CENTRAL CIRCULATION AND BOOKSTACKS

The person borrowing this material is responsible for its renewal or return before the Latest Date stamped below. You may be charged a minimum fee of $75.00 for each non-returned or lost item.

Theft, mutilation, or defacement of library materials can be causes for student disciplinary action. All materials owned by the University of Illinois Library are the property of the State of Illinois and are protected by Article 16B of Illinois Criminal Law and Procedure.

TO RENEW, CALL (217) 333-8400.

University of Illinois Library at Urbana-Champaign

JUL 1 1 2001

When renewing by phone, write new due date below previous due date.

L162
SOME PROPOSALS FOR MODIFYING THE
GRASS GRAPHICS LANGUAGE

by

PAUL EUGENE NIVERVILLE DES TROIS MAISONS

December 1972
SOME PROPOSALS FOR MODIFYING THE
GRASS GRAPHICS LANGUAGE*

by

PAUL EUGENE NIVERVILLE DES TROIS MAISONS

December 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

* This report supported in part by the Atomic Energy Commission under contract US AEC AT(11-1)1469 and submitted in partial fulfillment of the requirements of the Graduate College for the degree of Master of Science in Computer Science, February 1973.
ACKNOWLEDGMENT

The author wishes to thank Professor C. William Gear for the opportunity to do this work, and Barbara Armstrong for typing the manuscript.
The purpose of this thesis is to outline a reconfiguration of the Graphical Remote Access Support System (GRASS) as developed at the University of Illinois. In this paper, we shall attempt to improve upon the graphical procedures of GRASS [2], while still fulfilling the original system goals [1]. For this reason, familiarity with the current system is required.
TABLE OF CONTENTS

1. INTRODUCTION .................................................. 1

2. POSITIONING POINTS .......................................... 3
   2.1 Point Hierarchy ......................................... 3
   2.2 A More General Dependence ................................ 11
   2.3 Specifying the Hierarchy ................................ 12

3. TEXT EDITORS .................................................... 18
   3.1 Keyboard Oriented Editor ................................... 19
   3.2 Joystick Oriented Editor ................................... 21

4. DATA TYPES AND ACTIVITIES .................................. 23
   4.1 Adding ....................................................... 24
      4.1.1 Adding Mnemonics, Terminals, Comments .............. 25
      4.1.2 Adding Lines and Connections ........................ 26
   4.2 Deleting ..................................................... 28
   4.3 Moving ....................................................... 29
   4.4 Changing .................................................... 31
      4.4.1 Changing Mnemonics .................................. 32
      4.4.2 Changing Comments ................................... 34
      4.4.3 Changing Terminals ................................... 36

5. SPECIFICATION AND SYSTEM COMMANDS ......................... 37
   5.1 Miscellaneous System Directives .......................... 37
   5.2 The Specify State ......................................... 39

6. CONCLUSIONS .................................................... 43

LIST OF REFERENCES ............................................... 45
1. INTRODUCTION

Through the last several years there has been increasing interest in computer programs which analyze networks of interconnected, lumped components. At first, these programs were designed each to handle a particular type of network. An analogue computer package, for example, could not be directly applied to interconnections of mechanical components. This approach failed to take advantage of the common features among all network problems.

A system which did not possess this inflexibility was created at the University of Illinois [1]. Rather than restrict the user to communicating a fixed set of physical quantities (e.g., voltage and current) at interconnection points, this system allowed these sets to be user-defined. Another feature permitted the network elements to be constructed as needed. Their behavior was defined by the performance of a representative network of existing components, supplemented or supplanted by equations involving terminal and other variables. This, coupled with some simple rules for the interaction of quantities at terminals, was sufficient to achieve generality.

The procedures for supplying the above system with this information are described in [3]. The user communicated this data conversationally through a vector and alphanumeric graphics terminal—thus, he had immediate visual feedback on the success of the communication. The graphics routines provided were always sufficient to make known the user's intentions; however, there were several weaknesses with them. First, each tool seemed to be designed to solve an isolated problem. The result was a conglomeration of differently-styled procedures, as a group difficult to learn, and individually sometimes inconvenient to apply. Another weakness was the positional independence of points in a drawing from each other. This, together with the fine resolution possible with the graphics display, made it difficult to align a number of objects or
vertices horizontally or vertically—a common operation in laying out networks or drawing closed figures. Also, the system often required nearly identical information to be input; for example, to specify two orientations of the same network component.

In this paper, we shall redesign the graphics system, eliminating the above-mentioned faults. We shall start by examining the fundamental operation of specifying the position of points. We shall then develop a more powerful text editor. These tools will be combined in the following sections where the drawing and the constraining of networks and elements shall be presented.

Throughout, we shall try to attain the goal of uniformity. Whenever we consider a feature which might prove convenient, it will be imperative that this feature not disturb the system's overall structure. This should help ensure that the system is easy to learn, and keep implementation difficulties small (the special case is the bane of the well-ordered program). At times, we may seem to lose as much by this strategy as we gain. However, in this author's view, such order merits some sacrifice.
2. POSITIONING POINTS

In this section we shall develop the procedure for specifying locations in a drawing to the system. The mechanism we use will be of fundamental importance in our subsequent discussions, for this operation will always be embedded in positioning or identifying any item on the screen. Of course, we will demand from the display hardware the ability for us to indicate a spot on the screen with one quick action. Without this requirement, our system will be unbearably clumsy from the outset.

The reader should bear in mind throughout this section that a picture displayed on the screen will be either a network of devices or a mnemonic drawing of an individual device.

2.1 Point Hierarchy

When we locate a point on the screen, we may wish to express one of two ideas about it; the point may derive its placing either from the construction of the picture as a whole--in which case we shall call it autonomous, or from specific, pre-existing points--when we shall term it constrained. For example, the point may be the center of a star which we are about to draw, whose position is simply a matter of aesthetics. The location of the point is not derived, in any rigid sense, from other points in the picture. On the other hand, a corner of this same star would not be independent of the other corners, nor of the center. So, when we locate a point on the screen, we would like the point's autonomy or the nature of its constraint to be noted and recorded by the graphics system.

There is an infinity of ways in which a point may lose a degree of freedom, through constraint by another point. (For example, "point A will lie on the curve \( y = e^{.14x} + C \) through point B.") We will restrict ourselves to
optionally requiring the point to lie on the horizontal or vertical through another point. This device should be reasonably helpful, and is probably easier to implement than any other form of constraint; furthermore, examining it may indicate paths to greater generality.

Let us assume that the screen is referenced via a rectangular co-ordinate system. Thus, to use a point the computer must be able to quote its horizontal and vertical (or x and y) co-ordinates. Suppose that space is allocated within memory for this. Figure 2.1-la shows an autonomous point A at co-ordinate position (5.2, 4.1); and a point B, constrained vertically by A, at (6.2, 4.1). If this information is stored as in Figure 2.1-1b, A and B will be displayed at the proper locations, but the subordinate nature of B to A is not shown.

Weaknesses of this nature may not be apparent to the system user who is involved merely with adding new points to the picture. However, when he turns to other activities (namely, moving and deleting points) the faults become more glaring. In this particular case, moving A vertically would destroy the alignment of A and B. If B were truly constrained by A, it would shift vertically, also.

Figure 2.1-1c is an attempt at correcting this fault. Instead of storing a co-ordinate value in \( y_A \), we indicate that the data should be obtained from \( y_B \) by inserting a pointer. (An additional flag bit, not represented, will be necessary.) When we display or move point A, we remember to fetch or alter \( y_B \).
\[(5.2, 4.1) \quad \text{A} \quad (6.2, 4.1) \quad \text{B}\]

Figure 2.1-la

\[
\begin{array}{cc}
X_A & 5.2 \\
Y_A & 4.1 \\
\end{array} \quad \begin{array}{cc}
X_B & 6.2 \\
Y_B & 4.1 \\
\end{array}
\]

Figure 2.1-lb

\[
\begin{array}{cc}
X_A & 5.2 \\
Y_A & \phantom{4.1} \\
\end{array} \quad \begin{array}{cc}
X_B & 6.2 \\
Y_B & 4.1 \\
\end{array}
\]

Figure 2.1-1c

Figure 2.1-1. Representations of Two Horizontally Aligned Points

How may we add a new point, C, also constrained vertically by A as in Figure 2.1-2a? If we try to extend the above method, we will presumably make \(y_B\) point to \(y_C\), where the co-ordinate value is now stored (Figure 2.1-2b). In effect, we create a linked list with the autonomous point at the head, the points constrained by it following, and the last point containing the co-ordinate. Note that whereas the method of Figure 2.1-1b allows us to display a point when we can locate its positioning information, in Figure 2.1-2b we are forced to search to the end of a list before we can know a point's co-ordinates. However, moving an autonomous point will also move any points constrained by it.

\[(5.2, 4.1) \quad \text{A} \quad (6.2, 4.1) \quad \text{B} \quad (7.5, 4.1) \quad \text{C}\]

Figure 2.1-2a

\[
\begin{array}{cc}
X_A & 5.2 \\
Y_A & \phantom{4.1} \\
\end{array} \quad \begin{array}{cc}
X_B & 6.2 \\
Y_B & \phantom{4.1} \\
\end{array} \quad \begin{array}{cc}
X_C & 7.5 \\
Y_C & 4.1 \\
\end{array}
\]

Figure 2.1-2b

Figure 2.1-2. A Prepresentation of Three Aligned Points
Consider, now, what happens if we move B vertically to, say, $y_B = 5.2$. Since $y_C$ is at the end of the chain (actually a subchain) headed by $y_B$, this effectively moves all the points A, B, and C upward, keeping them tied together. Because of the data structure, A is constrained by B as surely as B is by A. This sort of mutual bondage ignores the fundamentally hierarchical nature implicit in the constraint idea. Moving B should separate it from A and C, and C should remain subordinate to A since it was defined in terms of A, not B. We need a tree type of data structure. And, we need some additional terminology. Let us describe the situation of A constraining B and C, A not necessarily autonomous, by the family relations: A is the parent of B and C, B and C are the children of A, and B is the sibling of C.

Restricting ourselves to vertical constraints for the moment, let us investigate how the tree structure needs to be implemented. (Bear in mind throughout, however, that two trees are required, one for each coordinate.) Since any point in the picture may have an arbitrary number of children, each node in the tree may have an arbitrary number of branches. This does not, of course, mean that storage for the point must be of variable size. We may store an arbitrary collection of trees as one binary tree (see [2]). We can do better than this, however, provided we allow some inefficiency in the delete operation. We simply store a pointer to the parent node in each child node in the tree. We may do this because the parent point of any other point is unique. As an example, consider a picture with two autonomous points A and B, each with one child point, $A_1$ and $B_1$. This situation is shown in Figure 2.1-3a with an arrow from a vertically constrained point to its parent. The data is stored as in Figure 2.1-3b; to show a pointer to $y_A$ instead of a co-ordinate value, we have written the entry as $y_A$. 
In Figures 2.1-4a and 4b, we have added autonomous point C to the above scene, and in Figures 2.1-5a and 5b an additional point $A_2$ constrained by A. Note that the entry for $y_{A_2}$ is a pointer to $y_A$, not $y_{A_1}$. Finally, in Figures 2.1-6a and 6b we add an additional level to the tree structure with point $A_{21}$, whose parent is $A_2$. Thus, a fairly general, although quite simple, tree has been stored, using no more space (except for flags) than was needed in the simple data structure of Figure 2.1-2.
Figure 2.1-4a

\[
\begin{array}{c|c}
 x_A & 2.0 \\
 y_A & 6.0 \\
\end{array} \quad \begin{array}{c|c}
 x_B & 1.0 \\
 y_B & 4.0 \\
\end{array} \quad \begin{array}{c|c}
 x_{A1} & 5.0 \\
 y_{A1} & y_A \\
\end{array} \quad \begin{array}{c|c}
 x_{B1} & 4.0 \\
 y_{B1} & y_B \\
\end{array}
\]

Figure 2.1-4b

Figure 2.1-5a

\[
\begin{array}{c|c}
 x_A & 2.0 \\
 y_A & 6.0 \\
\end{array} \quad \begin{array}{c|c}
 x_B & 1.0 \\
 y_B & 4.0 \\
\end{array} \quad \begin{array}{c|c}
 x_{A1} & 5.0 \\
 y_{A1} & y_A \\
\end{array} \quad \begin{array}{c|c}
 x_{B1} & 4.0 \\
 y_{B1} & y_B \\
\end{array}
\]

Figure 2.1-5b

Figure 2.1-5. A Second Point Dependent upon A is Added
Figure 2.1-6a

$$\begin{array}{c|c}
X_A & 2.0 \\
Y_A & 6.0 \\
\end{array} \quad \begin{array}{c|c}
X_B & 1.0 \\
Y_B & 4.0 \\
\end{array} \quad \begin{array}{c|c}
X_{A1} & 5.0 \\
Y_{A1} & Y_A \\
\end{array} \quad \begin{array}{c|c}
X_{B1} & 4.0 \\
Y_{B1} & Y_B \\
\end{array}$$

Figure 2.1-6b

$$\begin{array}{c|c}
X_c & 3.0 \\
Y_c & 2.0 \\
\end{array} \quad \begin{array}{c|c}
X_{A2} & 7.0 \\
Y_{A2} & Y_A \\
\end{array} \quad \begin{array}{c|c}
X_{A21} & 9.0 \\
Y_{A21} & Y_{A2} \\
\end{array}$$

Figure 2.1-6. A Third Level is Added to the Tree

To change the position of a point is a straightforward operation. Figure 2.1-7 illustrates the ease with which this is performed. We have taken point $A_2$ of Figure 2.1-6a and reconstrained it to a horizontal through $B_2$. This is accomplished merely by changing $y_{A2}$ to designate $y_{B2}$. Notice that point $A_{21}$ follows automatically. The only pitfall we must avoid is defining a co-ordinate directly or indirectly in terms of itself, which would be clearly meaningless. This check would be required in any tree structure.
As suggested above, deletion of a point can be more complicated than adding or moving. This will be true whenever the point is a parent. Whether we interpret this case as requiring additionally the deletion of all offspring, or, less drastically, demanding that children of the doomed point be put on the same level as its siblings, we will have to locate the afflicted offspring. The information to do this is not provided within the data structure, so it will be necessary to search all points to locate the children of any single point. There is no way to avoid the dilemma and still retain our simple data structure. Unfortunately, a threaded tree structure, the least required to both ascend and descend a tree, would more than double the storage for any given point. Before we do anything this drastic we should note that searching all points for one with a given co-ordinate will be a common operation even if not required in deleting a parental point.
2.2 A More General Dependence

We will, for most of the remainder of this paper, accept our single upward pointer, tree data structure, and all its inherent limitations. In this section we will briefly scan some of the possibilities opened by a particular, considerably more complicated data structure.

If each co-ordinate of a print could specify in addition to the parent co-ordinate, a signed offset to be added to this ancestral value, our system of constraints gains considerable power. The child point would not be required to occupy the same horizontal or vertical as a parent, yet it could be required to duplicate all of the parent's motions. With this device and adequate commands we could break a picture into a number of constituents which could be adjusted at will, yet relocated easily either individually or as part of the picture as a whole. For example, we might make a line drawing of a house, then after it was quickly completed, adjust the shape of its windows, relocate the door, or move the entire house to another spot in the scene by moving some particular point.

The idea of moveable constituents has a role to fill in network drawings as well. There are some network elements which consist pictorially of two or more associated parts requiring no physical proximity, yet intimately related by equations. An example is an electric relay whose coil and contacts are often more conveniently drawn in separate areas of a network, but which are definitely tied together in action.

Another possible application is in allowing the graphics system user to define a number of primitive shapes, such as rectangles, stars, stick figures. These could be copied into the display, then suitably modified, yet never lose their integral nature.

These ideas merit closer scrutiny, which will not be undertaken here.
2.3 Specifying the Hierarchy

We now come to the problem of how to communicate to the machine exactly how we want each point's co-ordinates constructed. We would certainly like the tools for this specification both powerful and convenient. It is also vital that they be flexible enough to communicate different senses in the varying contexts which will arise. Later we will discuss these contexts in some detail. For the moment we note that picture elements of a number of different classes, or data types, will be possible in the system, and the user will engage in certain activities universal to these classes (for example, adding them to the scene or moving them about). To make it clear which data types are involved, there will be a mode of operation corresponding to each type in which only points used in defining elements of that data type are available for reference.

Two goals which we will try to reach with our communication are the following. We want to keep the user as aware as possible of any machine state that will affect his actions in the current activity. (The information might include the set of candidates for a sought point, or the status of partially incomplete maneuvers.) We also wish to allow the user to abort any current activity unit.

As hardware constraints, let us assume that each display terminal is equipped with an analogue device whose position can determine a location on the screen. Such a device we will term a joystick, even though it may be a light pen, a tablet stylus, or some other invention. We will also assume that a sufficient number of buttons or switches are available to interrupt the display processor.

Every point has two co-ordinates, and each co-ordinate may be autonomous or constrained. Four buttons then would be adequate to define points, if one were willing to specify each co-ordinate of every point
separately. Such a design would be clumsy. In fact, there are nine ways of entering points which should require a minimum of fumbling with buttons. They are:

1. Constrain the new point horizontally to the last specified point, and take as the vertical the position specified by the joystick.
2. Do the same as 1 above, interchanging "horizontal" and "vertical."
3. Take both the horizontal and the vertical from the joystick position.
4. Constrain the new point horizontally to the last point specified, and vertically to an existing one specified by the joystick.
5. Do the same as 4 above, interchanging "horizontal" and "vertical."
6. Constrain the horizontal of the new point to an existing point specified by the joystick, and define the vertical autonomously by a different joystick position.
7. Do the same as 6 above, interchanging "horizontal" and "vertical."
8. Constrain the horizontal to a joystick-specified point, and the vertical to a different joystick-specified point.
9. Constrain the horizontal and vertical to the same joystick-specified point.

Operations 1, 2, 4, and 5 use a concept of a previously specified point, not discussed before. The idea is that under certain circumstances, when the user is adding elements to a picture, he will find it convenient to specify successive points aligned horizontally or vertically. This situation might arise, for example, in a relaxation problem in which the user wishes to define a grid of resistances. To implement this concept, the system must note the storage location of each newly defined point, and maintain a marker on the screen indicating its position.
If the user were not adding but moving points, the previous point would best be interpreted as the point being moved. In other activities, since the user would wish to specify only existing elements, all operations but 9 above would be illegal.

Each of operations 2-9 requires at least one search of available data points to find one to match the joystick position. However we implement this, we want the user to be able to consider the point located by the system, and accept it, or reject it and continue searching.

It appears as though the following system for defining points meets all the requirements of facility and power outlined above. Six buttons would be associated with the joystick. Before each new point is specified, a "working point" would be initialized horizontally and vertically to undefined. Three of the buttons would be used to modify the working point. The H button would affect the horizontal co-ordinate, the V button the vertical co-ordinate, and the B button would modify both. If, before any of these three buttons were pushed, the search (S) button were hit, a search (detailed below) would be initiated for a point "close" to the joystick position. If the point found were not acceptable, the search could be continued by pressing S again, and as often as necessary. To accept the proffered point, the user would hit one of H, V, or B causing the horizontal, the vertical, or both co-ordinates of the working point to be constrained to the located point.

Normally, once H, V, or B was hit, the "working point" would be taken as the one which was to be constructed. However, if the compose (C) button were pushed at the start of the definition process, two such hits would be required, allowing each co-ordinate of the working point to be defined separately. At this time of acceptance, should either of the working point's co-ordinates have remained undefined, it would designate the previously defined point's
co-ordinate in the adding activity, or would be set to the moved point's co-ordinate in the move activity (Section 4.3).

The remaining button of the six would be an abort (A) button. Hitting it would cancel any partially completed action. Since the abort button would be infrequently used, the other five buttons should be positioned so that the user could keep one finger on each throughout the drawing process. This would make the controls seem to be extensions of the users body, thus encouraging speed and minimizing error.

Exactly how to perform each of the 9 methods of definition above is detailed in the following list. The buttons are listed by their initials in the order in which they are pushed. In every case where the search button appears, it may be hit consecutively as often as necessary.

1. V
2. H
3. B or CHV or CVH
4. SV
5. SH
6. CSHV or CVSH
7. CSVH or CHSV
8. CSVSH or CSHSV
9. SB

One might note that since the C button is never pushed in the middle of an operation, and since one would not need to abort the operation except in the middle, we might combine the action of the C and A button. The author believes that this would obscure a rather clear-cut division of responsibility among the buttons, and should be avoided.
As an example of the use of our controls, let us locate the vertices of a rectangle, as in Figure 2.3-1. The corners are numbered in the figure so that we may refer to them in the text. First, locate position 1 with the joystick and press B; point 1 is now defined autonomously. Next, move the joystick to position 2, pressing V. Point 2 is defined in a vertical line with 1. Now, indicate location 3 and hit H to define point 3 on a horizontal with position 2. To ensure that point 4 is vertically above point 3, and horizontally aligned with 1, indicate position 1 with the joystick, press S until the system identifies point 1 as being found. Then press V. All four points will now be properly defined. If we then move point 1 in any direction, points 2 and 4 will be adjusted to maintain a rectangle. However, moving point 2 arbitrarily will cause point 3 to be shifted, but not 1, for 2 was constrained to 1, and not vice versa.

1. 4.
2. 3.

Figure 2.3-1. The vertices of a Rectangle

The searching which is so necessary a part of our joystick controls must be performed in a proper way to ensure its full usefulness as a tool in all possible activities. For this reason we present the exact procedure to be followed.

When the user initiates the search for a point by pushing the S button, the system must first note the joystick co-ordinates. Then all points of the currently available data (i.e. consistent with the current mode) must be examined for exact coincidence with the joystick point. Whenever a suitable point is located, its position should be indicated to the user. Should he reject the point by pressing S again, the search should be continued from the
point it was suspended. Each time the available points are exhausted, the scope of the search (that is, the tolerance allowed about the input point) should be increased and the set of points scanned anew. These further scans should ignore points already rejected. With these procedures the user could locate any point on the screen from an arbitrary joystick position, although it would save him time if his initial guess were good.
3. TEXT EDITORS

Our techniques for point definition will be a major tool in the describing of networks in our graphics system. In this section another equally important tool will be presented—the text editor. It will be used whenever textual information of any kind need be stored with the graphical data.

Even the very simplest of text editors can attain the ultimate goal of all: that of adding to and replacing strings of characters. However, some may be clumsy for all but very trivial operations. Our aim here is to provide enough power to keep boring, wasteful actions to a minimum. In particular, it should not be necessary to retype a line in order to change characters, delete characters without leaving gaps, or insert characters.

Two text editors which address themselves to these problems will be presented. Their differences will stem from two alternate philosophies. The first assumes that character manipulations should be totally keyboard oriented, even to the specification of positions of symbols within the text. The second rationale is that to quickly specify a position on the screen, even a location in text, the joystick should be used. The author favors the first idea over the second because of the inconvenience in shifting hands between the keyboard and the joystick; because a joystick is sometimes difficult to control to fine accuracy; and because text editing, often being sequential in nature, does not require the direct access capabilities of a joystick. More pressing considerations may influence the decision in particular cases, however, thus the dual presentation.

Both editors will assume that the entire text to be modified will appear alone on the screen in some standard location. Whenever a change is made to the text, the change will be made in place immediately. On some displays this may be automatic. With storage CRT screens, the system may
have to continuously regenerate the line under change, and blank and refill the screen each time one line is affected. This will, of course, be made easier by not having to display the picture along with the text. The necessity for keeping an up-to-date display must be emphasized: it is asking too much of the user to visualize both the desired result of a change and the current state of the text.

3.1 Keyboard Oriented Editor

In this section, the text will be considered as a stream of characters. Separate lines will exist on the display. However, within the stream they will be demarcated only by the presence of a single character: the line return. This character, though it has a special significance, will be manipulated as any other in the display.

A cursor will be used to indicate on the screen any particular location in the stream. Four buttons will control the movement of the cursor in the character string. Two of these will move it left or right sequentially past each position. The other two will cause it to jump up or down, indicating the first position on a line. These four buttons, plus one other, should repeat their action when held down, similar to a keypunch duplicate button. If such buttons do not exist on the keyboard, it should not be too difficult to add them.

Any time a character is entered on the keyboard, it will be inserted in the text between the character indicated by the cursor and the one to its immediate left, no characters will be deleted, and the cursor will be advanced past the inserted character. With this device, a line may be split into two by inserting a line return within it. Deletion of characters will be performed by the fifth repeating button, from left to right, one character at a time; the vacant space will be squeezed from the text; and the
cursor will appear to remain stationary. If a line return should be deleted, the lines it separated should be concatenated and displayed as one.

To allow movement of blocks of text, another operation and two special keys should be added. The FROM key will be used in pairs along with the cursor to mark the beginning and ending characters of a string to be copied. This string will be duplicated each time a pair of hits on the TØ key have occurred. This key also will be used in conjunction with the cursor to mark the beginning and ending characters of a block to be replaced by the duplicated string. The lengths of the two strings need not be equal. In fact, it may sometimes be convenient for either the source or the destination substring to be null. The null string would be any string whose ending character preceded the beginning character, and would be positioned in front of the beginning character.

Let us see how one would perform some common operations with this editor. To substitute one word for another, one would advance the cursor to the incorrect word, type the correct word, and then using the delete key would obliterate the unwanted characters. To append several lines of text to the end of that given, one would position the cursor at the last line return, and type his lines, beginning each with a line return. Moving a number of lines to before a certain line is only slightly more complicated. One would position the cursor to the first character to be copied, then hit the FROM key. Next, advance the cursor to the last line return to be moved, and again hit the FROM key. Then, one would move the cursor to the first character of the destination line, hit TØ, then left one character, hit TØ again. The original text block having been duplicated, one could obliterate it using either the delete key or by moving the null string in its stead. Typically, we have replaced a weak but easily specified command with a more powerful but more complicated one.
3.2 Joystick Oriented Editor

In this editor, the stream of characters idea and the role of the line return will be identical to that of the previous section. Also carried over will be the three manipulative operations, namely insertion of typed characters, deletion of existing characters, and replacement of one substring by another. These will appear in different form, however, since the joystick will be used to specify the range and the site of the operations.

Since the joystick console will have a number of buttons, it seems reasonable to use these to indicate the action to be taken with the text. Accordingly, the C (for cursor) button will direct the system to note the character position indicated by the joystick. (On systems without a cursor, or whose cursor is used to feed back the current position of the joystick, it will be necessary to display a special mark or character on the screen at the noted position.) Characters typed at the keyboard will be inserted at this position and the cursor advanced as with the Keyboard Editor. Substrings in the text will be taken as extending from the cursor position at the left to the joystick position at the right. Should the joystick be to the left of the cursor, the system will interpret this as the null string at the cursor position.

The V (for Void) button will cause the substring as defined above to be deleted. The H (Hold) button and the S (Substitute) button will take the place of the FROM and TO special keys of before. Hitting the H button will cause the substring terminating at the joystick position to be held for future duplication; pushing the S button will cause the indicated substring to be replaced by the held string.

Although it might be costly to implement, a useful application of the A (Abort) button would be to cause it to restore the text as it was immediately preceding the last operation.
To substitute one word for another, one would indicate the first character of the incorrect word with the joystick, and press C. One would then type the correct word, and afterwards indicate the last letter of the incorrect one, pressing V. To append lines to the end of the text, one would indicate the last line return with the joystick and press C, then type each new line desired, preceded by a line return.

To move a number of lines to before a certain line, specify the first character in the lines to be moved with the C button, and the last with the H button. Then indicate the first letter following the new location for the lines with the C button, and a previous character with the S button. Lastly, delete the copied lines at their original position.
4. DATA TYPES AND ACTIVITIES

We have developed the means for succinctly communicating the two basic types of information handled by graphics terminals: viz. textual and positional. Now we must build from them elements useful in specifying networks, and fit these elements coherently into an overall system structure.

In our system we want to define networks in terms of components of known behavior as well as by explicitly relating different points in the network through equations. Furthermore, we may want to use a network as a component in a more complex network, and to do this we need the ability to represent the simpler network by a line drawing with connection points. Thus, we must account for creating and using the following: lines, blocks of comment text, connection points (terminals), circuit connections, and mnemonic representations of networks.

If we examine these elements we will see that they fall into two categories--those that need a single position in order to be located, and those that need two. We, therefore, have a problem in how to treat all five coherently. We might possibly allow for four single-position data types: points, connection points, comments, and mnemonics. A line could then be a binary relation on points, and a circuit connection one on connection points. This might be an elegant approach; however, it raises the issues of the use of an unconnected point and of how to identify related points (we still effectively have two-point data types). We will, therefore, abandon this line of attack, although bearing in mind the similarity between lines and connections.

The user may want to do things with these data types other than add instances of them to the current network. He may also want to move an incorrectly positioned instance, change its acquired attributes, or delete it
when it is no longer needed. Regardless, his actions will always be related to one of the data types. To allow him to specify which he is currently interested in, we will define the system mode as referring to the current active data type. When in a particular mode, the user will be able to refer only to those predefined points giving position to elements of the current type. This mode will be displayed as a system status information, and the user will be able to select the current one by means of a five position switch on the joystick console. (Equally effective would be to list all five modes, with the current one marked, somewhere on the screen. The user could then select another by specifying the desired one with a joystick hit.) To allow the constraining of one type of element positionally to another of different type, switching between modes should not nullify a completed search for a point (see Section 2.3).

Note that it would be possible to consider the activities of adding, moving, changing, and deleting as partitioning the state of the system. This being so, we will require the current activity also to be displayed and provide a means of switching from one activity to another. Exactly what these activities imply when considered in combination with the five modes will be the topic of the next four sections.

4.1 Adding

As mentioned, we have two types of elements to deal with: those located completely by one point and those requiring two. Single point elements are not completely defined by their data type--for example, it is not enough to say we would like a mnemonic instance added at some point, we must specify which instance. Two-point elements do not require this extra information--a line is a line. However, other knowledge is required, namely, how to treat the
endpoint of the previously defined line. Whenever extra data such as this is necessary, it will be specified by selecting with the joystick a one-word descriptor of the option in the menu area. The joystick console has six buttons for specifying points, any subset of which could be permissible to choose an option. For instance, the S button might be allocated to mean "select." The option having been selected, it will apply in this mode for each point added to the screen until another option supersedes it. If the current mode or activity is altered, both the previous and the selected option will be remembered so that they may be restored and properly highlighted when the mode is again re-entered.

4.1.1 Adding Mnemonics, Terminals, Comments

When the system is in the mnemonic mode and the adding activity, the names of representations of the user's previously defined circuits will be displayed in the menu area. The user will select from them a desired mnemonic, then specify the locations of the instances desired. Each instance will appear immediately after its location is specified, available for whatever manipulations might be necessary.

In the terminal mode, the system will display the available terminal types, as obtained from the network type specifications (Section 5.2), in the menu area. Each instance of a terminal added in this mode will be considered external; that is, it will be one of the connection points included in a mnemonic representation of the current circuit. (Every terminal, external or otherwise, in a picture will have assigned to it a unique terminal number. The terminals of a network and its representation must be made to correspond in type and number.)
To maintain consistency with one-point data types, any comment which is to be displayed in a picture will have had to be predefined by means of a command (Section 5.1); all predefined comments will have their titles displayed in the menu area. Any such comment may be selected for display at a number of locations in the same spirit as mnemonics.

4.1.2 Adding Lines and Connections

Although to define a line of arbitrary length and orientation would require the specification of two points, in most cases a user does not need and would only be burdened by this kind of generality. Since almost every line he draws will have a common endpoint with another line, it would be more efficient to accept one of the endpoints of his most recent line, combined with a single new point, as specifying the line. This would allow the user to draw chains of lines or stars (i.e. radiating lines) depending on which endpoint of the previous line was used.

We will, therefore, allow the user four choices in the menu in the line mode. He may make the next point start a sequence of lines, all of which have a common beginning point. He may start a sequence of chained lines, whose beginning points are the previous line’s ending point. Alternately, without starting a new series of lines he may specify that either the previous line’s beginning or ending point is to be used, along with each new point. Let us, for discussion purposes, suppose the four options are displayed as NEW STAR, NEW CHAIN, STAR, and CHAIN, respectively. To draw the design in Figure 4.1.2-1 the user could indicate NEW CHAIN with the S button, then 2B, 3V, 4H, 5V, 6H, 2SV, 2SB in this order. The number here indicates the approximate vicinity of the joystick, and the following letters the joystick pushbuttons hit in order. He might wish to impose a different hierarchy among the points, however. Specifying first NEW STAR, he could hit 1B, 2H, 7H, 5V, 4V, then CHAIN, 2SH,
2SB, NEW CHAIN, 7SB, 5SV, 5SB. Assuming that the "previously specified point" of Section 2.3 is always set to the next starting point, the figure drawn will appear the same. The hierarchy of points will be different, however. In the second case, points 2, 4, 7, and 5 are constrained horizontally or vertically to point 1, while point 3 is constrained to points 2 and 4, and point 6 to points 5 and 7. This is radically different from the first structure.

![Diagram of points 1 through 7 with constraints](image)

Figure 4.1.2-1. An Arbitrary Shape to be Drawn

Since lines and connections have an almost identical appearance, we should like to specify connections as we do lines. It is possible to do just this if we resolve some additional complications.

The cause of these extra problems is the nature of the endpoints of connections: they must necessarily be terminals. Thus, whenever we request a connection terminating at a point, the point must either be the location of a pre-existing terminal or of one we want to add. How will we distinguish between the different requests? And, what type of terminal should be added here?

The answer to the first question is obvious if we observe that to specify the location of a new terminal we would not want to force it to be coincident with a pre-existing terminal. That is, to search for an old terminal, then accept both its co-ordinates (action 9 in Section 2.3) would be a useless way to position a new terminal. On the other hand, this would be the only good way to specify a pre-existing terminal. Consequently, with a new terminal will be added to the current network at the end of the connection if defining methods 1 through 8 of Section 2.3 are used to locate the endpoint.
When method 9 is used, however, the connection will be made to the pre-existing terminal at this location.

We may answer the second question by noting that an internal terminal should acquire the type of whatever terminal it is connected to. That it is connected to at least one can be ensured by allowing internal terminals to be added to the picture only in the above way. This will be ensured if whenever we specify NEW STAR or NEW CHAIN in connection mode, that the next point hit must be a predefined (and so already typed) terminal. Doing this will allow us always to verify the compatibility of two connected terminals since the type of each must be known at the time of connection.

4.2 Deleting

When the user wants a data instance to be deleted, there are no options which he must specify. His need is very simply expressed: a particular item must be removed from the picture. Consequently, the system will display no options in the menu area in the delete activity.

Identifying a single-point data instance is straightforward: in the proper mode the user searches for the point positioning the item using the joystick and the S button. When the system indicates that it has found the sought point, he pushes B to accept the point.

A two-point item requires two such identifications--one for each endpoint. During the second of these, the system may identify points which are not connected to the first endpoint at all since each search is carried out independent of any previous result. This poses no particular problem; the user would ignore such points anyway. When both points are identified, the system must then locate the line or connection joining the two indicated
points if one exists. During the searching process if the abort button is pushed, the user should be required to respecify both endpoints. Similarly, if the user changes mode or activity in the midst of specifying any element for deletion, he should be compelled to respecify the element from scratch.

When the system has marked an item for deletion, there are two things which might be necessary before deletion may be completed. If the deletion causes the removal of a parent point, all the offspring co-ordinates must be set to the value (be it pointer or positional) of the parent co-ordinate before deletion (see Section 2.1). Also, deletion of some elements may require the implicit deletion of others. While lines, connections, and comment may be discarded without requiring this, removal of a mnemonic instance or an internal or external terminal demands deletion of all associated connections as well.

4.3 Moving

When we move a data type instance, we must specify two things: the new location for the item, and the item itself. Now, the instance may be constrained to other elements in the picture. Such a bond, although embedded in the data structure for all points, would not be obvious from the display. As he searches for an item to be moved, the system should inform him of such a relationship, so that he may renege on his choice for relocation, rather than destroy a hierarchy which he still desires.

Depending on how the user intends to move a constrained instance, he may or may not be about to break a constraint. To keep from displaying unnecessary warnings, we shall require the new location to be supplied first, and then the current location. Then, when the system proffers a point
during searching for the correct element, it will know which points, if any, to highlight as being the parents of co-ordinates to be changed.

This indication of the constraining points of a candidate for moving is especially important when several points—all locating instances of one data type—coincide. This situation is improbable for any data type but lines for which it will be typical (since different lines often have common endpoints). The user should be able to probe the interdependence of these superposed points as he searches for the one he wants to move. Note that though there may often be one patriarchal point on which all other coincident points depend, circumstances may give rise to a situation where no such point exists. If it were the user's goal to move all lines from one point, in the former case he could accomplish this by locating the autonomous point. In the situation presented in Figure 4.3-1, however, to move points A, B, C to position (6.0, 7.0) could not be accomplished with fewer than two moves regardless of how it were approached.

![Figure 4.3-1a](image)

![Figure 4.3-1b](image)

Figure 4.3-1. Interdependent Points with No Single Patriarch
The user may wish to relocate an instance of some type to be constrained to one of another type. To be capable of this, the system should allow a switch of modes at any time up to the identification of the element to be moved without losing information. The same capability was allowed for under Adding, Section 4.1. It is not necessary to similarly allow a switch in activity. In fact, it would be safer to disallow it, or rather restart the destination specification when the activity is re-entered.

As has been implied in our discussions above, we will interpret moving a line as moving one endpoint at a time. Our other two-point data item, the connection, requires no moves at all since moving terminals and mnemonic instances will require as well shifting any associated connections. Therefore, all our moves will be specified the same way, i.e. giving the destination and the source positions, respectively. There are no options to supply, so none need appear in the menu.

4.4 Changing

There are two data types whose instances either fill perfectly a need in the picture, or are totally extraneous. If such an item is not exactly what is desired, we must delete it. These types are the line and the connection.

On the other hand, if one is dissatisfied with a comment, terminal, or mnemonic, one may wish to change the instance rather than obliterate it entirely.

To alter an element, one must identify both a change he would like to take place and the item to be changed. To be consistent with previous parts of the system, we shall require the possible changes to the current data types to be listed as options in the menu area. A desired option will be selected as
it was in changing lines, and the instance the option shall be applied to will be indicated as in the delete activity.

4.4.1 Changing Mnemonics

There are two different ways in which a user could desire to change a mnemonic instance: modification of its default parameters and reorientation of the representative drawing. While it is easy to see that the first is required, one might question the need for or the feasibility of the second.

Suppose the user had defined an equivalent circuit for a NPN transistor and had represented it by the mnemonic shown in Figure 4.4.1-l. He might need later to use the same device with essentially the same mnemonic, except rotated clockwise through 90°. With no facility to allow reorientation of a mnemonic instance, the system would in effect require him to redraw this representation, which was adequate in all respects except orientation. In order to cover all non-oblique uses of this representation, the user would have to save in the system eight essentially identical mnemonics. Extrapolating this pattern, we see that the menu, during the add mnemonics state, would be clogged with many reorientations of only a few different elements.

Figure 4.4.1-l. A Circuit Element which can be Usefully Oriented in Eight Ways
It is definitely feasible to build into the system the capability of displaying any mnemonic, consisting of lines and terminals only,* rotated through 90°, 180°, or 270°, and reflected about its horizontal axis. To display a point in the representation whose co-ordinates are \((x,y)\) with respect to the mnemonic's center, and whose center is located at \((X,Y)\), the system must calculate the screen co-ordinates \((x+X, y+Y)\). To display the same point rotated 90° counterclockwise about the center, the system must generate the co-ordinates \((X-y, Y+x)\), which is no more and no less complex. The only extra processing involved is the decision how to calculate the screen co-ordinates. Also, the only extra information to be stored with the mnemonic instance data is three additional bits (to indicate one of eight different possible orientations).

The menu need contain only two options for reorientation of mnemonic instances. ROTATE would cause selected instances to be rotated through 90° counterclockwise from their current displayed position; REFLECT similarly would cause reflection about the horizontal axis. More options could, of course, could be added for convenience.

In addition to these options, one must be added to allow modification of an instance's parameters. This option could be labeled, for example, PARAM. Selecting it, then choosing a specific device, would allow modification of a parameter list whose prototype was created when the mnemonic representation was saved (see Section 5.1). This modification will be done using the text editor. Returning from the editor will, of course, leave the system in the same mode and activity, with PARAM still selected.

* The rotation of comments will be discussed in the Section 4.4.2.
4.4.2 Changing Comments

In view of our permitting rotation of mnemonic instances, it is certainly desirable that comment instances could similarly be reoriented. This would permit text to be included with a mnemonic drawing, a handy (though not crucial) feature that would otherwise be excluded.

There are two ways that "rotated" comment text might be legibly displayed. In the first, characters would be generated to appear erect when read from left to right either from the bottom or from the right side of the screen. This would demand that the hardware be capable of displaying any character either right side up or "lying on its back," which is not a common feature. The second would generate the characters in normal orientation, but to be read either from left to right or from top to bottom. Allowance would need to be made for the text's locating point being positioned differently in different orientations with respect to the typed string. In Figure 4.4.2-1 is an illustration of the problem. The word COMMENT is displayed in different orientations along with two lines. The locating point is shown as a dot.

This scheme, although more likely than the first, suffers from the fact that most hardwares do not display characters of equal height and width. Thus, a comment displayed vertically would appear elongated with respect to rotated lines.

The options presented for the reorientation of comments should, for compatibility, be identical to those for lines.

Another possible change allowed for comment instances would be a change in text. Whether or not we allow for such alterations in our software will depend mainly on our attitude. We may consider that each instance
<table>
<thead>
<tr>
<th>NORMAL</th>
<th>ROTATION</th>
<th>REFLECTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMENT</td>
<td>0°</td>
<td>COMMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>90°</td>
<td>C</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>COMMENT</td>
<td>180°</td>
<td>COMMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>270°</td>
<td>C</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

Figure 4.4.2-1. All Possible Orientations of Displayed Text
merely represents a display of a centrally stored text block. In this, changing the text would require a modification of this master comment. Such an alteration would be reflected in each instance. Although possible, it is not likely we would want the change to have such a scope.

On the other hand, we may consider that the central comment is merely a prototype for the copies which are added to the picture. It would be entirely likely that we would want to allow modification of such copies in the same way that we permit alteration of the instances of mnemonics' parameter lists.

In either case, if we accept changes to comment instances, the option should be specified—as all options—in the menu; and it should apply until another is selected.

4.4.3 Changing Terminals

There are three types of useful changes to terminals which might be programmed into the system. The first of these would allow alteration of a terminal's internal/external attribute. It could be convenient, for instance, if the behavior of an element were characterized by a network containing no isolated external terminals. Internal connection points could then be modified to external as needed.

The other two modifications would apply to the invisible serial number assigned to each new terminal added to a picture. One change would be to make this number appear or disappear. This is vital since a lot of terminal numbers would seriously clutter the picture, but some must be known so that equations concerning associated terminal variables may be written. It would be convenient as well to be able to change the number of a terminal; this, however, might be difficult to implement.
5. SPECIFICATION AND SYSTEM COMMANDS

In order to round out the network description of the graphics system, we must cover two isolated topics. Section 5.2 will deal with specifying circuit variables and equations involving them. First, let us complete the set of commands necessary to direct the system's large scale actions.

5.1 Miscellaneous System Directives

When the user first gains control over the graphics system, he will need to know what pictures (networks or mnemonics) are available for modification. This information will also be necessary after he has finished defining or modifying a network or mnemonic. Accordingly, a list of available pictures should be displayed on the screen whenever no such work is in progress. In the list, each network should be followed by a sublist of all associated mnemonics. (It is possible that more than one mnemonic could refer to a given network. For example, both a generator and a motor could be special cases of a transducer between electrical and mechanical energy.)

From this list, the user must be able to select pictures for editing. We will, therefore, provide two fetch commands—one for networks, the other for mnemonics. They might take the form \#FN networkname and \#FM mnemonicname where \# is a special symbol preceding all commands. Two separate commands are needed because a network and one of its associated mnemonics could easily have the same name.

The picture thus activated will not be specifically a network or mnemonic, regardless of its history. Instead, it will acquire a type only when it is saved. This will allow networks and mnemonics to be edited using exactly the same techniques, as well as permitting construction of a mnemonic from a previously existing network.
The commands for saving will be #SN networkname for networks, and #SM networkname, mnemonicname for mnemonics—the latter indicating both the mnemonic name and the associated network. (If mnemonicname is omitted, it can be taken as identical to networkname.) When a mnemonic is saved, some special processing should take place. The parameter list for the mnemonic should be taken from the PARAMETER block specified for the picture [2].

The picture description should be checked for the occurrence of data elements illegal in mnemonics (e.g. mnemonic and internal terminal instances). It should be further checked to verify that external terminals match in type and number. Finally, before saving either kind of picture, the system must verify that the picture name is unique. Should any conflicts exist, the saving of the picture should fail.

A final command is necessary to destroy the active picture (and return to the display of picture names), as well as to delete existing pictures so that they may be replaced. This command could be #DM mnemonicname, #DN networkname, or #D ACTIVE. Requiring destruction of these pictures explicitly will help prevent the loss of data through a blunder, such as returning control of the terminal to the system before saving the active picture, or defining a new picture with an old picture name.

The file of comments associated with each picture, although of narrower scope than the pictures themselves, should be handled with similar commands. To create a comment under a certain name, the user will type #SC commentname, where name is the title of the comment (see Section 4.1.1). The system will then enter the text editor to accept the body of the comment; upon return the comment with its title will be saved. Modification of the
comment will be affected by fetching it with #FC commentname—the user will again be placed in the text editor, return from which will save the updated comment. Finally, deletion could be accomplished by typing #DC <name>.

We could increase the convenience of adding comments to the picture if we made the #SC part of the SAVE COMMENTS command, optional. In addition, newly created comment prototype should become the current type to be added to the picture at a joystick hit. Thus, in the ADD COMMENT state, the user could type in the name of some comment text, followed (in the text editor) by the body, then immediately insert it any number of times in the picture with joystick hits.

A storage screen system will need to have some provision for the user to specify that the display be regenerated, especially if information is not always maintained current on the screen. And, any system will need a similar facility for the user to signal returns from subroutines (such as the text editor). Both could be implemented as joystick buttons or special keyboard keys; alternately, two light buttons always displayed in the same location on the screen could be used.

Finally, the system will need some way of switching between the draw state, the specify state, and the analysis state. This could best be implemented as were the indicators for mode and activity.

5.2 The Specify State

In this state the user will add to the physical construction of a network or device whatever textual information is necessary to describe its performance. The information will be divided into a series of blocks of text. Each block will have a title displayed in the menu. When a title is selected, the text editor will be entered to allow the user to input and
modify information in the block. When the user is not in the text editor, he may use any of the system commands described in Section 5.1.

Included in the information blocks will be: allowable terminal types in the picture which are not obtainable from some device contained in the network; global, local, and special variables; parameters of the network or mnemonic; and equations relating the variables. These will all be, except for the modifications listed below, identical to the description of them in [3]. Not described there will be three more blocks specifying indices, initial conditions, and display variables.

In the reference quoted above, all variables are required to be single-valued; no subscripts are allows. Whenever the user deals with tensor-valued quantities in a lumped system, this can represent a significant hardship. We would like to give him the power to deal with such quantities in abbreviated form, without forcing him to resort to a programming style language. To implement this, we will introduce variables which can take on only the integer values, from one to a specified upper limit. We shall call these indices. The list of indices, each with its upper value, will be specified by choosing the appropriate menu option, and typing lines of the form

variable name, upper bound

Whenever a variable can appear in the first six options, we shall allow it to be followed by a subscript list in the form

[subscript expression, subscript expression,...]

The subscript expressions will be any FORTRAN style integer expression involving constants and indices only. In declaration blocks, the expressions will be evaluated using the largest value for each index; the result will be the maximum subscript value that can appear in each position. (The minimum subscript allowed will be 1.)
Whenever a noninteger expression can occur in any block, we will allow two functions to be used in conjunction with the subscripted variables; \( S(\text{expression}, \text{index 1, index 2, ...}) \) will be taken to mean
\[
\sum_{\text{index 1}} \sum_{\text{index 2}} \ldots \text{expression}
\]
and \( P(\text{expression}, \text{index 1, index 2, ...}) \) shall mean
\[
\prod_{\text{index 1}} \prod_{\text{index 2}} \ldots \text{expression}
\]
The sums and products will be taken over all possible index combinations.

Similarly, when an equation is written involving free (i.e. not dummy) subscripts, the equality will extend over all values of the free subscripts. We will, of course, have to guard against meaningless expressions such as \( A[I] = B[J] \).

As an example of subscripts being used, under the terminal type definitions we could have
\[
\text{PHYSICAL:E}(X[I]), I(FX[I])
\]
under indices
\[
1, 2
\]
and under equations
\[
X[I](1) = X[I](2) \\
FX[I](1) + FX[I](2) = M*X[I](1)"
\]
This would be the same as defining under terminal types
\[
\text{PHYSICAL:E}(X,Y), I(FX,FY)
\]
and under equations
\[
X(1) = X(2) \\
Y(1) = Y(2) \\
FX(1) + FX(2) = M*X(1)"
\]
\[
FY(1) + FY(2) = M*Y(1)"
\]
Under the initial conditions option, the user would be able to specify equations which would apply at time $t = 0$ only. It is reasonable to be able to include such relations with a network, rather than specify them all at the time the network is analyzed, because this allows initial conditions to be obtained from parameter values. This could be useful in a device to be used in a more complex circuit.

Finally, in the display variables block, the user will list all variables which he wants recorded as the simulation of a network's behavior progresses. The values of all other variables will be discarded when they are no longer needed during automatic analysis. This should cut down on the masses of useless information generated in the course of a simulation.
6. CONCLUSIONS

We have presented here the main features of a graphics system for specifying multi-level network models. This system is intended to suggest improvements to or possibly replace the system described in [2] and currently in use.

Our new system eases some of the major bottlenecks of the old one. Retyping whole lines in order to add or delete a few characters is unnecessary. Also, eliminated is the user's redrawing of a circuit element in several different orientations; the system is able to display mnemonics in one of several positions. Furthermore, no longer is painstaking control of the joystick required to draw horizontal lines or to join two predefined points.

The system also contains significant improvements in overall organization. In drawing pictures, the necessary manipulations of data elements have been categorized, and within a category, all data types are treated similarly. There is no discrimination between mnemonics and networks as they are being created or modified; the difference appears only in the command to save the picture. Whenever possible, we have fit required descriptive actions into this structure rather than treating them as separate special cases. For example, a distinct procedure is not used for assigning a specific type to a terminal. Instead, it acquires its type as a natural conjunct of being added to a picture. All this tightened structure should make learning to use the new graphics system easier, and help to eliminate user blunders.

There are probably many useful additions which could be made to this system. One of these was outlined in Section 2.2. Another would be
to provide a mechanism for enlarging or contracting a picture, allowing for example, a mnemonic's drawing to fill the screen when it was being created, but displaying it at a reduced size when it was included in complicated networks. If this were combined with a feature allowing variable resolution of the co-ordinates of a point indicated by the joystick the user could access only widely-spaced points when drawing an enlarged mnemonic, but have greater resolution when a picture contained more detail. These last two additions would improve the quality of mnemonic drawings and decrease user fatigue. Any such changes or additions to the system will have to be carefully planned to preserve the order we have gained in system organization and avoid a patchwork effect.
LIST OF REFERENCES


**UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT**

(See Instructions on Reverse Side)

<table>
<thead>
<tr>
<th>1. AEC REPORT NO.</th>
<th>COO-1469-0214</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. TITLE</td>
<td>SOME PROPOSALS FOR MODIFYING THE GRASS GRAPHICS LANGUAGE</td>
</tr>
<tr>
<td></td>
<td>BY: Paul Eugene Niverville des trois Maisons</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. TYPE OF DOCUMENT (Check one):</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. Scientific and technical report</td>
</tr>
<tr>
<td>b. Conference paper not to be published in a journal:</td>
</tr>
<tr>
<td>Title of conference:</td>
</tr>
<tr>
<td>Date of conference:</td>
</tr>
<tr>
<td>Exact location of conference:</td>
</tr>
<tr>
<td>Sponsoring organization:</td>
</tr>
<tr>
<td>c. Other (Specify): Thesis Research</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. AEC's normal announcement and distribution procedures may be followed.</td>
</tr>
<tr>
<td>b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.</td>
</tr>
<tr>
<td>c. Make no announcement or distribution.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5. REASON FOR RECOMMENDED RESTRICTIONS:</th>
</tr>
</thead>
</table>

**SUBMITTED BY: NAME AND POSITION (Please print or type)**

C. W. Gear
Professor and Principal Investigator

Organization
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Signature
[Signature]

Date: December 1972

FOR AEC USE ONLY

AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

---

**PATENT CLEARANCE:**

- a. AEC patent clearance has been granted by responsible AEC patent group.
- b. Report has been sent to responsible AEC patent group for clearance.
SOME PROPOSALS FOR MODIFYING THE GRASS GRAPHICS LANGUAGE

Abstract

The purpose of this research is to outline a reconfiguration of the Graphical Remote Access Support System (GRASS) as developed at the University of Illinois. In this paper, we shall attempt to improve upon the graphical procedures of GRASS, while still fulfilling the original system goals. For this reason, familiarity with the current system is required.

Descriptors

- point hierarchy
- adding mnemonics, terminals, comments
- adding lines and connections
- changing mnemonics, terminals, comments
- system directives
- the specify state

OSA I Field/Group

unlimited distribution