING OWN BRANCH
IF (PSIDE(ALIST(OLDPNT, PNUM)) == RIGHT)
    PSIDE(ALIST(POINT, PNUM)) = LEFT
ELSE
    PSIDE(ALIST(POINT, PNUM)) = RIGHT
$
ELSE IF (BNUM(V) == BNUM(W))$
    IF (BSIDE(BNUM(V)) == RIGHT)
        PSIDE(ALIST(POINT, PNUM)) = LEFT
    ELSE
        PSIDE(ALIST(POINT, PNUM)) = RIGHT
$
ELSE
    PSIDE(ALIST(POINT, PNUM)) = BSIDE(BNUM(V))$
$)
BY(BNUM(W)) = BY(BNUM(W)) * SPACE
IF (PSIDE(ALIST(POINT, PNUM)) == RIGHT)$
# SHIFT START VERTEX OF BRANCH UPWARD
    TEMP = BFIRST(BNUM(W))
    POS(TEMP, Y) = POS(TEMP, Y) + SPACE$
#
# NEXT, TRACE BACK THROUGH BRANCH STACK TO ACCUMULATE
# SIZE IN X DIMENSION.
I = BRSTK$# CURRENT BRANCH STACK POINTER
TMPB = BNUM(V)
TMPOS = POS(V, X)
IF (V \= STRTV)$
    TMPOS = TMPOS + POS(STRTV, X)$
    WHILE ( (I \= EMPTY) & (BRSTK(I, BR-NUMB) \= BNUM(W)))$
        IF (PSIDE(ALIST(POINT, PNUM)) \= BSIDE(TMPB))$
# MUST GO AROUND
            IF (BRSTK(I, R-HALF) \= TMPOS)$
                # EXTEND CURRENT BRANCH
                    TEMP = BRSTK(I, R-HALF) - TMPOS * SPACE
                    POS(STRTV, X) = POS(STRTV, X) + TEMP
                    BX(BNUM(V)) = BX(BNUM(V)) + TEMP
                    TMPOS = TMPOS + TEMP$
            TMPOS = TMPOS + BRSTK(I, L-HALF)
            I = I - 1$
        $)
START = 0 # START NEW PATH$
$)
IF (RSPNT \= EMPTY)$
    GO TO RETRN$
RETURN
END
SUBROUTINE ADREW2

# DRAW THE ABSTRACT GRAPH SPECIFIED BY ALIST USING THE POSITIONAL # INFORMATION PRODUCED BY ADREW1.

INCLUDE ABDCL

INTEGER BASE(2)
INTEGER MINY(MAX*BRANCH), MAXY(MAX*BRANCH)

INCLUDE ABCOM

MACRO(RECUR,1000)
MACRO(RETRN,1001)
MACRO(X,1)
MACRO(Y,2)

RSPNT = EMPTY
RSTART = 1
START = 0

# CENTER GRAPH AND DRAW FIRST VERTEX
BASE(X) = POS(1, X) + (1000 - BX(1)) / 2
BASE(Y) = POS(1, Y) + 33
POS(1, X) = BASE(X)
POS(1, Y) = BASE(Y)
CALL PNT (POS(1, X), POS(1, Y))
CALL ABVRT (1)
MINY(1) = POS(1, Y)
MAXY(1) = POS(1, Y) + YSIZE

# CALL DFS (1)
V = 1

# DFS (V)
RECUR
NXTPNT = A(V, FP)
WHILE (NXTPNT \= EMPTY)
  |
  POINT = NXTPNT
  NXTPNT = ALIST(POINT, FP)
  W = ALIST(POINT, VRTX2)
  IF (V < W)
    |
    # (V, W) IS AN ARC OF THE SPANNING TREE
    IF (START == 0)
      # START NEW PATH
      START = V
      POS(W, X) = POS(W, X) + BASE(X)
      POS(W, Y) = POS(W, Y) + BASE(Y)
      CALL PNT (POS(W, X), POS(W, Y))
      CALL ABVRT (W)
      |
      # DRAW CONNECTING LINE
      IF (POS(V, Y) == POS(W, Y))
        |
        CALL PNT ((POS(V, X) + XSIZE),
                   (POS(V, Y) + YSIZE/2))
        CALL LINE ((POS(W, X) - POS(V, X) - XSIZE), 0)
      |
      ELSE # DIFFERENT LEVELS
      |
      TEMP = POS(V, Y)
      IF (POS(W, Y) \> POS(V, Y))
        TEMP = TEMP + YSIZE
        CALL PNT ((POS(V, X) + XSIZE - XSIZE/4), TEMP)
\[ \text{TEMP} = \text{YSIZE} / 2 \]
\[ \text{IF (POS(W, Y) < POS(V, Y))} \]
\[ \text{TEMP} = -\text{TEMP} \]
\[ \text{CALL LINE (0, POS(W, Y) - POS(V, Y) - TEMP)} \]
\[ \text{CALL LINE (POS(W, X) - POS(V, X) - (XSIZE - XSIZE/4), 0)} \]
\[ \text{IF (BSTRT == 0)} \]
\[ \text{START NEW BRANCH} \]
\[ \text{BSTRT = W} \]
\[ \text{BASE(X) = POS(W, X)} \]
\[ \text{BASE(Y) = POS(W, Y)} \]
\[ \text{MINY(BNUM(W)) = POS(W, Y)} \]
\[ \text{MAXY(BNUM(W)) = POS(W, Y) + YSIZE} \]
\[ \text{# CALL DFS (W)} \]
\[ \text{RSPNT = RSPNT + 1} \]
\[ \text{RSTK(RSPNT, 1) = NXTPNT} \]
\[ \text{RSTK(RSPNT, 2) = V} \]
\[ \text{RSTK(RSPNT, 3) = W} \]
\[ \text{V = W} \]
\[ \text{GO TO RECUR} \]
\[ \text{RETNR} \]
\[ \text{NXTPNT = RSTK(RSPNT, 1)} \]
\[ \text{V = RSTK(RSPNT, 2)} \]
\[ \text{W = RSTK(RSPNT, 3)} \]
\[ \text{RSPNT = RSPNT - 1} \]
\[ \text{IF (BNUM(V) <= BNUM(W))} \]
\[ \text{# JUST BACKED UP THROUGH BRANCH BOUNDARY} \]
\[ \text{STRTV = BFIRST(BNUM(V))} \]
\[ \text{BASE(X) = POS(STRTV, X)} \]
\[ \text{BASE(Y) = POS(STRTV, Y)} \]
\[ \text{IF (MAXY(BNUM(V)) > MAXY(BNUM(W)))} \]
\[ \text{MAXY(BNUM(V)) = MAXY(BNUM(W))} \]
\[ \text{IF (MINY(BNUM(V)) < MINY(BNUM(W)))} \]
\[ \text{MINY(BNUM(V)) = MINY(BNUM(W))} \]
\[ \text{BSTRT = 0} \]
\[ \text{# READY TO START NEXT BRANCH} \]
\[ \text{IF (START == V)} \]
\[ \text{# BACKING UP A DEAD END PATH} \]
\[ \text{START = 0} \]
\[ \text{ELSE} \]
\[ \text{# (V, W) IS A FRONT OF THE PALM TREE} \]
\[ \text{START = 0} \]
\[ \text{IF (PSIDE(ALIST(POINT, PNUM)) == LEFT)} \]
\[ \text{CALL PNT ((POS(V, X) + XSIZE/4), (POS(V, Y) + YSIZE))} \]
\[ \text{# MUST GO AROUND ALL INTERMEDIATE BRANCHES} \]
\[ \text{I = BNUM(V)} \]
\[ \text{WHILE (I > BNUM(W))} \]
\[ \text{IF (MAXY(BNUM(W)) < MAXY(I))} \]
\[ \text{MAXY(BNUM(W)) = MAXY(I)} \]
\[ \text{I = I - 1} \]
\[ \text{# MAXY(BNUM(W)) = MAXY(BNUM(W)) + SPACE} \]
\[ \text{CALL LINE (0, MAXY(BNUM(W)) - (POS(Y, Y) + YSIZE))} \]
\[ \text{CALL LINE ((POS(W, X) - POS(V, X) + XSIZE/2), 0)} \]
\[ \text{CALL LINE (0, -(MAXY(BNUM(W)) - (POS(W, Y) + YSIZE)))} \]
\[ \text{ELSE} \]
\[ \text{# RIGHT} \]
\[ \text{CALL PNT ((POS(V, X) + XSIZE/4), POS(V, Y))} \]
# MUST GO AROUND ALL INTERMEDIATE BRANCHES
I = BNUM(V)
WHILE (I > BNUM(W))
  $(
    IF (MINY(BNUM(W)) > MINY(I))
      MINY(BNUM(W)) = MINY(I)
      I = I - 1
  )$
MINY(BNUM(W)) = MINY(BNUM(W)) - SPACE
CALL LINE (Ø, -(POS(V, Y) - MINY(BNUM(W))))
CALL LINE ((POS(W, X) - POS(V, X) * XSIZE/2), Ø)
CALL LINE (Ø, (POS(W, Y) - MINY(BNUM(W))))
$
)
IF (RSPNT \= EMPTY)
  GO TO RETRNN
RETURN
END
SUBROUTINE PNT (X, Y)

  # PROVIDED AS A SYNONYM FOR POINT FOR PROGRAMS THAT USE
  # POINT AS A VARIABLE NAME.

  INTEGER X, Y

  CALL POINT (X, Y)
  RETURN
END
SUBROUTINE ABVR T (N)

# DRAW A BOX WITH THE NUMBER N IN THE MIDDLE.
# START AND FINISH AT THE LOWER LEFT HAND CORNER,

REAL DIGITS(10), BLANK
INTEGER TENS, UNITS

DATA DIGITS /'0', '1', '2', '3', '4', '5', '6', '7', '8', '9' /
DATA BLANK /' '/

# DRAW BOX
CALL LINE (40, 0)
CALL LINE (0, 40)
CALL LINE (-40, 0)
CALL LINE (0, -40)
CALL LINE (7, 10, 0)

UNITS = N
TENS = 0
WHILE (UNITS .ge. 10)
  TENS = TENS + 1
  UNITS = UNITS - 10
ENDWHILE

IF (TENS .ge. 0)
  CALL TEXT (DIGITS(TENS + 1), 1)
ELSE
  CALL TEXT (BLANK, 1)
END IF

CALL TEXT (DIGITS(UNITS + 1), 1)

CALL LINE (-35, -10, 0)
RETURN
END
SUBROUTINE DRFLW

# DRAW FLOWSHEET

INCLUDE GRDEF      # GRAPHICS MNEUMONICS
INCLUDE FWDOC      # FLOWSHEET DISPLAY FILE
INCLUDE FDDCL      # COMMON FOR OVERLAYERED SUBPROGRAMS
INCLUDE FWCOM
INCLUDE FDCOM

FWDSP(1) = 0     # NEW FLOWSHEET
CALL PRAMTR (OFFSET, OFF, FWDSP)
CALL FDRWØ       # ASSIGN HANDS
CALL FDRW1       # POSITION UNITS
CALL FDRW2       # DO THE DRAWING
RETURN
END
# FILE NAME FDDCL
# VARIABLES FOR FLOWSHEET DRAWING

INCLUDE MCDCL  # EMBED'S VARIABLES
INCLUDE UPDCL  # SYMBOL INFO.
INCLUDE PROCCL  # PROCESS TOPOLOGY
INCLUDE FLOCL  # RELATE PROCOM TO MBCOM

MACRO(MAX*BRANCH,8)  # MAKE MAX*UNIT / 2
MACRO(SPACE,15)
MACRO(THIRD,5)    # MAKE SPACE / 3
MACRO(RECUR,1000)
MACRO(RECUR,1001)

INTEGER BSTRT  # FLAG: 0 => START NEW BRANCH
INTEGER USTRT  # FIRST UNIT OF BRANCH
INTEGER BSIDE(MAX*BRANCH)  # DIRECTION OF EACH BRANCH
INTEGER BFIRST(MAX*BRANCH)  # FIRST UNIT OF EACH BRANCH
INTEGER BNUM(MAX*UNIT)  # BRANCH NUMBER OF EACH UNIT
INTEGER POS(MAX*UNIT,2)  # X, Y COORDINATES OF EACH UNIT
INTEGER BX(MAX*BRANCH), BY(MAX*BRANCH)
# X, Y DIMENSIONS OF EACH BRANCH
INTEGER BRSTK(MAX*BRANCH,5)  # INFO. ON NESTED BRANCHES
MACRO(BR+NUMB,1)
MACRO(L+HALF,2)
MACRO(R+HALF,3)
MACRO(T+HALF,4)
MACRO(B+HALF,5)

INTEGER TMPP, THPOS  # TEMPORARIES
INTEGER UV, UW  # UNIT NUMBERS OF VERTICES V, W
INTEGER OLPNT
INTEGER N  # FOR ASSIGNING BRANCH NUMBERS
INTEGER UTYPE(MAX*UNIT)  # FOR SPIRALING CONSIDERATIONS

# FILE NAME FDCOM
# VARIABLES FOR FLOWSHEET DRAWING

INCLUDE MCOM  # MCOM IS JUST MBCOM WITH X AND Y REPLACED BY
# XDUM AND XDUM. THIS IS ONLY NECESSARY BECAUSE
# EMBED USES X AND Y AS VARIABLES AND THEY ARE
# MACRO'ED IN UPDCL. ELIMINATE THIS ON REWRITE!

INCLUDE UPCOM
INCLUDE PROCOM
INCLUDE FCOM
COMMON /FDCOM/ BSTRT, USTRT, BSIDE, BFIRST, BNUM, POS, BX, BY, BRSTK, UV, UW, N, UTYPE
SUBROUTINE FDRW0

# ASSIGN SIDES TO ALL FRONDS AND 'HANDS' TO ALL UNITS.
# PASS 1 OF 3 DEPTH-FIRST-SEARCHES TO DRAW THE FLOWSHEET.

INCLUDE FDCL      # VARIABLES IN COMMON WITH OTHER PASSES
    INTEGER S1, S2
    INTEGER TSIDE(MAX*UNIT)
    INTEGER SYMBW, SW, DIRW
    INCLUDE FDCOM

RSPNT = EMPTY    # TOP LEVEL OF RECURSION
    N = 1         # FIRST BRANCH
    V = 1         # START SEARCH AT VERTEX 1
    UV = FLWNE(UNNUM(V), 1)
    NNUM(UV) = N
    RSTRT = V
    USTRT = UV
    BSIDE(N) = LEFT

# CALL DFS (V)

# DFS (V)
RECUR
    UV = FLWNE(UNNUM(V), 1)
    NXTPNT = A(V, FP)
    WHILE (NXTPNT != EMPTY)
        %
        POINT = NXTPNT
        NXTPNT = ALIST(POINT, FP)
        W = ALIST(POINT, VRTX2)
        UW = FLWNE(UNNUM(W), 1)
        IF (UV == UW)
            %
                # RECORD WHICH WAY WE'RE GOING
                S1 = FLWNE(UNNUM(V), 2)
                S2 = FLWNE(UNNUM(W), 2)
                IF ((S2 - S1) == 1) + ((S1 == 4) & (S2 == 1))
                    TSIDE(UV) = RIGHT
                ELSE IF ((S1 - S2) == 1) + ((S1 == 1) & (S2 == 4))
                    TSIDE(UV) = LEFT
            %
        IF (V < W)
            %
                # (V, W) IS AN ARC
                IF (BSTRT == 0)
                    %
                        # NEW BRANCH
                        N = N + 1
                        IF (PSIDE(ALIST(OLDPNT, PNUM)) == LEFT)
                            BSIDE(N) = RIGHT
                        ELSE
                            BSIDE(N) = LEFT
                        %
                        BSTRT = W
            %
        NNUM(UW) = N
        # PROCEED WITH DEPTH-FIRST-SEARCH
        # CALL DFS (W)
        RSPNT = RSPNT + 1
        RSTK(RSPNT, 1) = NXTPNT
        RSTK(RSPNT, 2) = POINT
        RSTK(RSPNT, 3) = V
        RSTK(RSPNT, 4) = W
        V = W
        GO TO RECUR
RETRN
NXTPN = RSTK(RSPTN, 1)
OLDPN = RSTK(RSPTN, 2)
V = RSTK(RSPTN, 3)
W = RSTK(RSPTN, 4)
RSPTN = RSPTN - 1
UV = FLWNDE(UNUMB(V), 1)
UW = FLWNDE(UNUMB(W), 2)
BSTRT = 0
$)
ELSE
$(
# (V, W) IS A FROND
# IF NO SIDE ASSIGNED YET, DO IT NOW
IF (PSIDE(ALIST(POINT, PNUM)) == EMPTY)
$(
IF (BSTRT == 0)
$(
# AVOID ENCIRCLING OWN BRANCH
IF (PSIDE(ALIST(OLDPN, PNUM)) == RIGHT)
PSIDE(ALIST(POINT, PNUM)) = LEFT
ELSE
PSIDE(ALIST(POINT, PNUM)) = RIGHT
$
ELSE IF (BNUM(UV) == BNUM(UW))
$(
IF (BSIDE(BNUM(UV)) == RIGHT)
PSIDE(ALIST(POINT, PNUM)) = LEFT
ELSE
PSIDE(ALIST(POINT, PNUM)) = RIGHT
$
ELSE
PSIDE(ALIST(POINT, PNUM)) = BSIDE(BNUM(UV))
$
IF (UV == UW)
$(
# ASSIGN 'HAND' FOR GOOD
IF (PSIDE(ALIST(POINT, PNUM)) == RIGHT)
$(
IF (TSIDE(UV) == LEFT)
PRUNIT(UV, PDATA, HAND) = RIGHT
ELSE
PRUNIT(UV, PDATA, HAND) = LEFT
$
ELSE IF (PSIDE(ALIST(POINT, PNUM)) == LEFT)
PRUNIT(UV, PDATA, HAND) = TSIDE(UV)
$
$)
IF (RSPTN \= EMPTY)
GO TO RETRN
RETURN
END
SUBROUTINE FORW1

# PASS 2.0F 3 TO DRAW THE FLOWSHEET
# - POSITION THE UNITS

INCLUDE FDCCL  # VARIABLES IN COMMON WITH FORW2
INTEGER SYMBW, SW, DIRW
INCLUDE FDCOM

DO I = 1, MAX-UNIT
   BNUM(I) = 0
   RSPNT = EMPTY  # TOP LEVEL OF RECURSION
   BSTKP = EMPTY  # BRANCH STACK EMPTY
   N = 1  # FIRST BRANCH
   V = 1  # START DEPTH-FIRST-SEARCH ON VERTEX 1
   UV = FLWNDE(UNNUMB(V), 1)
   UTYPE(UV) = 0
   USTRT = UV
   RNUM(UV) = N
   RFIRST(N) = UV
   POS(UV, X) = 0
   POS(UV, Y) = 0
   BX(1) = USIZE(PRUNIT(UV, PDATA, SYMB), X)
   BY(1) = USIZE(PRUNIT(UV, PDATA, SYMB), Y)

   # CALL DFS(V)

   # DFS(V)
   RECUR
   UV = FLWNDE(UNNUMB(V), 1)
   NXTPNT = A(V, FP)
   WHILE (NXTPNT \= EMPTY)
      BEGIN
         POINT = NXTPNT
         NXTPNT = ALIST(POINT, FP)
         W = ALIST(POINT, VRTX2)
         UW = FLWNDE(UNNUMB(W), 1)
         IF (V \< W)
            BEGIN
               # (V, W) IS AN ARC
               IF (BNUM(UW) == 0)
                  BEGIN
                     # NEW UNIT TO POSITION
                     IF (PRUNIT(UW, PDATA, HAND) == EMPTY)
                        BEGIN
                           # ASSIGN HAND NOW
                           SYMBW = PRUNIT(UW, PDATA, SYMB)
                           SW = FLWNDE(UNNUMB(W), 2)
                           DIRW = USTRM(SYMBW, SW, STRDIR)
                           IF (DIRW == 1)
                              BEGIN
                                 PRUNIT(UW, PDATA, HAND) = LEFT
                              END ELSE
                              BEGIN
                                 PRUNIT(UW, PDATA, HAND) = RIGHT
                              END
                        END
                        IF (BSTRT == 0)
                           BEGIN
                              # START NEW BRANCH
                              # PUSH BRANCH INFO, ONTO STACK
                              BSTKP = BSTKP + 1
                              RRRSTK(BSTKP, BR+NUMB) = RNUM(UV)
                              RRRSTK(BSTKP, L+HALF) = POS(UV, X)+POS(USTRT, X)
                              TEMP = BX(RNUM(UV)) - RRRSTK(BSTKP, L+HALF)
                              RRRSTK(BSTKP, R+HALF) = TEMP
                           END
                  END
         END WHILE
   END
BRSTK(BSTKP, B-HALF) = POS(UV, Y) + POS(USTRAT, Y)
TEMP = BY(BNUM(UV)) - BRSTK(BSTKP, B-HALF)
BRSTK(BSTKP, T-HALF) = TEMP
USTRAT = UW
BSTRT = W
N = N + 1
BFIRST(N) = UW
CALL BRPOS
$
$
ELSE # CONTINUE DOWN SAME BRANCH
$(
CALL UNPOS
IF (UV \neq USTRAT)
$(
POS(UW, X) = POS(UW, X) + POS(UV, X)
POS(UW, Y) = POS(UW, Y) + POS(UV, Y)
$
$
BNUM(UW) = N
$
# PROCEED WITH DEPTH-FIRST-SEARCH
# CALL DFS (W)
RSPNT = RSPNT + 1
RSTK(RSPNT, 1) = NXTPNT
RSTK(RSPNT, 2) = POINT
RSTK(RSPNT, 3) = V
RSTK(RSPNT, 4) = W
V = W
GO TO RECUR

RETURN
# RETURNED FROM RECURSION
NXTPNT = RSTK(RSPNT, 1)
OLDPNT = RSTK(RSPNT, 2)
W = RSTK(RSPNT, 3)
RSPNT = RSPNT - 1
UV = FLWDE(UNNUMB(UV), 1)
UW = FLWDE(UNNUMB(UW), 1)
IF (BNUM(UV) \neq BNUM(UW))
$(
# JUST BACKED UP THROUGH A BRANCH BOUNDARY
USTRAT = BFIRST(BNUM(UV))
IF (BSIDE(BNUM(UV)) == LEFT)
$(
TEMP = BRSTK(BSTKP, T-HALF) + SPACE + POS(UW, Y)
POS(UW, Y) = TEMP
$
$
ELSE # BRANCH GOES RIGHT
$(
POS(UW, Y) = -(BY(BNUM(UW)) - POS(UW, Y))
POS(UW, Y) = POS(UW, Y) - BRSTK(BSTKP, B-HALF) - SPACE
POS(USTRAT, Y) = POS(USTRAT, Y) + SPACE + BY(BNUM(UW))
$
$
POS(UW, X) = POS(UW, X) + POS(UV, X)
POS(UW, Y) = POS(UW, Y) + POS(UV, Y)
BY(BNUM(UV)) = BY(BNUM(UV)) + SPACE + BY(BNUM(UW))
TEMP = POS(UV, X) + POS(USTRAT, X)
TEMP = BX(BNUM(UV)) - TEMP
TEMP = BX(BNUM(UW)) - TEMP
IF (TEMP \geq 0)
RSTK(BSTKP, 4) = BX(BNUM(UV)) + BX(BNUM(UW)) + TEMP
BSTKP = BSTKP - 1
$
$
BSTRT = \emptyset
# READY FOR NEW BRANCH
$)
ELSE

# (W <= V)
$
# (V, W) IS A FROND
BY(BNUM(UW)) = BY(BNUM(UW)) + SPACE
IF (PSIDE(ALIST(POINT, PNUM)) == RIGHT)
$(
# SHIFT START UNIT OF BRANCH UPWARD
TEMP = BFIRST(BNUM(UW))
POS(TEMP, Y) = POS(TEMP, Y) + SPACE
$
)
# NEXT, TRACE BACK THROUGH BRANCH STACK TO ACCUMULATE
# SIZE IN X DIMENSION.
I = BSTKP
# CURRENT STACK POINTER
TMPB = BNUM(UV)
TMPOS = POS(UV, X)
IF (UV != USTRT)

TMPOS = TMPOS + POS(USTRT, X)
WHILE ( (I \= EMPTY) & (BRSTK(I, BR+NUMB) >= BNUM(UW)) )
$(
IF (PSIDE(ALIST(POINT, PNUM)) \= BSIDE(TMPB))
# MUST GO AROUND

IF (BRSTK(I, R-HALF) > TMPOS)
$(
# EXTEND CURRENT BRANCH
TEMP = BRSTK(I, R-HALF) + TMPOS + SPACE
POS(USTRT, X) = POS(USTRT, X) + TEMP
BX(BNUM(UV)) = BX(BNUM(UV)) + TEMP
TMPOS = TMPOS + TEMP
$
)
TMPB = BRSTK(I, RR-NUMB)
TMPOS = TMPOS + BRSTK(I, L-HALF)
I = I - 1
$
)$

IF (RSPNT \= EMPTY)

GO TO RETRN
RETURN
END
SUBROUTINE BRPOS

# POSITION THE FIRST UNIT OF A NEW BRANCH

INCLUDE FDCOM

INTEGER SV, SW
INTEGER SYMBV, SYMBW
INTEGER DIRV, DIRW
INTEGER VX, WX
INTEGER Vi, V2
INTEGER UT, UDIR

SYMBV = PRUNIT(UV, PDATA, SYMB)
DIRV = USTRM(SYMBV, SV, STRDIR)
V1 = USTRM(SYMBV, SV, X)
V2 = USIZE(SYMBV, X) + THIRD

SYMBW = PRUNIT(UW, PDATA, SYMB)
DIRW = USTRM(SYMBW, SW, STRDIR)

# W FIRST
UDIR = DIRW
IF (PRUNIT(UW, PDATA, HAND) == LEFT)
$(
    IF (UDIR == 1)
        UDIR = 3
    ELSE IF (UDIR == 3)
        UDIR = 1
$(

UT = UTYPE(UV)
IF (UT = 5) & (UDIR == 2)
    VX = -THIRD
ELSE IF (UT = 6) & (UDIR == 2)
    VX = -THIRD
ELSE IF (UT = 6) & (UDIR == 4)
    VX = USIZE(SYMBV, X) + THIRD
ELSE IF (UT = 7) & (UDIR == 3)
    VX = USIZE(SYMBV, X) + 2 * THIRD
ELSE IF (UT = 7) & (UDIR == 4)
    VX = USIZE(SYMBW, X) + THIRD

# NOW V
UDIR = DIRV
IF (PRUNIT(UV, PDATA, HAND) == LEFT)
$(
    IF (UDIR == 1)
        UDIR = 3
    ELSE IF (UDIR == 3)
        UDIR = 1
$(

WX = USTRM(SYMBW, SW, X)
IF (PRUNIT(UW, PDATA, HAND) == LEFT)
    WX = USIZE(SYMBW, X) - WX
$)

SV = FLWNE(UNNUMB(V), 2)
SYMBV = PRUNIT(UV, PDATA, SYMB)
DIRV = USTRM(SYMBV, SV, STRDIR)
V1 = USTRM(SYMBV, SV, X)
V2 = USIZE(SYMBV, X) + THIRD

SW = FLWNE(UNNUMB(W), 2)
SYMBW = PRUNIT(UW, PDATA, SYMB)
DIRW = USTRM(SYMBW, SW, STRDIR)

# VARIABLES IN COMMON WITH FDRW'S
INTEGER SV, SW
# STREAM NUMBERS OF V AND W
INTEGER SYMBV, SYMBW
# SYMBOL TYPES OF UV AND UW
INTEGER DIRV, DIRW
# DIRECTIONS OF SV AND SW
INTEGER VX, WX
# X OFFSETS OF STREAMS FROM ORIGINS OF UNITS
INTEGER Vi, V2
# TWO POSSIBLE VALUES OF VX
ELSE IF ((UT == 8) & (UUST == 2))
  VX = USIZE(SYMBV, X) * THIRD
ELSE IF ((UT == 9) & (UDIR == 4))
  VX = USIZE(SYMBV, X) * THIRD
ELSE
  VX = USIZE(SYMBV, SV, X)
  IF (PRUNIT(UV, PDATA, HAND) == LEFT)
    VX = USIZE(SYMBV, X) - VX
ENDIF

RY(N) = USIZE(SYMBW, Y)
POS(UW, Y) = 0
POS(UW, X) = VX - WX
IF (POS(UW, X) < 0)
  POS(UW, X) = -POS(UW, X)
ENDIF
BX(N) = POS(UW, X) * USIZE(SYMBW, X)
RETURN
END
.UBROUTINE UNPOS
# POSITION NEXT UNIT IN A CONTINUING BRANCH

INCLUDE FDDCL
# VARIABLES IN COMMON WITH FORW'S
INTEGER SV, SW
# STREAM NUMBERS FOR V AND W
INTEGER SYMBV, SYMBW
# SYMBOLS FOR UNITS V AND W
INTEGER DIRV, DIRW
# DIRECTIONS FOR SV AND SW
INTEGER VX, W X
# X COORDINATES
INTEGER VY, WY
# Y COORDINATES
INTEGER UDIR
INCLUDE FDCOM

SV = FLWNLDE(UNNUMB(V), 2)
SW = FLWNLDE(UNNUMB(W), 2)
SYMBV = PRUNIT(UV, PDATA, SYMB)
SYMBW = PRUNIT(UW, PDATA, SYMB)
DIRV = USTRM(SYMBV, SV, STROIR)
DIRW = USTRM(SYMBW, SW, STROIR)

# ALIGN HORIZONTALLY
# W FIRST
UDIR = DIRW
IF (PRUNIT(UW, PDATA, HAND) == LEFT)
  $(
    IF (UDIR == 1)
      UDIR = 3
    ELSE IF (UDIR == 3)
      UDIR = 1
    $
  )
  UTYPE(UW) = UDIR
IF (UDIR == 1)
  WY = USIZE(SYMBW, Y) * THIRD
ELSE
  WY = USTRM(SYMBW, SW, Y)

# NOW V
UDIR = DIRV
IF (PRUNIT(UV, PDATA, HAND) == LEFT)
  $(
    IF (UDIR == 1)
      UDIR = 3
    ELSE IF (UDIR == 3)
      UDIR = 1
    $
  )
  IF ( (UTYPE(UV) == 1) & (UDIR == 2) )
    $(
      VY = -2 * THIRD
      UTYPE(UV) = 7
    )$
ELSE IF ( (UTYPE(UV) == 1) & (UDIR == 3) )
  $(
    VY = -THIRD
    UTYPE(UV) = 6
  )$
ELSE IF ( (UTYPE(UV) == 1) & (UDIR == 4) )
  $(
    VY = USTRM(SYMBV, SV, Y)
    UTYPE(UV) = 5
  )$
ELSE IF ( (UTYPE(UV) == 2) & (UDIR == 3) )
  $(
    VY = -THIRD
    UTYPE(UV) = 9
  )$
$\text{ELSE IF (U} \text{TYPE(UV) == 4) & (UDIR == 3)}$
$\text{VY = USIZE(SYMBV, Y) \& THIRD}
\text{U} \text{TYPE(UV) = 8}$
$\text{ELSE IF (U} \text{TYPE(UV) == 0)}$
$\text{IF (PRUNIT(UV, PDATA, HAND) == EMPTY)}$
$\text{IF (UDIR == 3)}$
$\text{PRUNIT(UV, PDATA, HAND) = LEFT}
\text{UDIR = 1}$
$\text{ELSE}$
$\text{PRUNIT(UV, PDATA, HAND) = RIGHT}$
$\text{IF (U} \text{TYPE(UV) == 0)}$
$\text{VY = THIRD}$
$\text{ELSE}$
$\text{VY = USTRM(SYMBV, SV, Y)}$
$\text{ELSE}$
$\text{VY = USTRM(SYMBV, SV, Y)}$

$\text{POS(UW, Y) = VY - WY \& RELATIVE TO POS(UV, Y)}$
$\text{TEMP = POS(USTRT, X)}$
$\text{IF (UV \neq USTRT)}$
$\text{TEMP = TEMP + POS(UV, X)}$
$\text{POS(UW, X) = BX(N) \& SPACE - TEMP}$
$\text{TEMP = POS(USTRT, Y) \& POS(UW, Y)}$
$\text{IF (UV \neq USTRT)}$
$\text{TEMP = TEMP + POS(UV, Y)}$
$\text{IF (TEMP < 0)}$
$\text{POS(USTRT, Y) = POS(USTRT, Y) - TEMP}
\text{BY(N) = BY(N) - TEMP}$
$\text{TEMP = TEMP + USIZE(SYMBW, Y)}$
$\text{IF (TEMP > BY(N))}$
$\text{BY(N) = TEMP}$
$\text{TEMP = POS(USTRT, X) \& POS(UW, X)}$
$\text{IF (UV \neq USTRT)}$
$\text{TEMP = TEMP + POS(UV, X)}$
$\text{IF (TEMP < 0)}$
$\text{POS(USTRT, X) = POS(USTRT, X) - TEMP}
\text{BX(N) = BX(N) - TEMP}$
$\text{TEMP = TEMP + USIZE(SYMBW, X)}$
$\text{IF (TEMP > BX(N))}$
$\text{BX(N) = TEMP}$

\text{RETURN}

\text{END}
SUBROUTINE FDRW2

# DRAW THE UNITS POSITIONED BY FDRW1

INCLUDE FDOCL
INTEGER BASE(2)  # BASE ADDRESS OF EACH BRANCH
INTEGER SYMBV, HANDV  # SYMBOL TYPE AND HAND OF VERTEX V'S UNIT
INTEGER SV, DIRV  # STREAM AND DIRECTION OF V
INTEGER SYMBOL, SW, DIRW  # SYMBOL, STREAM, AND DIRECTION OF W
INTEGER FSIDE, TX, TY  # TEMPORARIES
INTEGER XI, Y1, X2, Y2  # COORDINATES
INTEGER FSIDE, TX, TY, UI, UD, UDIR
INTEGER PXU, PYU, USY
INTEGER MINY(MAX*BRANCH), MAXY(MAX*BRANCH)
  # BOTTOM AND TOP OF EACH BRANCH
INCLUDE FDCOM

RSPNT = EMPTY
RSTART = 1
START = 0
N = 1
FX = BX(1)
FY = BY(1)

# CENTER FLOWSHEET AND DRAW FIRST UNIT
UV = BFIRST(1)
BNUM(UV) = 1
BASE(X) = POS(UV, X) + (1000 - BX(1)) / 2
BASE(Y) = POS(UV, Y) + 33
POS(UV, X) = BASE(X)
POS(UV, Y) = BASE(Y)
CALL PNT (POS(UV, X), POS(UV, Y))
PRUNIT(UV, COORD, XLOC) = POS(UV, X)
PRUNIT(UV, COORD, YLOC) = POS(UV, Y)
SYMBV = PRUNIT(UV, PDATA, SYMB)
IF (PRUNIT(UV, PDATA, HAND) == LEFT)
  HANDV = LUNIT
ELSE
  HANDV = RUNIT
CALL COPY (0, UDFILE(1, HANDV, SYMBV))
IX = TITLE(SYMBV, X)
IF (HANDV == LUNIT)
  IX = USIZE(SYMBV, X) - IX - 126
IY = TITLE(SYMBV, Y)
IX = IX + POS(UV, X)
IY = IY + POS(UV, Y)
CALL PNT (IX, IY)
CALL TEXT (PNAME(1, UV), 9)
MINY(1) = POS(UV, Y)
MAXY(1) = POS(UV, Y) + USIZE(SYMBV, Y)
DO I = 1, MAX+UNIT
  BNUM(UV) = I
V = 1
# CALL DFS (V)
# DFS (V)
RECUR
UV = FLWNE(UNUMB(V), 1)
NXTPNT = ATV(FP)
WHILE (NXTPNT \= EMPTY)
  POINT = NXTPNT
NXTPNT = ALIST(POINT, FP)
W = ALIST(POINT, VRTX2)
UW = FLWNDE(UNNUMB(W), 1)
IF (V < W)
  $$
  \text{# (V, W) IS A TREE ARC}
  \text{IF (BNUM(UW) == 0)}
  $$
  # NEW UNIT
  POS(UW, X) = POSS(UW, X) * BASE(X)
  POS(UW, Y) = POSS(UW, Y) * BASE(Y)
  CALL PNT (POS(UW, X), POS(UW, Y))
  PRUNIT(UW, COORD, XLOC) = POS(UW, X)
  PRUNIT(UW, COORD, YLOC) = POS(UW, Y)
  SYMBV = PRUNIT(UW, PO DATA, SYMB)
  IF (PRUNIT(UW, PO DATA, HAND) == LEFT)
    HANDV = LUNIT
  ELSE
    HANDV = RUNIT
  CALL COPY (0, UDFILE(1, HANDV, SYMBV))
  IX = UTITLEISYMBV, X)
  IF (HANDV == LUNIT)
    IX = USIZE(SYMBV, X) - IX - 126
  IY = UTITLEISYMBV, Y)
  IX = IX + POS(UW, X)
  IY = IY + POS(UW, Y)
  CALL PNT (IX, IY)
  CALL TEXT (PRNAME(1, UW), 9)
  IF (BSTRT == 0)
    $$
    \text{# NEW BRANCH}
    N = N + 1
    BSTRT = UW
    BASE(X) = POS(UW, X)
    BASE(Y) = POS(UW, Y)
    MINY(N) = POS(UW, Y)
    MAXY(N) = POS(UW, Y) + USIZE(SYMBV, Y)
    CALL BRD RW
    $$
  ELSE
    $$
    \text{IF ((POS(UW, Y) + USIZE(SYMBV, Y)) > MAXY(N))}
    \text{MAXY(N) = POS(UW, Y) + USIZE(SYMBV, Y)}
    \text{IF (POS(UW, Y) < MINY(N))}
    \text{MINY(N) = POS(UW, Y)}
    CALL UNRD W
    $$
  BNUM(UW) = N
  $$
# CALL DFS (W)
RSPNT = RSPNT + 1
RSTK(RSPNT, 1) = NXTPNT
RSTK(RSPNT, 2) = V
RSTK(RSPNT, 3) = W
V = W
GO TO RECUR
RET RAN
NXTPNT = RSTK(RSPNT, 1)
V = RSTK(RSPNT, 2)
W = RSTK(RSPNT, 3)
RSPNT = RSPNT - 1
UV = FLWNDE(UNNUMB(V), 1)
UW = FLWNDE(UNNUMB(W), 1)
IF (BNUM(UV) \&= BNUM(UW))
  $$
# JUST BACKED THROUGH BRANCH BOUNDARY
USTRT = BFIRST(BNUM(UV))
BASE(X) = POS(USTRT, X)
BASE(Y) = POS(USTRT, Y)
IF (MAXY(BNUM(UW)) > MAXY(BNUM(UV)))
    MAXY(BNUM(UV)) = MAXY(BNUM(UW))
IF (MINY(BNUM(UW)) < MINY(BNUM(UV)))
    MINY(BNUM(UV)) = MINY(BNUM(UW))
$)
BSTRT = 0  # READY FOR NEW BRANCH
$)
ELSE  # (V >= W)
$(
# (V, W) IS A FROND
IF (UV \= UW)
  $(
  # DRAW IT
  SV = FLWNEU(UNNUMB(V), 2)
  SW = FLWNEU(UNNUMB(W), 2)
  SYMBV = PRUNIT(UV, PDATA, SYMB)
  SYMBW = PRUNIT(UW, PDATA, SYMB)
  DIRV = USTRM(SYMBV, SV, STRDIR)
  DIRW = USTRM(SYMBW, SW, STRDIR)
  X1 = USTRM(SYMBV, SV, X)
  Y1 = USTRM(SYMBV, SV, Y)
  X2 = USTRM(SYMBW, SW, X)
  Y2 = USTRM(SYMBW, SW, Y)
  IF (PRUNIT(UV, PDATA, HAND) == LEFT)
    X1 = USIZE(SYMBV, X) - X1
    Y1 = X1 + POS(UV, X)
    X2 = USIZE(SYMBW, X) - X2
    Y2 = X2 + POS(UW, X)
  FSIDE = PSIDE(ALIST(POINT, PNUM))
  DO II = 1, 2
    $(
      IF (II == 1)
        $(
          # DO V
          UU = UV
          XT = X1
          YT = Y1
          UDIR = DIRV
          PUX = POS(UV, X)
          USX = USIZE(SYMBV, X)
          PUY = POS(UV, Y)
          USY = USIZE(SYMBV, Y)
        $)
      ELSE
        $(
          # DO W
          UU = UW
          XT = X2
          YT = Y2
          UDIR = DIRW
          PUX = POS(UW, X)
          USX = USIZE(SYMBW, X)
          PUY = POS(UW, Y)
          USY = USIZE(SYMBW, Y)
        $)
      U1 = UTYPE(UU)
    $)
$)
IF (PRUNIT(UU, PDATA, HAND) == LEFT)
   $(
      IF (UDIR == 3)
         UDIR = 1
      ELSE IF (UDIR == 1)
         UDIR = 3
   )
   TX = XT
   TY = YT
IF (((UT == 1) || (UT == 5) || (UT == 6) || (UT == 7)))
   $(
      IF ((UDIR == 2) && (FSIDE == LEFT))
      $(
         CALL PNT(XT, YT)
         TX = POS(UU, X) - THIRD
         CALL LINE(TX - XT, 0)
         XT = TX
         TX = PUX + USX + SPACE
         TY = PUY - 2 * THIRD
      )
      ELSE IF ((UDIR == 3) && (FSIDE == LEFT))
      $(
         TY = PUY - THIRD
         TX = PUX + USX + 2 * THIRD
      )
      ELSE IF ((UDIR == 4) && (FSIDE == LEFT))
      $(
         TX = PUX - THIRD
      )
      ELSE IF ((UT == 2) || (UT == 9))
      $(
         IF (((UDIR == 3) && (FSIDE == LEFT))
         $(
            TX = PUX + USX + 2 * THIRD
            TY = PUY - THIRD
         )
         ELSE IF (((UDIR == 4) && (FSIDE == LEFT)))
         TX = PUX + USX + THIRD
      )
      ELSE IF (UT == 3)
      $(
         IF ((UDIR == 2) && (FSIDE == RIGHT))
         TX = PUX + USX + THIRD
         ELSE IF (((UDIR == 4) && (FSIDE == LEFT)))
         TX = PUX + USX + THIRD
      )
      ELSE IF (UT == 4) || (UT == 8)
      $(
         IF ((UDIR == 2) && (FSIDE == RIGHT))
         $(
            TY = PUY + USY + THIRD
            TX = PUX + USX + 2 * THIRD
         )
         ELSE IF (((UDIR == 2) && (FSIDE == RIGHT)))
         TX = PUX + USX + THIRD
      )
      ELSE IF (UT == 10)
      IF (((UDIR == 2) && (FSIDE == RIGHT))
         + ((UDIR == 4) && (FSIDE == LEFT)))
      TX = PUX - THIRD
      ELSE IF ((UT == 11) && (FSIDE == LEFT))
      $(
         IF (UDIR == 4)
         TX = PUX - THIRD
         ELSE IF (UDIR == 1)
\$229\$

\$\text{TX} = \text{PUX} - 2 \cdot \text{THIRD} \\
\text{TY} = \text{PUY} - \text{THIRD} \$

\$\text{ELSE IF } ((\text{UT} == 13) \& (\text{FSIDE} == \text{RIGHT})) \$
\$\text{IF } (\text{UDIR} == 2) \$
\$\text{TX} = \text{PUX} - \text{THIRD} \\
\text{ELSE IF } (\text{UDIR} == 1) \$
\$\text{TX} = \text{PUX} - 2 \cdot \text{THIRD} \\
\text{TY} = \text{PUY} + \text{USY} + \text{THIRD} \$

\$\text{ELSE IF } (\text{UT} == 12) \$
\$\text{IF } ((\text{UDIR} == 4) \& (\text{FSIDE} == \text{LEFT})) \$
\$\text{TX} = \text{PUX} + \text{USX} + \text{THIRD} \\
\text{ELSE IF } ((\text{UDIR} == 4) \& (\text{FSIDE} == \text{RIGHT})) \$
\$\text{TX} = \text{PUX} + \text{USX} + \text{THIRD} \\
\text{CALL PNT (XT, YT)} \\
\text{CALL LINE (TX - XT, 0)} \\
\text{XT} = \text{TX} \\
\text{TX} = \text{PUX} - \text{SPACE} \\
\text{TY} = \text{PUY} + \text{USY} + 2 \cdot \text{THIRD} \$

\$\text{ELSE IF } ((\text{UDIR} == 1) \& (\text{FSIDE} == \text{RIGHT})) \$
\$\text{TX} = \text{PUX} - 2 \cdot \text{THIRD} \\
\text{TY} = \text{PUY} + \text{USY} + \text{THIRD} \$

\$\text{ELSE IF } ((\text{UDIR} == 2) \& (\text{FSIDE} == \text{RIGHT})) \$
\$\text{TX} = \text{PUX} - \text{THIRD} \$

\$\text{CALL PNT (XT, YT)} \\
\text{IF } (\text{TY} \leq \text{YT}) \$
\$\text{CALL LINE (0, TY - YT)} \\
\text{YT} = \text{TY} \$

\$\text{IF } (\text{TX} \leq \text{XT}) \$
\$\text{CALL LINE (TX - XT, 0)} \\
\text{XT} = \text{TX} \$

\$\text{IF } (\text{II} == 1) \$
\$\text{XI} = \text{XT} \\
\text{YI} = \text{YT} \$

\$\text{ELSE} \$
\$\text{X2} = \text{XT} \\
\text{Y2} = \text{YT} \$

\$\$

# GO AROUND INTERMEDIATE BRANCHES 
\text{IF } (\text{FSIDE} == \text{LEFT}) \$
\$\text{I} = \text{BNUM(UV)} \\
\text{WHILE } (\text{I} > \text{BNUM(UW)}) \$

\$\$
IF (MAXY(BNUM(UW)) < MAXY(I))
   MAXY(BNUM(UW)) = MAXY(I)
   I = I - 1

$\$
MAXY(BNUM(UW)) = MAXY(BNUM(UW)) + SPACE
CALL PNT (X2, Y2)
CALL LINE (0, MAXY(BNUM(UW)) - Y2)
CALL LINE (X1 - X2, 0)
CALL LINE (0, Y1 - MAXY(BNUM(UW)))
$\$
ELSE  \# RIGHT
$\$
   I = BNUM(UV)
WHILE (I > BNUM(UW))
   $\$
      IF (MINY(BNUM(UW)) > MINY(I))
      MINY(BNUM(UW)) = MINY(I)
      I = I - 1
   $\$
   MINY(BNUM(UW)) = MINY(BNUM(UW)) - SPACE
CALL PNT (X2, Y2)
CALL LINE (0, MINY(BNUM(UW)) - Y2)
CALL LINE (X1 - X2, 0)
CALL LINE (0, Y1 - MINY(BNUM(UW)))
$\$
$\$
IF (RSPNT \# EMPTY)
RETURN
GO TO RETRN
END
SUBROUTINE BRDRW

# CONNECT UW WITH UV WHEN UW IS THE FIRST UNIT OF A NEW BRANCH.

INCLUDE FDDCL       # SAME VARIABLES AS FDRW2
INTEGER SV, SW       # STREAM NUMBERS OF V AND W
INTEGER SYMBV, SYMBW # SYMBOL TYPES
INTEGER DIRV, DIRW  # DIRECTIONS
INTEGER X1, Y1, X2, Y2, TX, TY # COORDINATES
INTEGER UT, UDIR
INCLUDE FDCOM

SV = FLWNODE(UNNUMB(V), 2)
SW = FLWNODE(UNNUMB(W), 2)
SYMBV = PRUNIT(UV, PDATA, SYMB)
SYMBW = PRUNIT(UW, PDATA, SYMB)
DIRV = USTRM(SYMBV, SV, STRDIR)
DIRW = USTRM(SYMBW, SW, STRDIR)

# CALCULATE STREAM COORDINATES
X1 = USTRM(SYMBV, SV, X)
IF (PRUNIT(UV, PDATA, HAND) == LEFT)
    X1 = USIZE(SYMBV, X) - XI
Y1 = USTRM(SYMBV, SV, Y)
X1 = X1 + POS(UV, X)
Y1 = Y1 + POS(UV, Y)
X2 = USTRM(SYMBW, SW, X)
IF (PRUNIT(UW, PDATA, HAND) == LEFT)
    X2 = USIZE(SYMBW, X) - X2
Y2 = USTRM(SYMBW, SW, Y)
X2 = X2 + POS(UW, X)
Y2 = Y2 + POS(UW, Y)

# W FIRST
UDIR = DIRW
IF (PRUNIT(UW, PDATA, HAND) == LEFT)
    IF (UDIR == 1)
        UDIR = 3
    ELSE IF (UDIR == 3)
        UDIR = 1
    IF (UDIR == 1) & (BSIDE(N) == LEFT)
        TY = POS(UW, Y) + USIZE(SYMBW, Y) + THIRD
        TX = POS(UW, X) - 2 * THIRD
    ELSE IF (UDIR == 2) & (BSIDE(N) == LEFT)
        TX = POS(UW, X) - THIRD
        TY = Y2
    ELSE
        TX = X2
        TY = Y2
    CALL PNT (X2, Y2)
    IF (TY <= Y2)
        CALL LINE (0, TY - Y2)
        Y2 = TY
IF (TX ≠ X2)
$\!
\begin{align*}
&\text{CALL LINE (TX - X2, 0)} \\
&\text{X2 = TX} \\
&\text{S}
\end{align*}
$
#
\text{NOW V}
UDIR = DJIR
UT = UTYPE(UV)
IF (PRUNIT(UV, PDATA, HAND) == LEFT)
$\!
\begin{align*}
&\text{IF (UDIR == 1)} \\
&\quad\text{UDIR = 3} \\
&\text{ELSE IF (UDIR == 3)} \\
&\quad\text{UDIR = 1} \\
&\text{S}
\end{align*}
$
TX = XI
TY = Y1
IF (UT == 5) & (UDIR == 2)
$\!
\begin{align*}
&\text{TX = POS(UV, X) - THIRD} \\
&\text{S}
\end{align*}
$
ELSE IF (UT == 6) & (UDIR == 2)
$\!
\begin{align*}
&\text{TX = POS(UV, X) - THIRD} \\
&\text{S}
\end{align*}
$
ELSE IF (UT == 6) & (UDIR == 4)
$\!
\begin{align*}
&\text{TX = POS(UV, X) + USIZE(SYMBV, X) + THIRD} \\
&\text{S}
\end{align*}
$
ELSE IF (UT == 7) & (UDIR == 3)
$\!
\begin{align*}
&\text{TY = POS(UV, Y) - THIRD} \\
&\text{TX = POS(UV, X) + USIZE(SYMBV, X) + 2 * THIRD} \\
&\text{S}
\end{align*}
$
ELSE IF (UT == 7) & (UDIR == 4)
$\!
\begin{align*}
&\text{TX = POS(UV, X) + USIZE(SYMBV, X) + THIRD} \\
&\text{S}
\end{align*}
$
ELSE IF (UT == 8) & (UDIR == 2)
$\!
\begin{align*}
&\text{TX = POS(UV, X) + USIZE(SYMBV, X) + THIRD} \\
&\text{S}
\end{align*}
$
ELSE IF (UT == 9) & (UDIR == 4)
$\!
\begin{align*}
&\text{TX = POS(UV, X) + USIZE(SYMBV, X) + THIRD} \\
&\text{S}
\end{align*}
$
CALL PNT (XI, Y1)
IF (TY ≠ Y1)
$\!
\begin{align*}
&\text{CALL LINE (0, TY - Y1)} \\
&\text{Y1 = TY} \\
&\text{S}
\end{align*}
$
IF (TX ≠ X1)
$\!
\begin{align*}
&\text{CALL LINE (TX - X1, 0)} \\
&\text{X1 = TX} \\
&\text{S}
\end{align*}
$
CALL LINE (0, Y1 - Y2)
CALL LINE (X1 - X2, 0)
RETURN
END
SUBROUTINE UNDRW

# CONNECT UW WITH UV WHEN UW IS THE NEXT UNIT OF A CONTINUING BRANCH,

INCLUDE FDDCL    # SAME VARIABLES AS FDRW2
INTEGER SV, SH    # STREAM NUMBERS OF V AND W
INTEGER SYMBV, SYMBW # SYMBOL TYPES
INTEGER DIRV, DIRW # DIRECTIONS
INTEGER X1, Y1, X2, Y2, TX, TY # COORDINATES
INTEGER UT
INCLUDE FDCOM

SV = FLMNDE(UNNUMB(V), 2)
SW = FLMNDE(UNNUMB(W), 2)
SYMBV = PRUNIT(UV, PDATA, SYMB)
SYMBW = PRUNIT(UW, PDATA, SYMB)
DIRV = USTRM(SYMBV, SV, STRDIR)
DIRW = USTRM(SYMBW, SW, STRDIR)

# CALCULATE STREAM COORDINATES
X1 = USTRM(SYMBV, SV, X)
IF (PRUNIT(UV, PDATA, HAND) == LEFT)
   X1 = USIZE(SYMBV, X) - X1
Y1 = USTRM(SYMBV, SV, Y)
X1 = X1 + POS(UV, X)
Y1 = Y1 + POS(UV, Y)

X2 = USTRM(SYMBW, SW, X)
IF (PRUNIT(UW, PDATA, HAND) == LEFT)
   X2 = USIZE(SYMBW, X) - X2
Y2 = USTRM(SYMBW, SW, Y)
X2 = X2 + POS(UW, X)
Y2 = Y2 + POS(UW, Y)

UT = UTYPE(UW)
IF ( (UT == 1) + (UT == 5) + (UT == 6) + (UT == 7) )
   TY = POS(UW, Y) + USIZE(SYMBW, Y) + THIRD
ELSE
   TY = Y2
   CALL PNT (X2, Y2)
   IF (TY \= Y2)
      $(
         CALL LINE (0, TY - Y2)
         Y2 = TY
      )$

UT = UTYPE(UV)
TX = X1
TY = Y1
IF ( (UT == 6) + (UT == 9) + (UT == 12) )
   TY = POS(UV, Y) - THIRD
ELSE IF (UT == 7)
   $(
      TX = POS(UV, X) - THIRD
      TY = POS(UV, Y) - 2 * THIRD
   )$
ELSE IF (UT == 0)
   TY = POS(UV, Y) + USIZE(SYMBV, Y) + THIRD

CALL PNT (X1, Y1)
IF (TX \= X1)
   $(
      CALL LINE (TX - X1, 0)
      X1 = TX
   )$
IF (TY \n= Y1)
  \$
  CALL LINE (0, TY - Y1)
  Y1 = TY
  \$
  CALL POINT (X1, Y1)
  CALL LINE (X2 - X1, 0)
RETURN
END
SUBROUTINE VTSIM (MAIN, PTYPE)

# PRODUCE A HARD COPY, ON THE HP PLOTTER, OF WHAT'S ON THE VT-15 SCREEN.

INTEGER PC  # PROGRAM COUNTER
INTEGER MAIN  # DISPLAY FILE
INTEGER INST  # CURRENT INSTRUCTION REGISTER
INTEGER ADR  # ADDRESS REGISTER
INTEGER X, Y  # COORDINATE REGISTERS
INTEGER DIR  # DIRECTION REGISTER
INTEGER INT  # INTENSITY REGISTER
INTEGER TEMP  # NEW, IMPROVED TEMPORARY REGISTER
INTEGER CHAR1, CHAR2  # 5/7 PACKED ASCII REGISTERS
INTEGER CH(5)  # UNPACKED ASCII CHARACTERS.
INTEGER PTYPE  # 0 FOR HORIZONTAL AND 1 FOR VERTICAL PLOT

INCLUDE VTCOM

HORV = PTYPE  # PUT PLOT TYPE IN COMMON FOR VTPTL
CALL DCLOSE  # STOP THE VT-15 PROCESSOR
CALL GETADR (MAIN, PC)
IF (PC[0:10] == 1)
$\{
   # INDIRECT
   PC = PC[3:17]
   PC[0:2] = MAIN[0:12]
   CALL GETVAL (PC, PC)
   PC = PC[3:17]
$\}
PC = PC + 1  # SKIP THE DISPLAY FILE SIZE
CALL PU TVAL (PC, 0)  # THIS WILL SERVE AS A FLAG TO STOP THE SIMULATION
PC = PC + 1  # INITIALIZE PARAMETERS
SCALE = 0
ROTATE = 0
OFFSET = 0
REPEAT
$\{
   CALL GETVAL (PC, INST)  # FETCH
   PC = PC + 1
   # FLOW OF CONTROL INSTRUCTIONS
   IF (INST[0:12] == 6)
   $\{
      # DISPLAY JUMP
      ADR = INST[5:17]
      IF (INST[4:4] == 1)
      $\{
         # INDIRECT
         ADR[0:14] = PC[0:4]
         CALL GETVAL (ADR, ADR)
         ADR = ADR[3:17]  # DROP STATUS BITS
         IF (ADR == 0)
            BREAK  # THAT'S THE SIGNAL TO END
      $\}
      IF (INST[3:3] == 1)
      $\{
         # JMS
         CALL PUTVAL (ADR, PC)  # DEPOSIT RETURN ADDRESS
         PC = ADR + 1
      $\}
      ELSE  # (INST[3:3] == 0)
      $\{
      PC = ADR
   $\}$
$\}$
ELSE IF (INST[0:8] == ##235)
# DISPLAY SKIP
PC = PC + 1

# PARAMETER INSTRUCTIONS

ELSE IF (INST[0:5] == ##20)
$(
# PARAMETER 1
IF (INST[13:13] == 1)
SCALE = INST[14:17]
)$
ELSE IF (INST[0] == ##21)
$(
# PARAMETER 2
IF (INST[12:12] == 1)
ROTATE = INST[13:13]
IF (INST[16:16] == 1)
OFFSET = INST[17:17]
)$
ELSE IF (INST[0] == ##22)
$(
# SAVE
ADR = INST[5:17]
TEMP[0:13] = SCALE
TEMP[5:15] = ROTATE
TEMP[6:16] = OFFSET
CALL PUTVAL (ADR, TEMP)
)$
ELSE IF (INST[0] == ##23)
$(
# RESTORE
ADR = INST[5:17]
CALL GETVAL (ADR, TEMP)
SCALE = TEMP[0:13]
ROTATE = TEMP[5:5]
OFFSET = TEMP[6:6]
)$

# ABSOLUTE POSITIONING INSTRUCTIONS

ELSE IF (INST[0:3] == 3)
$(
# POINT/GRAPH PLOT
IF (INST[6:6] == 1)
$(
# X AXIS
X = INST[8:17]
IF (INST[7:7] == 1)
# GRAPH PLOT
Y = YPOS * SCALE
ELSE
# POINT PLOT
Y = YPOS
)$
ELSE
$(
# Y AXIS
Y = INST[8:17]
IF (INST[7:7] == 1)
# GRAPH PLOT
X = XPOS * SCALE
ELSE
# POINT PLOT
X = XPOS
)$
$)
CALL VTPLT (POINT, X, Y)
IF (INST[4:14] == 1)
  # INTENSIFY
  CALL VTPLT (DRAW, 0, 0)
$

# VECTOR INSTRUCTIONS
ELSE IF (INST[0:3] == 2)
$(
  # ARBITRARY VECTOR
  CALL GETVAL (PC, TEMP)
  PC = PC + 1
  X = INST[8:17]
  IF (INST[6:6] == 1)
    X = -X
  Y = TEMP[8:17]
  IF (TEMP[6:6] == 1)
    Y = -Y
  IF (INST[4:4] == 1)
    CALL VTPLT (DRAW, X, Y)
  ELSE
    CALL VTPLT (MOVE, X, Y)
$
ELSE IF (INST[0:3] == 6)
$(
  # ARBITRARY SHORT VECTOR
  X = INST[7:11]
  IF (INST[6:6] == 1)
    X = -X
  Y = INST[13:17]
  IF (INST[12:12] == 1)
    Y = -Y
  IF (INST[4:4] == 1)
    CALL VTPLT (DRAW, X, Y)
  ELSE
    CALL VTPLT (MOVE, X, Y)
$
ELSE IF (INST[0:2] == 4)
$(
  # BASIC VECTOR
  TEMP = INST[8:17]  # DATA
  DIR = INST[9:7]
  INT = INST[4:4]
  CALL BV ECS (TEMP, DIR, INT)
$
ELSE IF (INST[0:2] == 5)
$(
  # BASIC SHORT VECTOR
  TEMP = INST[8:10]  # DATA
  DIR = INST[5:7]
  INT = INST[4:4]
  CALL BV ECS (TEMP, DIR, INT)
  TEMP = INST[15:17]
  DIR = INST[12:14]
  INT = INST[11:11]
  CALL BV ECS (TEMP, DIR, INT)
$

# CHARACTER INSTRUCTIONS
ELSE IF (INST[0:3] == 0)
$(
  # CHARACTER INPUT
  TEMP = INST[10:17]
  CALL PCHAR (TEMP)
$
ELSE IF (INST[0:3] == 1)

ELSE IF (INST[4:17] == 1)

# INDIRECT
ADR[0:4] = PC[0:4]
CALL GETVAL (ADR, ADR)
ADR = ADR[3:17] # DROP STATUS BITS

REPEAT

# GET TWO WORDS OF 5/7 PACKED ASCII
CALL GETVAL (ADR, CHAR1)
ADR = ADR + 1
CALL GETVAL (ADR, CHAR2)
ADR = ADR + 1

# UNPACK
CH(1) = CHAR1[0:6]
CH(2) = CHAR1[7:13]
CH(3) = 0
CH(3)[11:14] = CHAR1[14:17]
CH(3)[15:17] = CHAR2[0:12]
CH(4) = CHAR2[3:9]
CH(5) = CHAR2[10:16]

DISPLAY UNTIL ALT MODE
DO I = 1, 5

$(
    TEMP = CH(I)
    IF (TEMP $= ##175) # ALT MODE
    CALL PCHAR (TEMP)
    ELSE
    BREAK
$(

UNTIL (TEMP == ##175)

CALL VTPUT (MOVE, 0, 0) # FORCE THE PEN UP
CALL DINIT (MAIN) # START THE DISPLAY BACK UP
RETURN
END
# FILE NAME: VTCOM RAT

MACRO(POINT,0)
MACRO(MOVE,1)
MACRO(DRAW,2)

INTEGER XPOS, YPOS  # CURRENT "BEAM" POSITION
INTEGER ROTATE, OFFSET, SCALE  # DISPLAY PARAMETERS
INTEGER HORV  # HORIZONTAL OR VERTICAL PLOT

COMMON /VTCOM/ XPOS, YPOS, ROTATE, OFFSET, SCALE, HORV
SUBROUTINE BVECST (DELTA, DIR, INT)

# DRAW BASIC VECTORS THAT COME FROM THE BASIC VECTOR, BASIC SHORT VECTOR
# AND CHARACTER INSTRUCTIONS,

INTEGER X, Y # COORDINATE REGISTERS
INTEGER DELTA # MAGNITUDE REGISTER
INTEGER DIR # DIRECTION REGISTER
INTEGER INT # INTENSITY REGISTER

INCLUDE VTCOM

IF (ROTATE == 1)
   DIR = MOD (DIR + 2, 8)

IF ( (DIR <= 1) & (DIR == 7) )
   X = DELTA
ELSE IF ( (DIR >= 3) & (DIR <= 5) )
   X = -DELTA
ELSE
   X = 0

IF ( (DIR >= 1) & (DIR <= 3) )
   Y = DELTA
ELSE IF ( (DIR >= 5) & (DIR <= 7) )
   Y = -DELTA
ELSE
   Y = 0

IF (INT == 1)
   CALL VTPLT (DRAW, X, Y)
ELSE
   CALL VTPLT (MOVE, X, Y)

RETURN
END
SUBROUTINE PCH(CHAR)

# PLOT THE ASCII CHARACTER, CHAR.

INTEGER CHAR  # SEVEN BIT ASCII, RIGHT JUSTIFIED.
INTEGER CODE  # TEMPORARY FOR BASIC VECTOR CODE
INTEGER DELTA # MAGNITUDE
INTEGER DIR  # DIRECTION
INTEGER INT  # INTENSITY
INTEGER ASCII(8, 64) # TABLE OF BASIC VECTOR CODES FOR EACH ASCII CHAR.
REAL RTEMP(1)  # USE TO CONVERT THE FOLLOWING NOS. TO OCTAL AS NEEEDED.

DATA ASCII /104, 044, 126, 104, 104, 142, 112, 274, 0, 0,
126, 104, 166, 041, 033, 104, 273, 0,
126, 104, 161, 152, 172, 161, 144, 207, 0,
104, 026, 144, 166, 207, 0, 0, 0,
102, 112, 132, 142, 166, 207, 0, 0,
126, 104, 066, 144, 025, 102, 002, 273, 0,
126, 104, 053, 041, 102, 002, 273, 0,
126, 104, 163, 144, 001, 102, 204, 0,
104, 126, 054, 162, 207, 0, 0, 0,
126, 004, 153, 173, 203, 0, 0, 0,
126, 066, 104, 203, 0, 0, 0, 0,
126, 172, 112, 166, 203, 0, 0, 0,
126, 061, 174, 061, 126, 063, 273, 0,
104, 126, 166, 166, 207, 0, 0, 0,
126, 104, 041, 004, 102, 002, 273, 0,
104, 126, 144, 166, 012, 173, 021, 202, 0,
126, 104, 163, 144, 001, 173, 203, 0,
104, 121, 134, 121, 104, 063, 273, 0,
026, 104, 042, 166, 205, 0, 0, 0,
126, 066, 104, 204, 063, 273, 0, 0,
002, 132, 124, 004, 164, 152, 205, 0,
126, 004, 166, 132, 152, 207, 0, 0,
121, 114, 121, 044, 161, 174, 161, 203, 0,
002, 124, 132, 004, 192, 001, 274, 0,
026, 104, 161, 154, 161, 104, 203, 0,
002, 142, 126, 102, 041, 276, 0, 0,
025, 174, 071, 202, 0, 0, 0, 0,
002, 102, 126, 142, 041, 276, 0, 0,
024, 112, 172, 061, 273, 0, 0, 0,
104, 281, 0, 0, 0, 0, 0, 0,
207, 0, 0, 0, 0, 0, 0, 0,
002, 120, 023, 123, 041, 276, 0, 0,
021, 012, 123, 001, 163, 001, 273, 0,
022, 194, 022, 144, 012, 166, 205, 0,
104, 123, 144, 122, 103, 031, 166, 205,
021, 114, 043, 141, 074, 141, 061, 204,
121, 113, 132, 151, 175, 145, 207, 0,
002, 102, 001, 112, 024, 174, 001, 273,
002, 024, 041, 276, 0, 0, 0,
002, 125, 002, 154, 024, 174, 061, 203,
011, 001, 124, 052, 104, 023, 0, 0,
012, 161, 151, 206, 0, 0, 0, 0,
023, 104, 012, 166, 203, 0, 0, 0,
024, 100, 203, 0, 0, 0, 0, 0,
021, 114, 062, 273, 0, 0, 0, 0,
126, 104, 166, 144, 021, 114, 062, 273,
001, 102, 041, 126, 151, 075, 201, 0,
025, 111, 102, 171, 154, 161, 104, 203,
104, 123, 143, 113, 144, 001, 276, 0,
083, 126, 153, 161, 105, 272, 0, 0,
102, 112, 132, 142, 122, 104, 063, 273.  # 5
126, 066, 104, 123, 144, 004, 273, 0.  # 6
026, 104, 161, 154, 161, 207, 0, 0.  # 7
126, 104, 163, 144, 063, 104, 123, 273.  # 8
004, 123, 144, 123, 104, 163, 273, 0.  # 9
022, 012, 160, 063, 160, 061, 205, 0, 0.  # 1
023, 012, 160, 063, 161, 151, 206, 0, 0.  # 1
021, 003, 132, 112, 001, 062, 273, 0.  # <
022, 104, 022, 144, 003, 274, 0, 0.  # =
021, 001, 112, 132, 003, 062, 273, 0.  # >
025, 111, 102, 171, 152, 163, 205, 0/  # ?

IF ( (CHAR > #37) & (CHAR < #137) )
%
# CHAR IS "PRINTABLE"
CHAR = CHAR[12117]  # DROP HIGH BIT
CHAR = CHAR * 1  # START A: 1 INSTEAD OF 0
DO * = 1, 8
%
CODE = ASCII(I, CHAR)  # VECTOR CODE FROM TABLE
# SOME YO-YO TYPED THE NUMBERS IN IN DECIMAL, SO CONVERT NOW,
ENCOD = (5, RTEMP, 1; ERR=3) CODE
1 FORMAT (15)
DECOD = (5, RTEMP, 2, ERR=3) CODE
2 FORMAT (05)
IF (CODE == 0)
    BREAK  # DONE WITH CHARACTER
DELTA = 2 * CODE(15:17)
DIR = CODE(12:14)
INT = CODE(11:11)
CALL BYECS (DELTA, DIR, INT)
$)
3 RETURN
END
SUBROUTINE VTPLT (TYPE, X, Y)

* PROVIDE HP PLOTTER COMMANDS OF THREE TYPES:
  * TYPE = POINT MOVES THE PEN TO A POSITION CORRESPONDING TO
  * VT-15 ABSOLUTE COORDINATES (X, Y),
  * TYPE = MOVE MOVES THE PEN TO A POSITION CORRESPONDING TO
    VT-15 RELATIVE COORDINATES (X, Y),
  * TYPE = DRAW DRAWS A LINE TO A POSITION CORRESPONDING TO
    VT-15 RELATIVE COORDINATES (X, Y),
  * THE TRANSLATION FROM VT-15 TO HP PLOTTER IS CONTAINED ENTIRELY
    WITHIN THIS SUBROUTINE.

INTEGER TYPE  # POINT, MOVE OR DRAW
INTEGER X, Y  # ABSOLUTE OR RELATIVE AS ABOVE
INTEGER XOFF  # POSITION OF OFFSET AREA IN HP UNITS
INTEGER XmRGN, YmRGN  # FOR ONE INCH MARGINS
INTEGER XvMrgn  # FOR A 2.5 INCH MARGIN
REAL Xscal, Yscal  # VT-15 TO HP UNITS
INTEGER upDwn  # 0 FOR PEN UP; 1 FOR PEN DOWN
INTEGER temp

INCLUDE VTCOM

DATA Xoff /7112/, XmRgn /344/, YmRgn /764/, XvMrgn /1868/
DATA Xscal /5.96/, Yscal /8.28/

IF (TYPE = DRAW)
  upDwn = 1
ELSE
  upDwn = 0
IF (TYPE <> POINT)
  $(
    (X, Y) ARE RELATIVE
    IF ( (X = 0) 0 (Y = 0) )
      $(
        # PEN ONLY
        CALL HPLOT (upDwn, 1, 0, 0)
        RETURN
      )
      X = XPOS = X + (SCALE * 1)
      Y = YPOS + Y + (SCALE * 1)
  )

XPOS = X
YPOS = Y

IF ( Horv = 1)
  $(
    # VERTICAL PLOT (NO OFFSET AREA)
    IF (OFFSET = 1)
      RETURN
      TEMP = Y
      Y = 1023 - X
      X = TEMP
    )

# CONVERT TO HP UNITS
X = IFFIX ( (Xscal * FLOAT (X)) + 0.5)
Y = IFFIX ( (Yscal * FLOAT (Y)) + 0.5)
IF (Horv = 0)
  X = X + XmRgn
ELSE
  X = X + XvMrgn
Y = Y + YmRgn
IF (OFFSET == 1)
    X = X * XOFF

CALL HPILOT (UPDOWN, 0, X, Y)
RETURN
END