INTRODUCTION

Faced with the problem of developing a multiterminal interactive graphics display system, analysis of past experience led to three specific problem areas which must be addressed in order to build a workable system. Not necessarily in order of importance or complexity these areas are (1) the difficulty in specifying interaction between the user and the computer, (2) the complexity of handling large quantities of graphic data and its interrelationship with the display hardware, and (3) the need to get away from assembly (or assembly type) languages so that one need not be a senior systems programmer in order to write a graphics program.

In past systems, the primary concern has been the handling of display images. The specification of interaction between the user and the computer has appeared of secondary importance if it was addressed at all. Unfortunately, a human being sitting in front of an array of keys, buttons, knobs, pens (both light and tablets) along with sets of lights, noise makers and the display itself cannot be programmed quite as cleanly as a card reader or magnetic tape handler. Experience has shown that attempts to control all the interactive devices available on most display systems leads to several months of “call-back debugging” to handle those unforeseen situations where “that key was supposed to be disabled.”

The problem of handling graphical data has led many people to the conclusion that some form of data structure is required to facilitate efficient utilization of the display. Closely related to this problem is the development of a true compiler-based language to support graphics efforts. To get valid applications for graphics systems, people who understand the application must write the programs. As long as systems programmers must be employed to write application programs, graphics will remain in the experimental, almost useful state. Current languages, either assembly or pseudo-

compiler, are either so complex or unnatural that applications programmers are driven away.

A detailed analysis of these problems evolved into a design for a multiterminal interactive graphics display system supported by a compiler level language. This design brings together a number of concepts which have been used before and several new proposals. It is not intended to be the answer to everyone’s problem, but from our experience it is a significant advance over what is generally available.

The system environment consists of an SEL 840MP processing unit with 64K of 24 bit core memory, movable head disk, multiple magnetic tape units, a card reader, a line printer and three modified SEL 816A graphics consoles. Interaction devices provided at each console are light pen, three shaft encoders, a bank of function switches with lamps independently programmable and an ASCII compatible keyboard. In addition to the user controlled interaction devices the system provides the application programmer with two special interaction aids. A clock pulse occurring at one second intervals is available to any interactive program. The application programmer should treat this interaction aid as an asynchronous “interrupt” since his program response is subject to system loading at execution time. The precise time of day is available should exact time be necessary to the application. The second special interaction aid is a program-to-program communication package. In essence one program may transmit a message to another, provided the latter has been enabled to accept the message. We feel this notify interaction capability is extremely powerful in a command and control environment.

To the system, jobs are classified either batch or interactive. Further, interactive jobs may be either graphics or non-graphics in nature. Batch jobs are the type most commonly supported by standard operating systems. An interactive job is one generated by the AIDS compiler and must contain at least one interaction
specification. A non-graphics interactive job is one which employs either the clock or notify capability. A number of non-graphics interactive jobs concurrent in a system provide considerable enrichment. Allocation of memory and files as well as execution priority are based on job classification.

Before describing the fundamental concepts employed by AIDS, the distinction between users, programmers, and systems development types must be pointed out. A user is the ultimate consumer of a graphics system and is expected to know the application being run on the display but understand nothing of the workings of the display itself. A programmer is a person well versed in the application and possessing reasonable knowledge about the graphics system and language. He should know only the compiler language (to require him to learn assembly language would reduce his effectiveness in his application specialty) and cannot be expected to understand or perform “tricks” with the display. A systems type is one who has very limited knowledge of the applications being developed but is thoroughly familiar with the details of the display system.

It is the user's responsibility to utilize the display system effectively. The programmer must develop programs which allow the user to concentrate on his application and to develop confidence that the machine is helping rather than opposing him. Ideally, after sitting at the console and logging on, the user should forget that he is directing a computer and be led in a natural way through his application with all his concentration and effort being directed at that application. The systems developer must devise a reliable operating system offering in "a reasonable manner" all of the capabilities of the hardware being used. To the extent that he succeeds, the programmer and user will be able to perform their jobs better.

This short divergence probably has validity in many areas, but when considered with the present state of graphics development it has particular truth. The complexities of graphic systems have caused the systems developers to concentrate on devising reliable, capable operating systems which unfortunately don't offer these capabilities in what programmers consider "a reasonable manner." Thus, most programmers are reluctant (at best) to undertake graphics programs and those that do, are so enmeshed in the system that they lose their close contact with the application they're programming. Ultimately the user suffers from lack of good application programs and develops the idea that graphics is just a cute toy.

AIDS LANGUAGE AND SYSTEM DESCRIPTION

The AIDS system deals with each of the three problem areas mentioned above. The Interactive Operating System provides a simple means of organizing and specifying user interaction at the terminal. The Graphic Structure Commands allow orderly development of complex display structures. The AIDS Pre-Compiler removes the programmer from assembly language problems while providing valuable bookkeeping and error detection functions.

The interactive specification problem

The major contribution of the AIDS development is a simplified means of specifying user-computer interaction. With the WHEN Interactive Specification Statement, the programmer details precisely what interactive devices should be enabled, when, and what is to be done if one of these is activated. The system was devised from analysis of the way a programmer designs the proposed user interaction with the computer. Consider a user at a graphics console. Before any pictures appear on the screen or lights flash etc., the program passes through an initialization phase. A picture is presented, certain of the interactive devices are enabled for the user's choice, and the program then pauses waiting for the user to decide what to do. Selecting one of the enabled devices causes a burst of computer activity, perhaps changing a picture, perhaps enabling a different set of devices, but eventually resulting in another pause where the user must make a selection. Looking at the interactive program as a whole, the process appears as a series of interrelated pauses and bursts of activity which can be described in a state table type notation. In the Appendix, Figure 4, states are represented by circles and conditions active in a state are represented by lines proceeding from one state to another. The conditions are written in quotes along with the responses to each condition (see example, Figure 4). This state table development is the process which a programmer, consciously or not, goes through in designing his application program. The extent to which the transitions between states are made in a natural way determines the effectiveness of the program to the user.

In the past, the specification and processing of all interactive devices, was handled entirely by the programmer, allowing uncertainty to develop as to which devices are active. AIDS uses the WHEN statement as an extension of the state table to specify interaction. The operating system then performs all enabling and disabling functions. For each condition in each state, a WHEN statement is written detailing the state, condition (interactive device), and the responses (what to do if that device is activated by the user). The program which results consists of a main program handling initialization of the necessary pictures, tables,
etc., followed by a series of statements derived from the state table and detailing what the programmer wishes to do.

Given this unusual program form, the AIDS pre-compiler maps each WHEN statement into a form usable by the operating system. In this way the system developer has provided a buffer between the complexities of his system and the desired simplicity which the programmer demands. The programmer has an easily analyzed and well organized specification of his thoughts which can readily be revised or expanded and the user has a far better chance of developing confidence in the equipment since there are fewer chances that it will fail to do what he commands.

**Description of WHEN statements**

The following is a summary of interactive control statements associated with the AIDS Operating System.

\[
\text{WHEN IN STATE } n, \text{ IF condition } a, \text{ THEN } \ldots \text{ response } \ldots
\]

This is the fundamental statement which is written directly from the programmer's state table. The pre-compiler offers considerable flexibility in that the programmer can make his statements as wordy or concise as he wishes, offering a self-documenting, easily readable program or a tight, easily coded form for quick preparation of test routines. The simplest form of the statement is:

\[
\text{WHEN } n, \text{ condition } a, \ldots \text{ response } \ldots
\]

Where \ldots \ "n\" is a positive integer corresponding to the arbitrary number assigned to each state in the state table. If a condition is active in a number of states, a list of the states can be given in place of the single state number. If a condition is active in all states, (an emergency panic button, for example) it can be declared to be in state "0" and will be active at all times.

\ldots a condition may be any of the interactive devices available on the terminal being used, or the clock or notify as mentioned earlier. Here again a long form of the condition is available for the finished documented program, but a short, easy to use form will also be recognized.

\ldots a response can consist of any single AIDS or Fortran statement or any sequence of statements excluding another WHEN statement. Since any statement can be included in a response translating the programmer’s state table into AIDS code is quite simple. Anything that can be done in the main program can also be done in a response.

**ENABLE STATE n**

This statement causes a particular state to be activated (i.e., the conditions specified in WHEN statements containing this state will now be searched for by the operating system). No further action is required by the programmer to activate interactive devices. Whenever he wished to enable a new set of devices, he enables another state.

**WAIT**

This statement needs background explanation. All of the routines which we developed for this system are written as reentrant code so that one copy will suffice for all terminals. However, the Fortran library is not reentrant and the code produced by the compiler has the same flaw. A serious problem develops if one allows the main program to be interrupted. If the response calls a library routine which the main program was in the process of executing, the main program's return will be destroyed along with any temporary storage locations. The WAIT statement was instituted to relieve this problem.

The ENABLE statement activates the conditions associated with a state but the main program will not be interrupted by satisfaction of a condition until a WAIT statement has been encountered. The effect of the WAIT command is to indicate that unless the user satisfies an enabled condition, the program needs no more CPU time. At the beginning of each time slice normally allocated to this program a check of the interactive devices is made, if no conditions were met, the time is allocated to another program. The program is reactivated as soon as an enabled condition is satisfied.

**ENDWAIT**

This statement is available in case the programmer would like to reactivate the main program to do more than its initialization role. An ENDWAIT statement executed in a response will cause the main program to restart following the WAIT statement at which it is currently stopped. In our experience this feature is a valuable one for developing parallel processes.

**CHECK**

The combination of WAIT-ENDWAIT doesn’t quite complete the picture. WAIT causes the program to relinquish control and needs an ENDWAIT to restart it. There are times in a long calculation process where the programmer would like to see if any conditions have
been satisfied without stopping the main program. CHECK provides this capability. Upon executing a CHECK the system will determine if any responses remain to be satisfied. If any exist they will be executed and then the main program continued; if not, the main program is continued immediately.

Graphical data management problem

In deciding to provide a means of graphical structure manipulation for AIDS, several considerations were involved. First, the programmer had to be relieved of the tedious problems of constructing display instruction files, and yet any capability which he had before must be available in the new system. Next, something more than the display subroutine capability offered by many hardware systems and echoed without improvement by software systems must be provided. Some reasonable means of building and organizing files in a logical and concise manner was needed. Third, a capability to link non-graphical data to the display structure must be available so that, for example, by selecting a circuit element on the screen with a light pen the programmer could easily determine the notation, component value, and other non-graphical data for that element.

The basic form of the graphical data structure which we chose is a variation on the GRIN Graphical Structure as described by Christensen. The elements of the structure described there satisfy most of the requirements mentioned above and offer the most natural, easy to use yet powerful structure which we had encountered. The basic elements which will be described here are directly related to the GRIN system and many of the operations which our system provides are similar to GRIN commands; however, we made no attempt to duplicate the elaborate dual processor (GE635 and PDP-9) program execution, or memory management schemes which are part of the GRIN design. In addition, a major difference between systems is the executable data structure which we implemented as opposed to the interpretive structure developed for GRIN. Their design has a PDP-9 dedicated to each display which they calculate is idle much of the time and therefore should be used if possible to speed display execution. The PDP-9 is used to interpret everything except the actual display code. In our design the main CPU is time sliced between three displays and several “background” jobs and isn’t available for file interpretation forcing us to devise an executable data structure.

Description of AIDS graphical data structure

The basic elements of the graphical data structure are the SET, INSTANCE, IMAGE, and LABEL blocks. An IMAGE is a collection of points, lines, and/or characters which is considered to be the most basic form of display entity. It is the only element which contains displayable code. An INSTANCE defines the occurrence of an IMAGE. Each time an image is displayed on the screen it is specified and positioned by means of an INSTANCE. A SET is a collection of INSTANCES whose occurrence can again be defined by another INSTANCE. An elaborate “tree-like” structure of these basic elements can be developed describing many applications in an organized manner and allowing quick retrieval of information at any level of the structure.

A trivial example of the structure and a convenient way of diagramming it (as specified by Christensen) is given here. Consider the picture of a house in Figure 1. The typical non-structured display file for this picture might describe each element in a sequence of instructions or subroutines containing instructions. Figure 2 shows a possible AIDS data structure representation of this picture which is not significantly improved over the subroutine description method. However, Figure 3 illustrates a much more complicated representation of the picture organized by the programmer according to his particular need to retrieve specific information at the various data structure levels. Note on the diagrams the representation of IMAGES as rectangles containing a drawing of what the IMAGE will produce on the screen, INSTANCES as lines connecting SETS with IMAGES or other SETS (an INSTANCE defines the occurrence of a SET or IMAGE) and SETS as circles which collect together INSTANCES at various levels. The illustrated
structure shown in Figure 3 is exaggerated but for many complex applications this type of multi-level structure is vital.

The LABEL block is used to store non-graphical data associated with any graphical element. It could be attached to each of the INSTANCES defining occurrences of the IMAGE WINDOW in Figure 3, stating the sash dimensions of each window. The organization of data within a LABEL block is entirely up to the programmer, the system provides a means of entering and retrieving data and of associating the block with any graphical element. As a follow-on to the AIDS system, a list processing capability could be implemented to improve the handling of these blocks. Examples of AIDS Structure Manipulation Statements are given in the next section.

Compiler level language requirement

The pre-compiler idea has been used on a number of systems and we feel it is a valuable tool in providing the flexibility which an interactive graphics programming system requires. An alternative is to write a complete compiler and unless significant resources are available this approach should be undertaken with considerable caution. Fortran is a good computational language but is not well suited to either interactive specification or graphical manipulation, therefore we decided to maintain the algebraic qualities of Fortran and let the pre-compiler handle all interactive and graphic statements. Examples of the AIDS statements as given here and in the Appendix exhibit little similarity to Fortran and in particular are designed to be readable by a non-programmer.

We have already discussed the Interactive Specification Statement. The pre-compiler extracts the necessary information from these commands and passes it to the operating system. The Fortran compiler was not modified to handle interactive capabilities. The following examples show how the AIDS graphic structure manipulation statements are constructed. First the SET, INSTANCE, and IMAGE associations are specified by:

```
SET ALPHA .CONTAINS. INSTANCE BETA
INSTANCE BETA .DEFINES. IMAGE GAMMA
```

where ALPHA, BETA, and GAMMA are graphic elements declared at the beginning of the program. The descriptors SET, INSTANCE, and IMAGE are optional; the verbs .CONTAINS. and .DEFINES. determine the nature of the association and the pre-compiler further checks to be sure the elements on each side of the verbs are of the proper type. This is a valuable service which the pre-compiler furnishes since a type error undetected here can cause multiple errors later. A picture can be specified as a series of individual statements or as a concatenation in one single statement. Thus the structure in Figure 2 can be specified as:

```
SET PICTURE .CONTAINS. INSTANCE A
*.DEFINES. IMAGE TREE
*.AND. INSTANCE B
*.DEFINES. IMAGE
*.AND. INSTANCE C
*.AND. C(1) .DEFINES. WINDOW
*
*
```

Building graphical data within an IMAGE is done with the INSERT command. The following are typical examples:

```
INSERT INTO IMAGE PICTURE: A LINE FROM 100, 200, TO X, Y
INSERT INTO PICTURE: TEXT *THIS IS AN EXAMPLE*
INSERT: FORMAT 100 AT IX, IY/(ARRAY (I), I = 1, 25)
```

Other capabilities include:

Detaching any element from another
Clearing or copying an IMAGE
Positioning or determining the position of an INSTANCE
Entering and Fetching Data from a LABEL Block and associating it with any graphic element
Showing (causing to be displayed) any SET and all structure below that SET
Destroying any element not currently needed to conserve core
Many of the functions of the AIDS pre-compiler could have been bypassed by allowing simple calls to be added to the Fortran input deck. However, the interactive specification tables, the structure generation statements, the compile time diagnostics, and the ease of compiler maintenance make the pre-compiler concept attractive.

CONCLUSIONS

The principal programmer complaint at the present time is the slow response time. Some of this can be attributed to the generalization of the system functions and to the currently untuned status of the overall system. Another complaint concerns the lack of editing facilities at the image level. Sufficient "handles" have been designed into the system to address this problem but as yet no effort has been made to specify the functions. A possible weakness concerns the fact that the system provides no queuing of interrupts with associated user control of priorities.

On the positive side, a number of programmers with limited experience have been writing reasonably complex interactive graphics applications in a matter of days rather than the weeks previously required.

REFERENCES

1 N A BALL H Q FOSTER W H LONG
I E SUTHERLAND R L WIGINGTON
A shared memory computer display system
IEEE Transactions on Electronic Computers Vol EC15
No 5 October 1966
2 I E SUTHERLAND
Sketchpad: A man-machine graphical communications system
Proceedings of the 1963 Spring Joint Computer Conference
3 S T WALKER
A study of a graphic display computer—Time sharing link
Masters Thesis Electrical Engineering Department
University of Maryland College Park June 1968
4 C CHRISTENSEN E N PINSON
Multi-function graphics for a large computer system
Proceedings of Fall Joint Computer Conference 1967
5 R G LOOMIS
A design study on graphics support in a fortran environment
Proceedings of Third Annual SHARE Design Automation Workshop New Orleans La May 1966
6 W M NEWMAN
A system for interactive graphical programming
Proceedings of the 1968 Spring Joint Computer Conference
7 W M NEWMAN
A high-level programming system for a remote time-shared graphics terminal

S R L WIGINGTON
Graphics and speech computer input and output for communications with humans
Computer Graphics Utility/Production/Art Thompson Book Company

APPENDIX

As an example of an AIDS program, we have selected a problem that everyone is familiar with: (1) draw an object on the screen, (2) position it wherever desired, and (3) be able to delete it from the screen. Figure 4 illustrates the State Diagram of the action which this program is to perform. Table I lists the Interaction Requirements which are derived from the State Diagram. With the help of these figures, the WHEN Interactive statements needed for this program are easily written. Figure 5 illustrates the display element structure which is developed by the program.

Initially all display data elements which are going to be used in the program must be declared. There will be one SET, four individual INSTANCES and one array of 20 INSTANCES, and a similar number of IMAGES. In addition certain Fortran variables are declared Type INTEGER. The first relational statement creates an occurrence (INSTANCE CROSS) of the IMAGE TRACK. The next two INSERT commands create a
TABLE I—Interaction Requirements for the Example

<table>
<thead>
<tr>
<th>STATE</th>
<th>CONDITION</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DRAW Light Button</td>
<td>Add an INSTANCE and IMAGE to the display. Position the Tracking Cross at 500,500 on the screen and enable State 2.</td>
</tr>
<tr>
<td>1</td>
<td>MOVE Light Button</td>
<td>Enable Light Pen for selection of object to move and enable State 3.</td>
</tr>
<tr>
<td>1</td>
<td>ERASE Light Button</td>
<td>Enable the Light Pen for selection of object to be erased.* Enable State 4.</td>
</tr>
<tr>
<td>Draw Function</td>
<td>Button 1 L</td>
<td>Get starting point of curve to be drawn from KNOB 1 and KNOB 2; OLDX = KNOB1; OLDY = KNOB2.</td>
</tr>
<tr>
<td>Move Function</td>
<td>KNOB1 or KNOB2</td>
<td>Position Tracking Cross according to current counts of KNOB1 and KNOB2; CURX = KNOB1; CURY = KNOB2.</td>
</tr>
<tr>
<td></td>
<td>Button 2 L</td>
<td>Add a line segment to the current IMAGE from OLDX, OLDY to CURX, CURY; OLDX = CURX, OLDY = CURY.</td>
</tr>
<tr>
<td></td>
<td>Button 3 L</td>
<td>Curve drawing is complete. Remove Tracking Cross from screen.</td>
</tr>
<tr>
<td></td>
<td>LPWLB</td>
<td>Set up for next IMAGE. Enable State 1.</td>
</tr>
<tr>
<td></td>
<td>Button 4 L</td>
<td>Set up for next IMAGE. Enable State 1.</td>
</tr>
<tr>
<td>Erase Function</td>
<td>NLB</td>
<td>Erase object selected.</td>
</tr>
<tr>
<td></td>
<td>Button 24 L</td>
<td>End wait mode. Job is complete.</td>
</tr>
</tbody>
</table>

* Note: User may not erase the Light Buttons ‘MOVE,’ ‘DRAW’ or ‘ERASE,’ however, movement of these Light Buttons is allowed.

cross in IMAGE TRACK, the occurrence of which is positioned at 500,500 by the POSITION command. The next six commands similarly create occurrences of the words DRAW, MOVE and ERASE. The Fortran variable I, used to control allocation of images created by the DRAW function, is set to 1. The next command causes SET PICTURE to be displayed. You will note that nothing will appear on the screen since nothing is yet attached to PICTURE and SETS contain no display ‘ink.’ Enable State 1 causes the system to activate those interactive devices defined in State 1. All other devices are inactive and will not burden the system with wasted interrupt processing. The WAIT command notifies the system that the main program no longer needs the CPU. As soon as a condition is met for the current state the program will be given CPU time and execution will begin at the first command of the response corresponding to the condition met.

There are three active conditions in State 1: LB DRAW, LB MOVE, LB ERASE. LB is an AIDS mnemonic for Light Button. LB DRAw notifies the system that INSTANCE ‘DRAW’ is a light button and that whenever State 1 is enabled, the system should display this Light Button. Thus when State 1 is enabled the words DRAW, MOVE, and ERASE will appear in the upper right corner of the screen. Further selection of one of these Light Buttons will cause the corresponding response to be executed.

Selecting Light Button ‘DRAW’ causes INSTANCE CROSS to appear on the screen and INSTANCE BRANCH (1) to be attached to PICTURE. Only the cross will be seen since no ink has been inserted into LEAF (1). Next State 2 is enabled, the KNOB counts for KNOB 1 and KNOB 2 are initialized to 500,500 and the response is complete. Selecting Light Button ‘MOVE’ turns on the Light Pen for Set Picture and enables State 3.

In States 2 and 3 rotation of either KNOB 1 or KNOB 2 will update CURX and CURY with the counts at KNOB 1 and KNOB 2 respectively.

Conditions of the form “if button n L” cause the system to turn on the lamp associated with button n
### TABLE II—DRAW-MOVE-ERASE Program

**PROGRAM DRAW**

C THIS IS AN AIDS PROGRAM WHICH ALLOWS ONE TO DRAW FIGURES, MOVING THEM, AND DELETE THEM FROM THE SCREEN

C **DECLARATION STATEMENTS**

SET PICTURE

INSTANCE CROSS, DRAW, MOVE, ERASE, BRANCH (20)

IMAGE TRACK, DRAWL, MOVEL, ERASEL, LEAF (20)

INTEGER OLDX, OLDY, CURX, CURY

C **INITIALIZATION**

INSTANCE CROSS. DEFINES. IMAGE TRACK

INSERT INTO IMAGE TRACK: A LINE FROM -10,0, TO +10,0

INSERT: LINE 0, -10,0, +10

POSITION INSTANCE CROSS, AT 500,500

INSERT DRAWL: TEXT *DRAW*, 900,900

INSERT MOVEL: TEXT /MOVE/, 900,850

INSERT ERASEL: TEXT XERASEX, 900,800

INSTANCE DRAW. DEFINES. IMAGE DRAWL

MOVE. DEFINES. MOVEL

ERASE. DEFINES. ERASEL

I = 1

SHOW SET PICTURE

ENABLE STATE 1

C **INTERACTION STATEMENTS**

C **DRAW FUNCTION**

WHEN IN STATE 1, IF LB DRAW, THEN

SET PICTURE. CONTAINS. INSTANCE CROSS

AND. INSTANCE BRANCH (I)

. DEFINES . LEAF (I)

ENABLE STATE 2

PUT 500,500 IN KNBCNT

ENDRESPONSE

C **UPDATE TRACKING CROSS**

WHEN IN STATES 2,3, IF KNOB1/KNOB 2, THEN

GET KNBCNT INTO CURX, CURY

POSITION INSTANCE CROSS AT CURX, CURY

ENDRESPONSE

C **SETUP STARTING POINT OF LINE**

WHEN IN STATE 2, IF BUTTON 1 L, THEN

GET KNBCNT INTO OLDX, OLDY;

C **ADD A LINE SEGMENT FROM LAST ENDPOINT**

WHEN IN STATE 2, IF BUTTON 2 L, THEN

INSERT INTO IMAGE LEAF (I): A LINE FROM OLDX, OLDY, TO CURX, CURY

OLDX = CURX

OLDY = CURY;

C **CURVE COMPLETE, GO BACK TO INITIAL STATE**

WHEN IN STATE 2, IF BUTTON 3 L, THEN

DETACH CROSS

I = I+1

ENABLE STATE 1;
TABLE II—(Continued)

<table>
<thead>
<tr>
<th></th>
<th>MOVE FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>WHEN IN STATE 1, IF LB MOVE, THEN</td>
</tr>
<tr>
<td></td>
<td>SETUP SET PICTURE: PENON</td>
</tr>
<tr>
<td></td>
<td>ENABLE STATE 3;</td>
</tr>
<tr>
<td>C</td>
<td>LIGHT PEN ENABLED, NOW SELECT OBJECT TO MOVE</td>
</tr>
<tr>
<td></td>
<td>WHEN 3, IF LPH,</td>
</tr>
<tr>
<td></td>
<td>GET INSPEN INTO DUMMY</td>
</tr>
<tr>
<td></td>
<td>GET INSPOS OF INSTANCE DUMMY INTO X, Y</td>
</tr>
<tr>
<td></td>
<td>PUT X, Y INTO KNBCNT</td>
</tr>
<tr>
<td></td>
<td>POSITION CROSS, AT X, Y</td>
</tr>
<tr>
<td></td>
<td>SET PICTURE . CONTAINS. INSTANCE CROSS;</td>
</tr>
<tr>
<td>C</td>
<td>CROSS MOVED TO DESIRED LOCATION, NOW MOVE OBJECT</td>
</tr>
<tr>
<td></td>
<td>WHEN 3, IF BUTTON 4 L,</td>
</tr>
<tr>
<td></td>
<td>POSITION DUMMY AT CURX, CURY</td>
</tr>
<tr>
<td></td>
<td>ENABLE STATE 1;</td>
</tr>
<tr>
<td></td>
<td>DETACH CROSS;</td>
</tr>
<tr>
<td>C</td>
<td>ERASE FUNCTION</td>
</tr>
<tr>
<td></td>
<td>WHEN 1, LB ERASE, SETUP SET PICTURE: PENON</td>
</tr>
<tr>
<td></td>
<td>ENABLE STATE 4;</td>
</tr>
<tr>
<td>C</td>
<td>CHOSE OBJECT TO BE DELETED</td>
</tr>
<tr>
<td></td>
<td>WHEN 4, NLB,</td>
</tr>
<tr>
<td></td>
<td>GET INSPEN INTO DUMMY</td>
</tr>
<tr>
<td></td>
<td>DESTROY DUMMY</td>
</tr>
<tr>
<td></td>
<td>GET IMG PEN INTO DUMMY</td>
</tr>
<tr>
<td></td>
<td>DESTROY DUMMY</td>
</tr>
<tr>
<td></td>
<td>ENABLE STATE 1;</td>
</tr>
<tr>
<td>C</td>
<td>PANIC BUTTON</td>
</tr>
<tr>
<td></td>
<td>WHEN IN STATE φ, IF BUTTON 24 L, THEN ENDWAIT;</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

when that state is enabled as well as specify an interaction requirement.

"DETACH CROSS" removes the cursor from the screen.

Condition LPH in State 3 specifies that Light Buttons should be treated as normal display entities. That is, the Light Buttons as well as the constructed display objects may be moved with the move function.

Condition NLB in State 4 specified that the light pen is enabled but Light Buttons are not to be displayed so they will not inadvertently be erased.

Destroy DUMMY frees up the memory occupied by object DUMMY.

Finally the State φ response specifies that the main program is to continue. The main program will execute the Fortran STOP function. A condition in State φ is active in all states and so button 24 defines our panic mode exit.